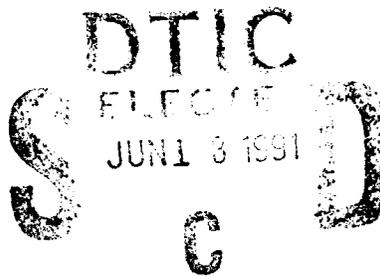


AD-A236 796



RL-TR-91-62  
Final Technical Report  
May 1991



2

# FIRST-ORDER GRADIENT DESCENT TRAINING OF ADAPTIVE DISCRETE-TIME DYNAMIC NETWORKS

Stanford University

Stephen W. Piche and Bernard Widrow

Approval For	
DTIC Copy	<input checked="" type="checkbox"/>
DTIC Plus	<input type="checkbox"/>
Unannounced	<input type="checkbox"/>
Justification	
By	
Distribution	
Availability Codes	
Dist	Avail and/or special
A-1	



APPROVED FOR PUBLIC RELEASE; DISTRIBUTION UNLIMITED.

91-01817



Rome Laboratory  
Air Force Systems Command  
Griffiss Air Force Base, NY 13441-5700

91 6 11 096

This report has been reviewed by the Rome Laboratory Public Affairs Office (PA) and is releasable to the National Technical Information Service (NTIS). At NTIS it will be releasable to the general public, including foreign nations.

RL-TR-91-62 has been reviewed and is approved for publication.

APPROVED: *Richard N. Smith*

RICHARD N. SMITH  
Project Engineer

APPROVED: *John A. Graniero*

JOHN A. GRANIERO  
Technical Director  
Directorate of Communications

FOR THE COMMANDER: *Billy G. Oaks*

BILLY G. OAKS  
Directorate of Plans & Programs

If your address has changed or if you wish to be removed from the Rome Laboratory mailing list, or if the addressee is no longer employed by your organization, please notify RL(DCCR) Griffiss AFB NY 13441-5700. This will assist us in maintaining a current mailing list.

Do not return copies of this report unless contractual obligations or notices on a specific document require that it be returned.

# REPORT DOCUMENTATION PAGE

Form Approved  
OMB No. 0704-0188

Public reporting burden for this collection of information is estimated to average 1 hour per response, including the time for reviewing instructions, searching existing data sources, gathering and maintaining the data needed, and completing and reviewing the collection of information. Send comments regarding this burden estimate or any other aspect of this collection of information, including suggestions for reducing this burden, to Washington Headquarters Services, Directorate for Information Operations and Reports, 1215 Jefferson Davis Highway, Suite 1204, Arlington, VA 22202-4302, and to the Office of Management and Budget, Paperwork Reduction Project (0704-0188), Washington, DC 20503.

1. AGENCY USE ONLY (Leave Blank)		2. REPORT DATE May 1991	3. REPORT TYPE AND DATES COVERED Final Apr 89 - Sep 90	
4. TITLE AND SUBTITLE FIRST-ORDER GRADIENT DESCENT TRAINING OF ADAPTIVE DISCRETE-TIME DYNAMIC NETWORKS			5. FUNDING NUMBERS C - F30602-88-D-0025 TASK - C-9-2404 PE - 62702F PR - 4519 TA - 2P WU - P5	
6. AUTHOR(S) Stephen W. Piche, Bernard Widrow			8. PERFORMING ORGANIZATION REPORT NUMBER	
7. PERFORMING ORGANIZATION NAME(S) AND ADDRESS(ES) Stanford University Department of Electrical Engineering Stanford CA 94305			10. SPONSORING/MONITORING AGENCY REPORT NUMBER RL-TR-91-62	
9. SPONSORING/MONITORING AGENCY NAME(S) AND ADDRESS(ES) Rome Laboratory (DCCR) Griffiss AFB NY 13441-5700			11. SUPPLEMENTARY NOTES Rome Laboratory Project Engineer: Richard N. Smith/DCCR/(315)330-3091	
12a. DISTRIBUTION/AVAILABILITY STATEMENT Approved for public release; distribution unlimited.			12b. DISTRIBUTION CODE	
13. ABSTRACT (Maximum 200 words)  This paper describes the training of discrete-time dynamic systems with adaptive parameters (recurrent neural networks) using first-order gradient descent algorithms. To facilitate the explanation of these algorithms, a standard representation of a discrete-time dynamic system is defined. Any differentiable discrete dynamic system may be put in this standard representation and trained using a gradient descent algorithm. Using the standard representation, we described two general types of learning algorithms. The first is based upon the discrete-time Euler-Lagrange equations, and the second is based upon a recursive update of the output gradients. Both the epochwise and on-line versions of these algorithms are presented. When the dynamic system is implemented by a neural network, the epochwise algorithm based on the Euler-Lagrange equations is equivalent to backpropagation-through-time and the on-line method based on the recursive equation is the same as recursive backpropagation. It is shown that the epochwise versions of the algorithms are equivalent. The two on-line versions of the algorithms are shown to be approximately equivalent. Because of the equivalence of the algorithms, selection of an appropriate gradient descent algorithm is based solely upon computational efficiency and storage requirements. Accordingly,  (continued)				
14. SUBJECT TERMS Neural Nets Algorithm			15. NUMBER OF PAGES 40	
			16. PRICE CODE	
17. SECURITY CLASSIFICATION OF REPORT UNCLASSIFIED	18. SECURITY CLASSIFICATION OF THIS PAGE UNCLASSIFIED	19. SECURITY CLASSIFICATION OF ABSTRACT UNCLASSIFIED	20. LIMITATION OF ABSTRACT U/L	

Block 13 (Continued)

a discussion of these two properties of the algorithms is included. To illustrate the differences between the algorithms and the usefulness of the standard representation, two examples are included.

# 1 Introduction

The ability of humans to control and interact with a complex environment has motivated our study of adaptive discrete-time dynamic systems which are fully or at least partially composed of neural networks. Humans regularly perform certain tasks easily and proficiently which have proven difficult to reproduce with machines. Examples of such tasks include driving a car, recognizing the differences between a cat and a dog, and understanding spoken language. Humans learn how to accomplish these tasks in part by interacting with, manipulating and eventually controlling their environment. In order to build machines which accomplish these difficult tasks, it may be necessary both to mimic humans learning through environmental interaction and to model the low level functions of the human nervous system. The algorithms presented in this paper provide one possible technique for accommodating both of these requirements. These algorithms train neural networks, which model the low level functions of the human nervous system, by interacting with a user-specified environment.

An adaptive dynamic system which is composed of a neural network and a set of equations which describe the environment may be used as an adaptive model of the interaction of a human with its environment. In this model, the neural network receives the state of the environment as input. Using the state in combination with a desired goal, the neural network outputs a control signal which manipulates the environment. Using the training algorithms presented in this paper, system performance often becomes strongly human-like. The first example of Section 9 gives a prime example of this phenomenon.

Although the human ability to control its environment motivates us, our primary interest lies in the development of engineering tools rather than in the modeling of humans. The training of discrete-time dynamic systems has applications in the engineering fields of pattern recognition, nonlinear control, adaptive control and adaptive digital filtering. It is in these areas that the material in this paper will be of most immediate use.

Currently, the theory on first-order gradient descent training of adaptive discrete-time dynamic networks is described in separate and unrelated terms in several papers by different authors [1,2,3,4,5,6]. Our goal is both to bring together in a coherent manner and to expand the theory on this subject. A coherent presentation of the subject is achieved by deriving the learning algorithms using a standard representation of a dynamic system. The derivation of the generalized forms of the existing algorithms provides for extensions of current algorithms. Bringing together and generalizing the theory in this manner should facilitate the selection of appropriate gradient descent algorithms for problems requiring discrete-time recurrent networks. It should be noted that a forthcoming paper by Williams and Zipser [7] contains a detailed discussion of adapting dynamic neural networks, whereas, this paper presents the concepts of first-order gradient descent for systems composed both fully or partially of neural networks.

The paper is composed of ten sections. Section 2 introduces notation and the standard representation for a discrete-time dynamic system. Section 3 presents the differences between on-line and epochwise training. An introduction to gradient descent training of static systems is given in Section 4. Section 5 presents the algorithms used for adapting discrete-time dynamic systems. The equivalence of the algorithms is presented in Section 6. A comparison of the computational and storage requirement of the algorithms is included in Section 7. Two techniques of speeding-up the on-line training algorithms are discussed in Section 8. Section 9 presents two applications which illustrate the usefulness of the theory discussed in the paper, and Section 10 provides a conclusion.

## 2 System Definition

In this section, a standard representation of any discrete-time dynamic system is proposed. This representation is used in the derivation of the learning algorithms. In addition, the ordered derivative, which simplifies the calculation of derivatives of complex systems, is introduced.

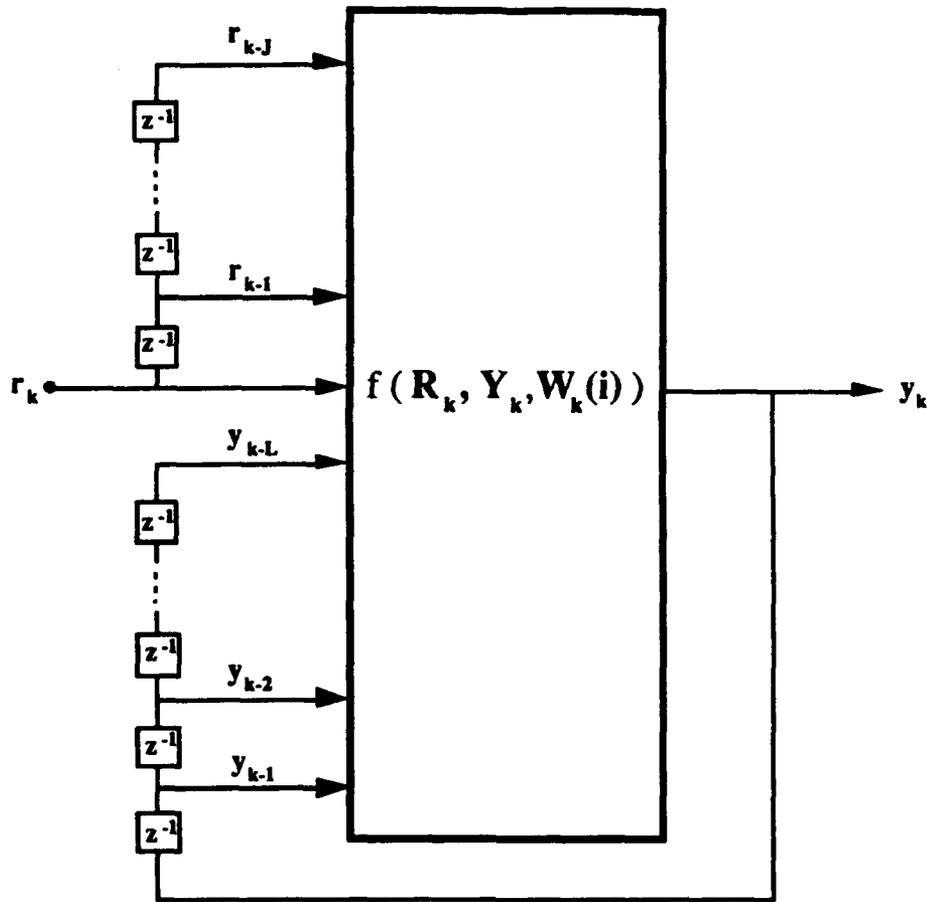


Figure 1: Standard Representation.

## 2.1 The Standard Representation

Let  $k$  denote the iteration of a discrete-time dynamic system, with  $k = 0$  representing the first iteration of the system. A standard representation of discrete-time dynamic system is illustrated in Figure 1. The input of the system at iteration  $k$  is defined by two components. The first component,  $\mathbf{R}_k$ , is composed of an external input vector,  $\mathbf{r}_k$ , and the previous  $J$  delayed versions of this vector,  $\mathbf{r}_{k-1}, \dots, \mathbf{r}_{k-J}$ . The external input vector is a length  $M$  column vector,  $\mathbf{r}_k \in R^{[M \times 1]}$ . Therefore, the external inputs to the system including delayed inputs is a length  $(J + 1)M$  column vector of the form  $\mathbf{R}_k = [\mathbf{r}_k^T, \mathbf{r}_{k-1}^T, \dots, \mathbf{r}_{k-J}^T]^T \in R^{[(J+1)M \times 1]}$ . The second component,  $\mathbf{Y}_k$ , is made up of the previous  $L$  output vectors. The output vector at iteration  $k$  is a length  $N$  column vector,  $\mathbf{y}_k \in R^{[N \times 1]}$ . Therefore,  $\mathbf{Y}_k$  is a length  $LN$  column vector of the form  $\mathbf{Y}_k = [\mathbf{y}_{k-1}^T, \mathbf{y}_{k-2}^T, \dots, \mathbf{y}_{k-L}^T]^T$ .

At iteration  $k$ , let the adaptive parameter vector, which we shall refer to as the weight vector, be selected from a set of weight vectors. In general, this set of weight vectors is generated by the training algorithm as discussed in Section 3. Assuming a weight vector to be a length  $Q$  column vector, the weight set  $\mathbf{W}$  has the form  $\mathbf{W} = \{\mathbf{W}(0), \mathbf{W}(1), \dots, \mathbf{W}(i), \dots\}$ . The use of the  $i^{\text{th}}$  weight vector at the  $k^{\text{th}}$  iteration shall be denoted  $\mathbf{W}_k(i)$ . Finally, let  $w_k(i)$  denote any weight of the vector  $\mathbf{W}_k(i)$ . Of course,  $w_k(i)$  is a scalar.

By denoting any element of a discrete-time dynamic system which is connected to a delay as an output and including it in  $\mathbf{y}_k$ , any discrete-time system can be written as

$$\mathbf{y}_k = f(\mathbf{R}_k, \mathbf{Y}_k, \mathbf{W}_k(i)) \quad (1)$$

where the function  $f$  contains no delay. Thus the schematic representation in Figure 1 can be used to

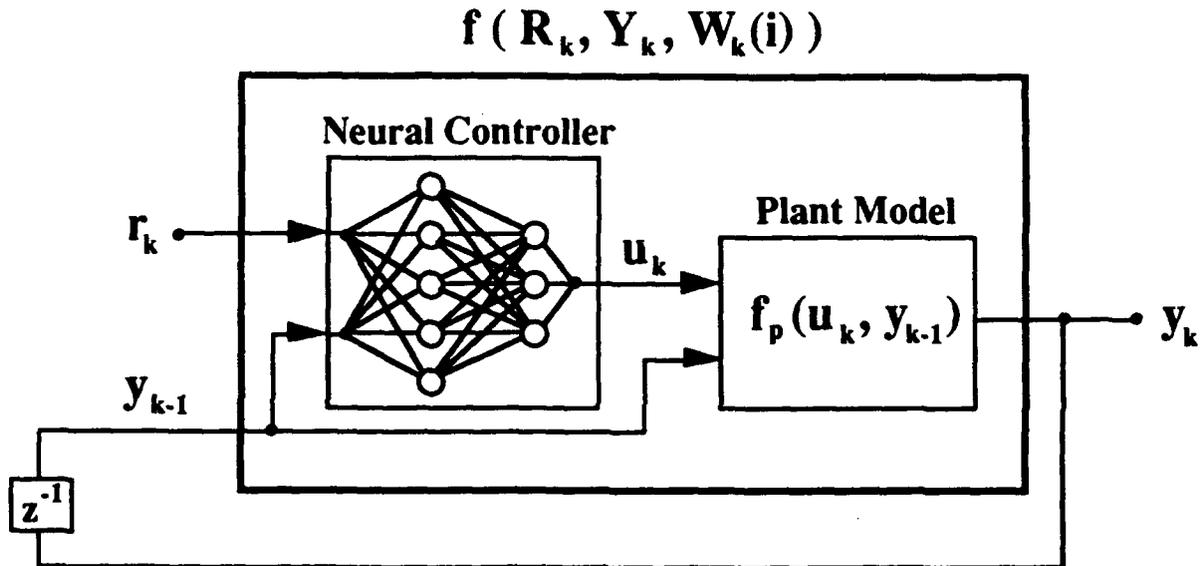


Figure 2: Neural Controller and Plant in the Standard Representation.

describe any discrete-time dynamic system. The first-order gradient descent algorithms to be described in this paper are defined in terms of the standard representation shown in Figure 1. Because any adaptive discrete-time dynamic system can be put in the standard representation, such systems can be trained using the algorithms described in this paper provided that the output vector,  $y_k$ , is differentiable with respect to the weight vector,  $W_k(i)$ , and the recurrent input,  $Y_k$ . The need for this requirement will become evident in Section 5.

## 2.2 An Example of a System in the Standard Representation

In order to illustrate the use of the standard representation, an adaptive discrete-time dynamic system which consists of a neural network and an environment model, is shown schematically in Figure 2. In this figure, we use the traditional term *plant model* instead of *environmental model*. We define the dynamic system of Figure 2 to be a neural network controller-plant system.

Because we often use the controller-plant system to illustrate important points throughout this paper, it is useful at this point to discuss this system in greater detail. The plant model may take two different forms. The most general is simply a set of equations which map previous states and current control to the next state,  $y_k = f_p(u_k, Y_k)$ , where  $Y_k$  are the previous states of the plant and  $u_k$  is the control signal vector. If the equations of the model are nonlinear, adapting the structure of Figure 2 results in a neural controller which implements a nonlinear state feedback control law. This technique provides a method of designing a nonlinear controller for a nonlinear plant.

Using the second form of a plant model, a neural network model, has applications in the field of adaptive control [3,5,8]. In this case, the plant model takes the form  $y_k = f_p(u_k, Y_k, W^p)$ , where  $W^p$  is the weight vector of the neural network model. This model can be updated on-line using plant input-output data. Because the algorithms presented in this paper use the plant model to update the controller, the neural network controller can adapt to changes in the plant.

## 2.3 Ordered Partial Derivatives

Because we are interested in using first-order gradient descent to update the adaptive weights, we are required to calculate a partial derivative of the associated dynamic system. The ordered partial derivative, which is a special partial derivative for an ordered set of equations, provides a mathematical technique for easily

finding derivatives of complex dynamic systems [6].

In order to discuss the ordered derivative, we must first introduce the concept of an ordered set of equations. Let  $\{z_1, \dots, z_i, \dots, z_j, \dots, z_n\}$  be a set of  $n$  variables whose values are determined by a set of  $n$  equations. This set of equations is defined to be an ordered set of equations if each variable  $z_i$  is a function only of the variables  $\{z_1, \dots, z_{i-1}\}$ . Thus, the equation for any variable of an ordered set of equations can be written as

$$z_i = f(z_1, \dots, z_{i-1})$$

Because of the ordered nature of this set of equations, the variables  $\{z_1, \dots, z_{i-1}\}$  must be calculated before  $z_i$  can be computed.

When calculating the partial derivative of a variable it is necessary to specify which variables are held constant and which are allowed to vary. Typically, if this is not specified, it is assumed that all variables are held constant except those terms appearing in the denominator of the partial derivative. This is the convention we have adopted in this paper.

The ordered partial derivative, which is defined only for variables of an ordered set of equation, is a partial derivative whose constant and varying terms are determined using the ordered set of equations. The constant terms of the order partial derivative of  $z_j$  with respect to  $z_i$ , which is denoted  $\partial^+ z_j / \partial z_i$  in order to distinguish it from an ordinary partial derivative, are  $\{z_1, \dots, z_{i-1}\}$ . The varying terms are  $\{z_i, \dots, z_j, \dots, z_n\}$ .

The ordered derivative is usually found using either of two chain rule expansions. The first expansion, which is expressed in vector form as

$$\frac{\partial^+ z_j}{\partial z_i} = \frac{\partial z_j}{\partial z_i} + \sum_{k=i}^j \frac{\partial^+ z_j}{\partial z_k} \frac{\partial z_k}{\partial z_i} \quad (2)$$

was shown by Werbos in his thesis [6]. The proof of the second chain rule expansion

$$\frac{\partial^+ z_j}{\partial z_i} = \frac{\partial z_j}{\partial z_i} + \sum_{k=i}^j \frac{\partial z_j}{\partial z_k} \frac{\partial^+ z_k}{\partial z_i} \quad (3)$$

uses arguments similar to those used to prove the first expansion.

Finally, one comment on mathematical notation, throughout this paper it is assumed that a partial derivative of the form,  $\partial a / \partial b$ , where  $a \in R^{[A \times 1]}$  and  $b \in R^{[B \times 1]}$ , is a matrix of the form  $R^{[A \times B]}$ .

### 3 Epochwise and On-Line Training

Any adaptive algorithm adjusts the parameters of a system so that the system responds to a set of inputs in some desired manner. First-order gradient descent algorithms accomplish this goal by minimizing an error function. The definition of this error function is dependent upon whether the system is operating in an epochwise or on-line mode. In this section, both epochwise and on-line training are defined. The epochwise and on-line error functions as well as their associated weight update equations are also presented.

#### 3.1 Epochwise Training Algorithms

An epoch is a forward iteration of the dynamic system from iteration  $k = 0$  to  $k_f$ , where  $k_f$  is the final iteration. An epochwise training algorithm is any algorithm in which training takes place after each epoch or a series of epochs of the dynamic system.

In order to use the gradient descent epochwise algorithms, an error must be defined. It is common for this error to be a function only of the outputs of the dynamic system,  $\{y_0, \dots, y_{k_f}\}$ , and a set of desired

output vectors,  $\{d_0, \dots, d_{k_f}\}$ , with  $d_k \in R^{[N \times 1]}$ . The desired output vectors and the set of inputs vectors associated with the desired output vectors are given in a training set. This set is composed of  $P$  elements with element  $p$  taking the form  $\{R_{0p}, Y_{0p}, r_{1p}, \dots, r_{k_f p}, d_{1p}, \dots, d_{k_f p}\}$ . It should be noted that the desired response need not be defined for each iteration, only for the final iteration  $k_f$ . A commonly used epochwise error function is

$$E = \sum_{p=0}^P \sum_{k=0}^{k_f} \frac{1}{2} (d_{kp} - y_{kp})^T (d_{kp} - y_{kp}) \quad (4)$$

which is the sum of the squared error over the entire training set. The error of Equation 4 is calculated using an ordered set of equations. Because of the ordering of this set, the error is always the last calculation performed in this set.

Utilizing gradient descent, the epochwise algorithms presented in this paper update the weights using

$$w(i+1) = w(i) - \mu \frac{\partial^+ E}{\partial w(i)} \quad (5)$$

where  $\mu$ , the learning rate, is a suitably chosen positive constant. The update rule generates a new weight vector  $W(i+1)$  from the vector  $W(i)$ . If the error function defined in Equation 4 is used, the weights are updated after cycling through the training set. Therefore, if weight vector  $W(1)$  is used for the first cycle through the training set, weight vector  $W(i)$  is used for the  $i^{\text{th}}$  cycle through the set. Generally, the weight vector of the first cycle,  $W(1)$ , is randomly initialized.

Although first-order gradient descent provides the basis for adaptation of the weights, the algorithms discussed in this paper have come to be known by the method for which they calculate the error gradient of Equation 5. Hence, it should be remembered, that even though backpropagation-through-time and recursive backpropagation, two algorithms presented in Section 5, have quite different names, they both perform first-order gradient descent.

### 3.2 On-line Training Algorithms

If the weight update of an algorithm at the current iteration  $k'$  depends only on the states of the system at iterations  $\{k', k' - 1, k' - 2, \dots\}$ , then the algorithm is defined to be an on-line training algorithm. The implied dependence only upon the current and past values of the system allows the weight updates to be computed in real-time in most cases. The key difference between on-line and epochwise training algorithms is that an on-line algorithm adapts the weights of the system as it runs while an epochwise training algorithm only updates the weights after the final iteration. The primary reason for using an on-line algorithm is that as the number of iterations in an epoch becomes very large, it becomes computationally inefficient to update the weights only after each epoch. Therefore, on-line algorithms, which adapt the weights as the system runs, must be used.

In the on-line case, an error is defined for each iteration. At the current forward iteration,  $k'$ , the error,  $E_{k'}$ , is often a function of the desired response vector,  $d_{k'}$ , and the output vector,  $y_{k'}$ . It is common to use the on-line error function

$$E_{k'} = \frac{1}{2} (d_{k'} - y_{k'})^T (d_{k'} - y_{k'}) \quad (6)$$

It is well known that in the on-line case, minimization of Equation 6 using first-order gradient descent at each iteration results in the minimization of the mean square error [9]; therefore, using the error defined by Equation 6 minimizes

$$\mathbf{E}[E_{k'}] = \mathbf{E}\left[\frac{1}{2} (d_{k'} - y_{k'})^T (d_{k'} - y_{k'})\right]$$



Figure 3: Static System.

where  $\mathbf{E}$  is the expected value operator.

Often, on-line training algorithms update the weights at each iteration based upon the gradient of the error function. At iteration  $k'$ , the on-line update rule is expressed as

$$w(k' + 1) = w(k') - \mu \frac{\partial^+ E_{k'}}{\partial w(k')} \quad (7)$$

where  $\mu$  is a suitably chosen constant and  $E_{k'}$  is the appropriate on-line error function. Equation 7 is usually initialized by a random setting of  $w(0)$ . The application of Equation 7 generates a new vector of weights at each iteration. When using an on-line algorithm, it is common for the weights at the iteration  $k'$  to be selected from the vector  $\mathbf{W}(k')$ , therefore,  $w_{k'}(k') = w(k')$ .

## 4 Static System Algorithms

In order to facilitate the discussion of the training algorithms for discrete-time dynamic systems, it is useful to introduce the first-order gradient descent algorithms for static systems. A static system contains no feedback, therefore, a static system has the structure shown in Figure 3, where  $\mathbf{r} \in R^{[M \times 1]}$  is the input vector and  $\mathbf{y} \in R^{[N \times 1]}$  is the output vector. A static system can be described by the following equation

$$\mathbf{y} = f(\mathbf{r}, \mathbf{W}(i)).$$

### 4.1 The Backpropagation Algorithm

As in the dynamic system case, the first-order gradient descent techniques for static systems depends upon minimizing an error. In general, this error is a function of the output, and the output is a function of the weights. Therefore, using the chain rule of Equation 2, the error gradient may be expressed as

$$\frac{\partial^+ E}{\partial w(i)} = \frac{\partial^+ E}{\partial \mathbf{y}} \frac{\partial \mathbf{y}}{\partial w(i)} = \frac{\partial E}{\partial \mathbf{y}} \frac{\partial \mathbf{y}}{\partial w(i)}$$

Assuming that the appropriate equations for the error and the output are available and differentiable, an expression for the error gradient with respect to each weight can be found by differentiating the error and output equations.

If more is known about the structure of the system, it is possible to use this information to decrease the number of computations needed to find the error gradients. The backpropagation algorithm of Rumelhart *et al* [4] does precisely this. Based upon the fact that the static system is composed of a layered feedforward neural network, the backpropagation algorithm efficiently computes the error gradients for such a static network. Using the backpropagation algorithm, the error gradient

$$\frac{\partial^+ E}{\partial w(i)} = \lambda \frac{\partial \mathbf{y}}{\partial w(i)} \quad (8)$$

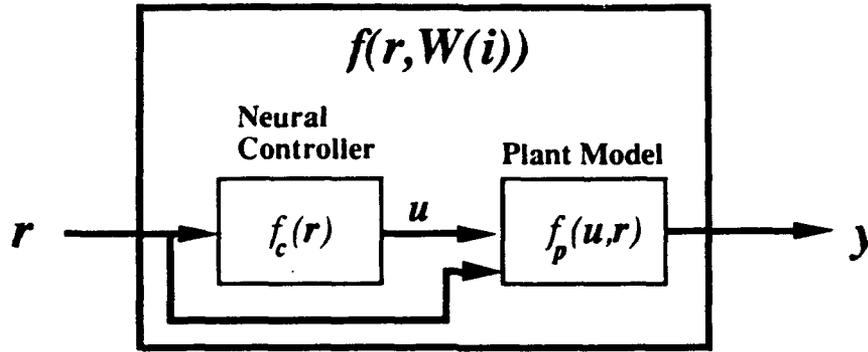


Figure 4: Static Neural Network Control System.

where

$$\lambda = \frac{\partial E}{\partial y}.$$

is calculated by backpropagating the vector,  $\lambda$ , through the neural network. It should be noted that any equation which takes the form of Equation 8, can be calculated using backpropagation, provided the structure of the system is a neural network.

## 4.2 Static Controller-Plant System

As we shall next show, the error gradient of a static controller-plant system takes the form of Equation 8. In static neural control applications, the system is composed of two components, the controller and plant, as shown in Figure 4. The controller is implemented by a multilayered neural network while the plant may be modeled by a neural network or a set of equations. When the plant is modeled using a neural network, the combination of the controller and plant form a static neural network. Therefore, the error gradient with respect to each weight of the neural controller takes the form of Equation 8 and can be calculated using the backpropagation algorithm.

If the plant is modeled by a set of equations, a differentiation of the plant equations with respect to the control vector in combination with a backpropagation can be used to compute the error gradient. The error of the system of Figure 4 is a function of the plant output, which is a function of the controller output. The controller output is a function of the weights of the network. Therefore, using the chain rule, the error gradient can be written as

$$\frac{\partial E}{\partial w(i)} = \lambda' \frac{\partial u}{\partial w(i)} \quad (9)$$

where

$$\lambda' = \frac{\partial E}{\partial y} \frac{\partial y}{\partial u}.$$

Equations 8 and 9 have the same form. Because  $u$  is the output of a neural network, the error gradients with respect to the controller's weights are found by backpropagating  $\lambda'$  through the neural network controller. The backpropagation term,  $\lambda'$ , is computed by multiplying  $\partial E/\partial y$  by the plant Jacobian matrix,  $\partial y/\partial u$ , which is calculated from the plant equations.

### 4.3 Training Neural Networks Implemented on Chip

It is also possible to train a static feedforward network which is implemented on a VLSI chip using first-order gradient descent. In this case, it is assumed that the precise mathematical equations of the neural network system are not known. Therefore, the derivatives cannot be calculated mathematically. Instead, they can be computed by introducing small perturbations into the hidden layer nodes. The resulting perturbation at the output divided by the node perturbation approximates the output gradient with respect to the node. This method, in conjunction with LMS [9], can be used to calculate the output gradient with respect to the weights. This technique of calculating the gradient is known as Madaline Rule III (MR III) [10]. A VLSI chip manufactured by Intel supports this type of training [11].

The first-order gradient techniques discussed above are used not only for adapting static systems, they are also a key component of the rules used for adapting discrete-time dynamic systems. It is shown in the next section that these techniques are required for training dynamic systems composed fully or partially of neural networks.

## 5 Algorithms

In this section two basic types of algorithms, Euler-Lagrange based algorithms and recursive gradient update algorithms, both of which are used for training discrete-time dynamic systems, are discussed. The epochwise and on-line versions of both these types of algorithms are presented. As indicated by their name, the Euler-Lagrange algorithms are based on the discrete-time Euler-Lagrange equations [12]. These equations are used to calculate the error gradient with respect to the weights. The backpropagation-through-time algorithm, which is used for epochwise training of dynamic neural networks, is an example of an Euler-Lagrange based algorithm. The recursive gradient update algorithm is based upon a recursive equation for the output gradient which is derived from the dynamic system definition, Equation 1. The error gradient is easily computed using this output gradient. The recursive backpropagation algorithm, which is used for on-line training of dynamic neural networks, is an example of a recursive gradient update algorithm. The epochwise and on-line versions of algorithms based upon the Euler-Lagrange equations and the recursive gradient update equation can be used to train a variety of dynamic systems which contain neural networks as will be shown periodically in the remainder of this section.

Before deriving the Euler-Lagrange and recursive gradient update algorithm, it is worth mentioning that the stability of these algorithms cannot be guaranteed. Therefore, when using these algorithms, one must constantly monitor their performance. If instability becomes a problem when using one of these algorithms, it is often necessary to change certain parameters of the algorithm, often the learning rate, to overcome the problem.

For any given discrete-time dynamic system problem, either an Euler-Lagrange based algorithm or a recursive gradient update algorithm can be used to train the system. In Section 6, it is shown that the Euler-Lagrange based algorithm and recursive gradient update algorithm compute approximately the same error gradient for a given problem in both the epochwise and on-line case. Even though the algorithms are inherently equivalent, the computational and storage requirements of the algorithms are different. Therefore, the selection of the appropriate algorithm for a specific problem should be based upon the computational and storage requirements. These requirements are derived in Section 7.

### 5.1 An Algorithm Based on the Euler-Lagrange Equations

The discrete-time Euler-Lagrange equations in the calculus of variations provide a standard technique for calculating the first-order gradients of an error function. Using these equations, it is possible to calculate the epochwise gradient of any discrete-time dynamic system provided that the differentiability requirements on the system, discussed in Section 2, are met. When the dynamic system is composed fully of a feedforward neural network, the error gradient can be calculated using a combination of the Euler-Lagrange equations and the backpropagation algorithm. This combination is the basis of the backpropagation-through-time algorithm which was first introduced by Werbos [6]. A number of researchers including Nguyen and Widrow [3],

Pearlmutter [2] and Jordan [8] have successfully used the backpropagation-through-time algorithm to train dynamic networks.

In this section, the discrete-time Euler-Lagrange equations are first derived. Next, an epochwise training algorithm which uses these equations is discussed. Finally, training of dynamic systems composed fully or partially of neural networks using the Euler-Lagrange based algorithm is presented.

### 5.1.1 Discrete-Time Euler-Lagrange Equations

In order to use first-order gradient descent, we need to find the error gradient,  $\partial^+ E/\partial w(i)$ , for any given epoch. This gradient can be derived using the first chain rule expansion, Equation 2, and the following ordered set of equations, which are generated at each epoch.

$$\begin{aligned}
 \mathbf{W}_0(i) &= \mathbf{W}(i) \\
 \mathbf{y}_0 &= f(\mathbf{R}_0, \mathbf{Y}_0, \mathbf{W}_0(i)) \\
 \mathbf{W}_1(i) &= \mathbf{W}(i) \\
 \mathbf{y}_1 &= f(\mathbf{R}_1, \mathbf{Y}_1, \mathbf{W}_1(i)) \\
 &\vdots \\
 \mathbf{W}_k(i) &= \mathbf{W}(i) \\
 \mathbf{y}_k &= f(\mathbf{R}_k, \mathbf{Y}_k, \mathbf{W}_k(i)) \\
 &\vdots \\
 \mathbf{W}_{k_f}(i) &= \mathbf{W}(i) \\
 \mathbf{y}_{k_f} &= f(\mathbf{R}_{k_f}, \mathbf{Y}_{k_f}, \mathbf{W}_{k_f}(i)) \\
 E &= f(\mathbf{y}_0, \mathbf{y}_1, \dots, \mathbf{y}_{k_f}, \mathbf{d}_0, \mathbf{d}_1, \dots, \mathbf{d}_{k_f})
 \end{aligned}$$

Using the first chain rule expansion for an ordered system, Equation 2, we can expand the ordered derivative,  $\partial^+ E/\partial w(i)$ , to obtain

$$\frac{\partial^+ E}{\partial w(i)} = \frac{\partial E}{\partial w(i)} + \sum_{k=0}^{k_f} \left( \frac{\partial^+ E}{\partial \mathbf{y}_k} \frac{\partial \mathbf{y}_k}{\partial w(i)} + \frac{\partial^+ E}{\partial \mathbf{W}_k(i)} \frac{\partial \mathbf{W}_k(i)}{\partial w(i)} \right). \quad (10)$$

The terms  $\partial E/\partial w(i)$  and  $\partial \mathbf{y}_k/\partial w(i)$  are equal to zero because  $E$  and  $\mathbf{y}_k$  are not a direct function of  $w(i)$ <sup>1</sup>. Thus, we find the expansion of Equation 10 can be written as

$$\frac{\partial^+ E}{\partial w(i)} = \sum_{k=0}^{k_f} \frac{\partial^+ E}{\partial \mathbf{W}_k(i)} \frac{\partial \mathbf{W}_k(i)}{\partial w(i)} = \sum_{k=0}^{k_f} \frac{\partial^+ E}{\partial w_k(i)}. \quad (11)$$

We need to find an expression for the term  $\partial^+ E/\partial w_k(i)$ . This expression can be found by expanding the ordered derivative using Equation 2.

$$\frac{\partial^+ E}{\partial w_k(i)} = \frac{\partial E}{\partial w_k(i)} + \sum_{j=k}^{k_f} \frac{\partial^+ E}{\partial \mathbf{y}_j} \frac{\partial \mathbf{y}_j}{\partial w_k(i)} + \sum_{j=k+1}^{k_f} \frac{\partial^+ E}{\partial \mathbf{W}_j(i)} \frac{\partial \mathbf{W}_j(i)}{\partial w_k(i)}.$$

<sup>1</sup> In our definition of the partial derivative, all terms are held constant except the terms in the denominator of the partial derivative. Therefore, if the function which defines the numerator of the partial derivative does not contain the terms of the denominator directly, then the partial derivative is zero.

The terms  $\partial E/\partial w_k(i)$  and  $\partial \mathbf{W}_j(i)/\partial w_k(i)$  are equal to zero. The term  $\partial y_j/\partial w_k(i)$  is nonzero only when  $k = j$ . Using these results, we find the ordered derivative,  $\partial^+ E/\partial w_k(i)$ , to be

$$\frac{\partial^+ E}{\partial w_k(i)} = \frac{\partial^+ E}{\partial y_k} \frac{\partial y_k}{\partial w_k(i)} \quad (12)$$

Substituting Equation 12 into Equation 11, the error gradient is

$$\frac{\partial^+ E}{\partial w(i)} = \sum_{k=0}^{k_f} \frac{\partial^+ E}{\partial y_k} \frac{\partial y_k}{\partial w_k(i)} = \sum_{k=0}^{k_f} \lambda_k \frac{\partial y_k}{\partial w_k(i)} \quad (13)$$

where

$$\lambda_k = \frac{\partial^+ E}{\partial y_k}$$

The term  $\partial y_k/\partial w_k(i)$  of Equation 13 is easy to calculate. The term  $\partial^+ E/\partial y_k$  must still be expanded. Once again, using the chain rule expansion, Equation 2, we expand,  $\partial^+ E/\partial y_k$ , to find

$$\lambda_k = \frac{\partial^+ E}{\partial y_k} = \frac{\partial E}{\partial y_k} + \sum_{j=k+1}^{k_f} \left( \frac{\partial^+ E}{\partial y_j} \frac{\partial y_j}{\partial y_k} + \frac{\partial^+ E}{\partial \mathbf{W}_j(i)} \frac{\partial \mathbf{W}_j(i)}{\partial y_k} \right) \quad (14)$$

The term  $\partial \mathbf{W}_j(i)/\partial y_k$  is equal to zero. The term  $\partial y_j/\partial y_k$  is also equal to zero when  $j > k + L$ , where  $L$  is the maximum number of delays in the feedback of dynamic system. Using these results, Equation 14 can be written as

$$\lambda_k = \frac{\partial E}{\partial y_k} + \sum_{j=1}^L \frac{\partial^+ E}{\partial y_{k+j}} \frac{\partial y_{k+j}}{\partial y_k} \quad (15)$$

$$= \epsilon_k + \sum_{j=1}^L \lambda_{k+j} \frac{\partial y_{k+j}}{\partial y_k} \quad (16)$$

where

$$\epsilon_k = \frac{\partial E}{\partial y_k}$$

Equation 16 is a backward difference equation which can be solved using the following boundary conditions

$$\lambda_{k_f} = \epsilon_{k_f} \quad (17)$$

$$\lambda_{k_f+1}, \dots, \lambda_{k_f+L} = 0 \quad (18)$$

We shall refer to equations 16, 17 and 18 as the Euler-Lagrange gradient equations. These gradient equations along with

$$\frac{\partial^+ E}{\partial w(i)} = 0$$

form the discrete-time Euler-Lagrange equations. Equation 19 guarantees that a solution of these equations results in either a minimum or maximum. It usually is not possible to find the analytic solution of these equations. Instead, numerical methods, such as first-order gradient descent, are used to search for an approximate solution.

### 5.1.2 Implementation of the Algorithm

Having derived the Euler-Lagrange equations, we introduce the Euler-Lagrange based algorithm which is used to calculate the error gradient with respect to the weights for an epoch. First, the discrete-time dynamic equation of the system, Equation 1, is iterated forward in time from iteration 0 to  $k_f$ . An appropriately selected training set element is used to supply the boundary conditions,  $\mathbf{R}_0, \mathbf{r}_0, \dots, \mathbf{r}_{k_f}$  and  $\mathbf{Y}_0$ , for the forward iteration of Equation 1. Next, the error gradients with respect to the outputs,  $\lambda_k$ , are calculated by backward iterating Equation 16 using the boundary conditions of equations 17 and 18. Finally, the results of the backward sweep are used to compute the error gradient, Equation 13.

The technique for calculating the error gradient presented above is independent of the dynamic system. As long as the output gradient with respect to the weights and recurrent inputs exists, the epochwise error gradient of a discrete-time dynamic system with adaptive parameters may always be computed using this method.

### 5.1.3 Training Dynamic Systems Composed of Neural Networks

If the system function,  $f$ , is implemented by a feedforward neural network, an interesting observation can be made. The error gradient summation computation, equation 13, contains terms of the form  $\lambda_k \partial \mathbf{y}_k / \partial w_k(i)$ . It is shown in Section 3.2 that terms of this form can be calculated using a backpropagation provide that  $\mathbf{y}_k$  is the output of a neural network. Therefore, in the dynamic network case, each term of Equation 13 can be computed by backpropagating the vector  $\lambda_k$  through the dynamic network. Furthermore, the summation terms of the backward sweep calculation, Equation 16, are of the form  $\lambda_{k+j} \partial \mathbf{y}_{k+j} / \partial \mathbf{y}_k$ . Once again, in the dynamic network case, these terms may be calculated using the backpropagation algorithm. Although it may seem that a large number of backpropagations are required for each epoch to calculate the error gradient, only  $k_f + 1$  backpropagations are needed. By backpropagation of the vector  $\lambda_k$  at each iteration of the backward sweep and by storing the results of this backpropagation in memory, the minimum number of backpropagations can be achieved. This technique of calculating the error gradient is known as the backpropagation-through-time algorithm.

Neural network controllers can be designed using the Euler-Lagrange based algorithm. Systems with the neural network plant model may be trained using the backpropagation-through-time algorithm. If the plant model consists of a set of equations, the summation terms of both the backward sweep calculation, Equation 16, and error gradient computation, Equation 13, may be calculated by backpropagating the vector  $\lambda'_k = \lambda_k \partial \mathbf{y}_k / \partial \mathbf{u}_k$  at each iteration of the backward sweep.

Finally, a combination of the Euler-Lagrange based algorithm and the MRIII algorithm can be used to train a discrete-time neural network which is implemented on a VLSI chip. The summation terms of both the backward sweep calculation, Equation 16, and the error gradient summation computation, Equation 13, can be computed using the MRIII algorithm at each backward iteration of the Euler-Lagrange based algorithm.

## 5.2 An On-Line Algorithm Based on the Euler-Lagrange Equations

The Euler-Lagrange based algorithm is an epochwise training technique. In many applications, such as real-time filtering and adaptive control, it is necessary to allow on-line training. In these cases, an on-line version of the Euler-Lagrange based algorithm, which is introduced in this section, can be used.

Generally, in the on-line case, at each forward iteration of the dynamic system,  $k'$ , the error gradient is first calculated and the weights are updated based upon this calculation. Using the results of the previous section, the error gradient could be calculated by iterating Equation 16, which is repeated here,

$$\lambda_k = \epsilon_k + \sum_{j=1}^L \lambda_{k+j} \frac{\partial y_{k+j}}{\partial y_k} \quad (19)$$

backwards through time from iteration  $k'$  to 0. Because it is common to use the mean squared error in the on-line case, the boundary conditions of Equation 19 would take the form

$$\lambda_{k'} = -(\mathbf{d}_{k'} - \mathbf{y}_{k'}) \quad (20)$$

$$\lambda_{k'+1}, \dots, \lambda_{k'+L} = 0 \quad (21)$$

$$\epsilon_0, \dots, \epsilon_k, \dots, \epsilon_{k'-1} = 0. \quad (22)$$

Finally, the results of the backward sweep are summed to produce the on-line error gradient using

$$\frac{\partial^+ E_{k'}}{\partial w(k')} = \sum_{k=0}^{k'} \frac{\partial^+ E_{k'}}{\partial w_k(k')} = \sum_{k=0}^{k'} \lambda_k \frac{\partial y_k}{\partial w_k(k')}$$

which is similar to Equation 13 of the previous section.

### 5.2.1 Problems Associated with On-Line Implementation

Two problems arise when one attempts to determine the on-line error gradient of a dynamic system in this manner. First, the number of iterations,  $k'$ , for most on-line applications quickly becomes large. Because the error gradient is calculated by iterating a difference equation backwards from  $k'$  to 0, the number of computations required to calculate the gradient grows linearly with  $k'$ . Obviously, this technique of calculating the gradient quickly becomes computationally expensive. Instead of using an algorithm whose computations grows linearly with the current iteration count, it is better to use an algorithm whose computations remain constant and are independent of the current iteration count. This can be accomplished by iterating Equation 19 backwards through time a constant,  $T$ , number of iterations. Using this idea, the error gradient is calculated by first iterating Equation 16 backwards in time from iteration  $k'$  to  $k' - T$  using the appropriate boundary conditions. After this computation, the error gradient is calculated using

$$\frac{\partial^+ E_{k'}}{\partial w(k')} \approx \sum_{k=k'-T}^{k'} \frac{\partial^+ E_{k'}}{\partial w_k(k')} = \sum_{k=k'-T}^{k'} \lambda_k \frac{\partial y_k}{\partial w_k(k')} \quad (23)$$

Of course, the error gradient computed using this method is an approximation of the true gradient. An example will illustrate the nature of this approximation. Figure 5, shows the values of  $\partial^+ E_{k'}/\partial w_k(k')$  for some given dynamic system. By summing only a portion of these terms, it can be observed in Figure 6 that the resulting approximate error gradient is a *windowed* version of the true gradient. Thus, the validity of the approximate gradient depends upon how much of the gradient lies outside of the window of length  $T$ .

The second problem with directly using the Euler-Lagrange equations in the on-line case results from the weight changes at each iteration. In the on-line case, the system difference equation is

$$\mathbf{y}_k = f(\mathbf{R}_k, \mathbf{Y}_k, \mathbf{W}_k(k)) \quad (24)$$

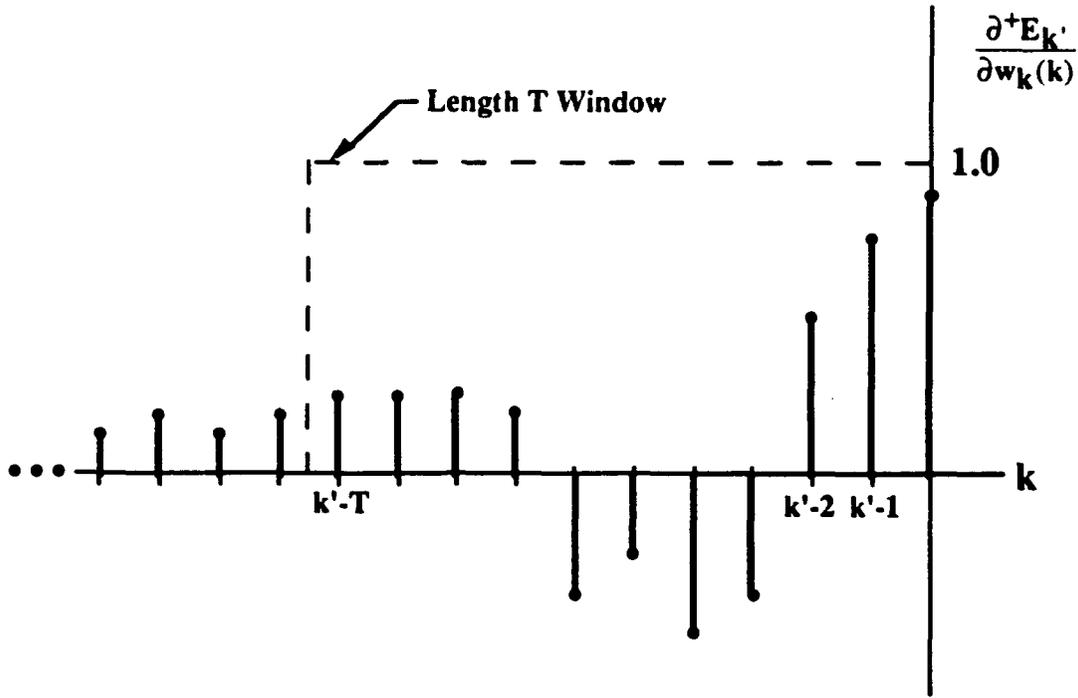


Figure 5: Error Gradients of a Typical System.

where the weight vector,  $\mathbf{W}_k$ , changes at each iteration according to Equation 7. The on-line error gradient,  $\partial E_{k'}/\partial w(k')$ , is defined with respect solely to the  $k'^{th}$  weight vector. However, this weight vector is only used at iteration  $k'$ . Because the weight  $w(k')$  only appears at iteration  $k'$ , one could conclude incorrectly that the error gradient could be calculated based solely upon iteration  $k'$ . This is incorrect because the weight vectors are related by Equation 7. In fact, in most cases of interest, the weights change slowly from iteration to iteration because the learning rate,  $\mu$ , of Equation 7 is small. Under this condition, we find

$$\mathbf{W}(k) \approx \mathbf{W}(k-1).$$

Using this approximation, the on-line error gradient summation of Equation 23, can be written as

$$\frac{\partial E_{k'}}{\partial w(k')} \approx \sum_{k=k'-T}^{k'} \frac{\partial^+ E_{k'}}{\partial w_k(k)} = \sum_{k=k'-T}^{k'} \lambda_k \frac{\partial y_k}{\partial w_k(k)} \quad (25)$$

where  $w_k(k)$  has replaced  $w_k(k')$  in the partial derivative  $\partial y_k/\partial w_k(k)$ . The combination of Equation 25 and a backward sweep of length  $T+1$  allows the calculation of an approximate error gradient.

The error gradient approximation can be improved by exponentially weighting the terms of Equation 25. Because in most cases the weight change between  $w(k)$  and  $w(k')$  tends to become larger as  $k$  is decreased starting from  $k'$ , it can be argued that the approximation  $\partial E_{k'}/\partial w_k(k) \approx \partial E_{k'}/\partial w_k(k')$  becomes less valid as  $k$  decreases. Under this assumption, when calculating the error gradient, the influence of the less accurate terms of Equation 25 should proportionally be reduced. This can be accomplished using the following exponential weighting scheme

$$\frac{\partial E_{k'}}{\partial w(k')} \approx \sum_{k=k'-T}^{k'} \alpha^{k'-k} \frac{\partial^+ E_{k'}}{\partial w_k(k)} = \sum_{k=k'-T}^{k'} \alpha^{k'-k} \lambda_k \frac{\partial y_k}{\partial w_k(k)}. \quad (26)$$

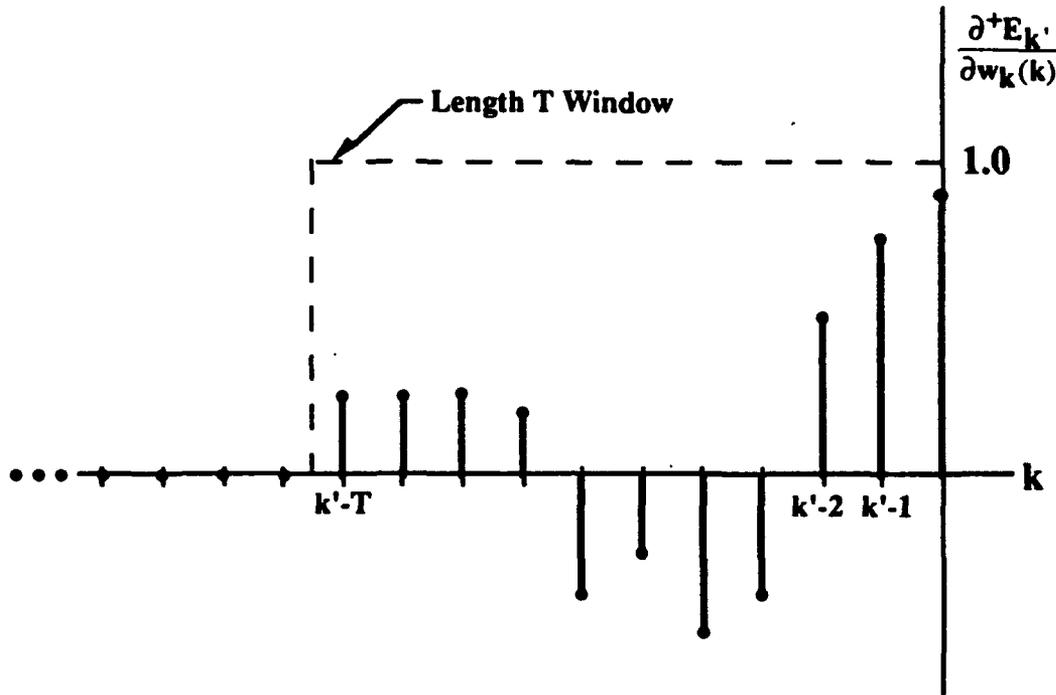


Figure 6: Windowed Error Gradients.

where the constant  $0.0 < \alpha < 1.0$  is the weighting coefficient. The addition of exponential weighting causes the window to have the form shown in Figure 7. It should be noted, that exponential weighting may be desirable even in the epochwise training case if the backward sweep computation of Equation 16 is unstable.

### 5.2.2 Implementation of the Algorithm

Having derived approximate solutions for the two problems of implementing the Euler-Lagrange equations on-line, we can now present the on-line Euler-Lagrange based algorithm. The algorithm is based upon the following sequences being performed at each iteration  $k'$ : a forward propagation based on Equation 24, a backward sweep of length  $T + 1$  using Equation 19 and the boundary conditions of equations 20, 21, and 22, a calculation of the error gradient, Equation 26, and an update of the weights based upon Equation 7.

For a dynamic system with  $Q$  weights, the on-line algorithm outlined above requires that  $(T + 1)Q$  weight vectors be stored. In most cases, this is an impractical amount of storage. Under these circumstances, an approximate output vector of the form

$$\mathbf{y}_k \approx f(\mathbf{R}_k, \mathbf{Y}_k, \mathbf{W}_k(k')) \quad (27)$$

may be used in Equation 19, the backward sweep equation, instead of the output vector defined in Equation 24. Equation 27 is a function of only one weight vector,  $\mathbf{W}(k')$ , therefore, only this vector needs to be stored. Using the approximation of Equation 27 results in a good approximation of  $\partial E_{k'} / \partial w_k(k')$  provided that the weights change slowly.

In the on-line case, the summation terms of both the backward sweep calculation, Equation 19, and the error gradient computation, Equation 26, may be calculated in exactly the same manner as in the epochwise case, which is discussed in the last part of Section 5.1. Therefore, dynamic systems composed of a neural network or neural controller-plant system may be trained on-line. The on-line Euler-Lagrange algorithm can be used in conjunction with the backpropagation algorithm to calculate the error gradient of a dynamic neural network. This technique of calculating the on-line error is known as on-line backpropagation-through-time.

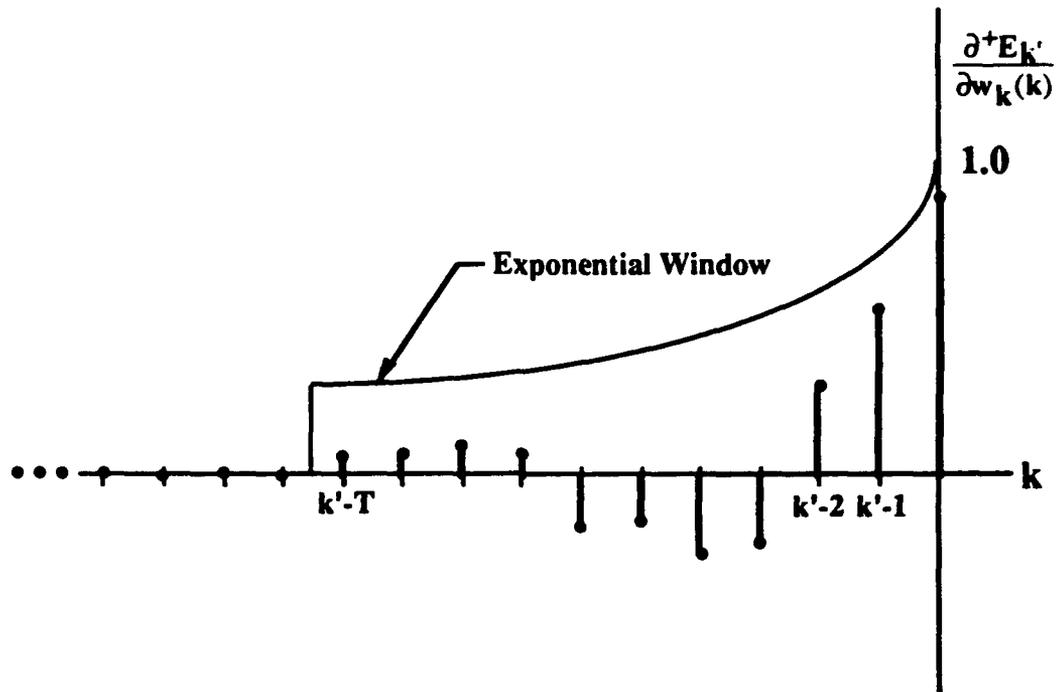


Figure 7: Exponentially Windowed Error Gradients.

One difficulty associated with the on-line Euler-Lagrange based algorithm is the selection of appropriate values for the constants  $T$  and  $\alpha$ . Like the selection of the learning rate,  $\mu$ , there are no analytic procedures for choosing these constants. Instead, the selection should be based upon knowledge of the dynamic system, desired convergence rate and required misadjustment upon convergence.

### 5.3 Recursive Gradient Update Algorithm

The recursive gradient update algorithm provides yet another method for adapting a discrete-time dynamic system composed fully or partially of a neural network. The earliest version of the algorithm, which adapted a single linear node, was introduced by White in 1975 [13]. The algorithm received much attention during the later 70's in the adaptive signal processing community. It was found to suffer from stability problems and much of the recent research has been dedicated to overcoming this problem [14]. The recursive gradient update algorithm for nonlinear networks became well known because of a paper by Williams and Zipser [1]. Although this paper dealt only with single layer nonlinear networks, their version of the recursive algorithm could be generalized to multilayered networks by appropriately selecting the connections between input and output. In this section, the recursive algorithm for a general system, which may be composed partially or fully of a neural network, is presented.

In both the epochwise and on-line cases, the recursive algorithm utilizes first-order gradient descent to minimize an appropriate error function. The difference between the Euler-Lagrange based algorithm and the recursive algorithm is the method in which the error gradient with respect to the weights is calculated. To gain an understand of the difference between the algorithms, the epochwise version of recursive algorithm is developed.

#### 5.3.1 Epochwise Error Gradient

The Euler-Lagrange based algorithm is derived using the first chain rule expansion, Equation 2. In this section, the second chain rule expansion, Equation 3, is used to derive the recursive gradient update algorithm. Once again, we assume the error,  $E$ , is calculated using the ordered set of equations shown in Section 5.1.1.

We begin by expanding the error gradient using the second chain rule expansion, Equation 3, to obtain the following result.

$$\frac{\partial^+ E}{\partial w(i)} = \frac{\partial E}{\partial w(i)} + \sum_{k=0}^{k_f} \left( \frac{\partial E}{\partial \mathbf{y}_k} \frac{\partial^+ \mathbf{y}_k}{\partial w(i)} + \frac{\partial E}{\partial \mathbf{W}_k(i)} \frac{\partial^+ \mathbf{W}_k(i)}{\partial w(i)} \right) \quad (28)$$

The two terms,  $\partial E/\partial w(i)$  and  $\partial E/\partial \mathbf{W}_k(i)$ , are equal to zero because the error,  $E$ , is not a direct function of  $w(i)$  and  $\mathbf{W}_k(i)$ . Therefore, Equation 28 can be written as

$$\frac{\partial^+ E}{\partial w(i)} = \sum_{k=0}^{k_f} \frac{\partial E}{\partial \mathbf{y}_k} \frac{\partial^+ \mathbf{y}_k}{\partial w(i)}. \quad (29)$$

The first term of Equation 29,  $\partial E/\partial \mathbf{y}_k$ , is easy to compute. The second term,  $\partial^+ \mathbf{y}_k/\partial w(i)$ , can be found by using the second chain rule expansion once more.

$$\frac{\partial^+ \mathbf{y}_k}{\partial w(i)} = \frac{\partial \mathbf{y}_k}{\partial w(i)} + \sum_{j=0}^k \frac{\partial \mathbf{y}_k}{\partial \mathbf{W}_j(i)} \frac{\partial^+ \mathbf{W}_j(i)}{\partial w(i)} + \sum_{j=0}^{k-1} \frac{\partial \mathbf{y}_k}{\partial \mathbf{y}_j} \frac{\partial^+ \mathbf{y}_j}{\partial w(i)} \quad (30)$$

The term  $\partial \mathbf{y}_k/\partial w(i)$  equals zero. The term,  $\partial \mathbf{y}_k/\partial \mathbf{W}_j(i)$ , of the first summation is nonzero only when  $k = j$ , therefore, this summation only contains one nonzero term. Furthermore, it is easy to verify that this term can be simplified to  $\partial \mathbf{y}_k/\partial w_k(i)$ . Finally, the first term of the second summation,  $\partial \mathbf{y}_k/\partial \mathbf{y}_j$ , is nonzero only when  $k - j \leq L$ . Using these results, Equation 30, can be written as

$$\frac{\partial^+ \mathbf{y}_k}{\partial w(i)} = \frac{\partial \mathbf{y}_k}{\partial w_k(i)} + \sum_{j=1}^L \frac{\partial \mathbf{y}_k}{\partial \mathbf{y}_{k-j}} \frac{\partial^+ \mathbf{y}_{k-j}}{\partial w(i)}. \quad (31)$$

The summation in Equation 31 can be eliminated using  $\mathbf{Y}_k = [\mathbf{y}_{k-1}, \dots, \mathbf{y}_{k-L}]^T$

$$\frac{\partial^+ \mathbf{y}_k}{\partial w(i)} = \frac{\partial \mathbf{y}_k}{\partial w_k(i)} + \frac{\partial \mathbf{y}_k}{\partial \mathbf{Y}_k} \frac{\partial^+ \mathbf{Y}_k}{\partial w(i)}. \quad (32)$$

This equation can be used to recursively calculate the output gradients for the entire epoch. It is usually initialized using

$$\frac{\partial^+ \mathbf{Y}_0}{\partial w(i)} = 0. \quad (33)$$

### 5.3.2 Implementation of the Algorithm

The epochwise error gradient can be calculated by first using Equation 32 to determine  $\partial^+ \mathbf{y}_k/\partial w(i)$  for each iteration of the epoch. Once these ordered derivatives are determined, the error gradient can be calculated using Equation 29. In order to calculate the error gradient using this method, the output gradient at each iteration of the epoch must be available. If the outputs are stored in memory, a total of  $(k_f + 1)NQ$  memory slots are required, where  $N$  is the number of outputs and  $Q$  is the number of weights. In many cases, this amount of memory may not be available. The memory requirements may be reduced to  $(L + 1)NQ$  using a recursive calculation. Let  $S_k$ , an intermediate error gradient sum, be defined as

$$S_k = \sum_{j=0}^k \frac{\partial E}{\partial y_j} \frac{\partial^+ y_j}{\partial w(i)}$$

Using the recursive equation

$$S_{k+1} = S_k + \frac{\partial E}{\partial y_{k+1}} \frac{\partial^+ y_{k+1}}{\partial w(i)} \quad (34)$$

which is initialized by

$$S_0 = \frac{\partial E}{\partial y_0} \frac{\partial^+ y_0}{\partial w(i)} \quad (35)$$

it follows

$$\frac{\partial^+ E}{\partial w(i)} = S_{k_f}$$

The epochwise recursive update algorithm is implemented using the equations derived immediately above to calculate the error gradient. For a given epoch, the feedback input gradient,  $\partial^+ Y_0 / \partial w(i)$ , is initialized using equation 33 while the intermediate error gradient sum,  $S_k$ , is initialized using Equation 35. At each iteration, from the initial iteration  $k = 0$  to the final iteration  $k = k_f$ , the following sequence is performed: the system is forward propagated using the function  $f$  defined in Equation 1, the output gradient is computed using the recursive calculation of Equation 32 and the intermediate error gradient is updated using Equation 34. Because  $S_k$  and  $\partial^+ y_k / \partial w(i)$  are calculated at each iteration, only the previous  $L$  output gradients need to be stored in memory, therefore, the storage requirements of the algorithm are approximately  $(L + 1)NQ$ . After the final iteration, the error gradient is available as  $S_{k_f}$ .

In order to update the output gradient at each iteration using Equation 32, it is necessary to compute the direct output gradient,  $\partial y_k / \partial w_k(i)$ , and the Jacobian matrix,  $\partial y_k / \partial Y_k$ . Many different techniques can be used to calculate these terms depending upon the form of the dynamic system. As shown below, if the structure of the discrete-time dynamic system is a neural network, these two components can be found using  $N$  backpropagations of  $N$  appropriately selected vectors  $\{\lambda_1, \dots, \lambda_n, \dots, \lambda_N\}$ . These vectors are all backpropagated through the  $k^{\text{th}}$  iteration of the dynamic network.

The  $N$  vectors take the form

$$\lambda_{nj} = \begin{cases} 1 & \text{if } n = j \\ 0 & \text{otherwise} \end{cases}$$

where  $\lambda_{nj}$  is the  $j^{\text{th}}$  element of the row vector  $\lambda_n \in R^{[1 \times N]}$ . Each vector has only one nonzero component in the  $n = j$  column. Because  $y_k$  is the output vector of a neural network, the backpropagation of the vector  $\lambda_n$  through the network at iteration  $k$  results in the calculation of the output gradient of the  $n^{\text{th}}$  output, as indicated by

$$\lambda_n \frac{\partial y_k}{\partial w_k(i)} = \frac{\partial y_k(n)}{\partial w_k(i)}$$

Furthermore, as shown by,

$$\lambda_n \frac{\partial \mathbf{y}_k}{\partial \mathbf{Y}_k} = \frac{\partial y_k(n)}{\partial \mathbf{Y}_k}$$

backpropagating the  $n^{\text{th}}$  vector,  $\lambda_n$ , back to the input nodes results in the calculation of the  $n^{\text{th}}$  row of the Jacobian matrix. This shows that the direct output gradient,  $\partial \mathbf{y}_k / \partial \mathbf{w}_k(i)$ , and Jacobian matrix,  $\partial \mathbf{y}_k / \partial \mathbf{Y}_k$ , can be computed using  $N$  backpropagations through the  $k^{\text{th}}$  iteration of the dynamic network. Using this technique in the epochwise recursive gradient update algorithm shall be referred to as epochwise recursive backpropagation.

The computation of the direct output gradient and the Jacobian matrix for a neural controller-plant system is similar to the calculation for the dynamic neural network. In fact, if the plant model is a neural network, the technique discussed above can be used directly. If the plant model is based upon a set of equations, once again, the direct output gradient and Jacobian matrix can be computed using  $N$  backpropagations through the neural network. In this case, the backpropagated vectors are of the form

$$\lambda'_n = \lambda_n \frac{\partial \mathbf{y}_k}{\partial \mathbf{u}_k} = \frac{\partial y_k(n)}{\partial \mathbf{u}_k}$$

The backpropagation of  $\lambda'$  through the neural controller of iteration  $k$  results in the computation of the direct output gradient as indicated by

$$\lambda'_n \frac{\partial \mathbf{y}_k}{\partial \mathbf{w}_k(i)} = \frac{\partial y_k(n)}{\partial \mathbf{u}_k} \frac{\partial \mathbf{u}_k}{\partial \mathbf{w}_k(i)} = \frac{\partial y_k(n)}{\partial \mathbf{w}_k(i)}$$

Similarly, it can be shown that the  $n^{\text{th}}$  row of the Jacobian matrix can be computed using the backpropagation of  $\lambda'_n$  to the inputs of the neural controller.

If the system is composed of a neural network implemented in hardware, the two terms  $\partial \mathbf{y}_k / \partial \mathbf{w}_k(i)$  and  $\partial \mathbf{y}_k / \partial \mathbf{Y}_k$  of Equation 32 can be computed using MRIII. The MRIII algorithm uses the output gradient to calculate the error gradient. Therefore, this algorithm can be used without modification to find the two terms of Equation 32.

The epochwise recursive algorithm can be used to calculate the epochwise error gradient of any discrete-time dynamic system. In the next section, we show that this algorithm is easily extended to the on-line case.

#### 5.4 On-Line Recursive Gradient Update Algorithm

The on-line version of the recursive gradient update algorithm is easily derived from the epochwise version. The calculation of the output gradient at each iteration performed by the epochwise version of the algorithm depends only upon the current and past values of the dynamic system. The lack of dependence on future values of the network in calculating the output gradient makes the algorithm attractive for on-line implementations.

In the on-line case, the mean squared error is often minimized by updating the weights at each iteration based upon an error gradient of the form

$$\frac{\partial E_{k'}}{\partial \mathbf{w}(k')} = \frac{\partial E_{k'}}{\partial \mathbf{y}_{k'}} \frac{\partial^+ \mathbf{y}_{k'}}{\partial \mathbf{w}(k')} \quad (36)$$

$$= -(\mathbf{d}_{k'} - \mathbf{y}_{k'})^T \frac{\partial^+ \mathbf{y}_{k'}}{\partial \mathbf{w}(k')} \quad (37)$$

The second term of Equation 37 may be calculated using the recursive gradient update calculation, Equation 32, which was derived in the previous section. Using this approach, the squared error is reduced at each iteration, and the mean squared error is approximately minimized.

The calculation of the output gradient using the recursive update computation,

$$\frac{\partial^+ \mathbf{y}_{k'}}{\partial \mathbf{w}(k')} = \frac{\partial \mathbf{y}_{k'}}{\partial \mathbf{w}_{k'}(k')} + \frac{\partial \mathbf{y}_{k'}}{\partial \mathbf{Y}_{k'}} \frac{\partial^+ \mathbf{Y}_{k'}}{\partial \mathbf{w}(k')} \quad (38)$$

is based upon the assumption that the weights are constant. However, in the on-line case, the weights change from iteration to iteration according to Equation 7. In order to use the recursive equation, it is assumed that the weights change slowly or equivalently

$$\mathbf{W}(k) \approx \mathbf{W}(k-1).$$

This assumption allows

$$\frac{\partial^+ \mathbf{Y}_{k'}}{\partial \mathbf{w}(k')} = \left[ \frac{\partial^+ \mathbf{y}_{k'-1}}{\partial \mathbf{w}(k')}^T, \frac{\partial^+ \mathbf{y}_{k'-2}}{\partial \mathbf{w}(k')}^T, \dots, \frac{\partial^+ \mathbf{y}_{k'-L}}{\partial \mathbf{w}(k')}^T \right]^T$$

of Equation 38 to be approximated as

$$\frac{\partial^+ \mathbf{Y}_{k'}}{\partial \mathbf{w}(k')} \approx \left[ \frac{\partial^+ \mathbf{y}_{k'-1}}{\partial \mathbf{w}(k'-1)}^T, \frac{\partial^+ \mathbf{y}_{k'-2}}{\partial \mathbf{w}(k'-2)}^T, \dots, \frac{\partial^+ \mathbf{y}_{k'-L}}{\partial \mathbf{w}(k'-L)}^T \right]^T.$$

Because of this approximation, the on-line error gradient of Equation 37 can only be approximately calculated.

In a manner similar to the on-line Euler-Lagrange based algorithm, the accuracy of the error gradient approximation can be improved by introducing exponential weighting. The exponentially weighted output gradient can be calculated using

$$\frac{\partial \mathbf{y}_{k'}}{\partial \mathbf{w}(k')} \approx \frac{\partial \mathbf{y}_{k'}}{\partial \mathbf{w}_{k'}(k')} + \frac{\partial \mathbf{y}_{k'}}{\partial \mathbf{Y}_{k'}} \mathbf{F} \frac{\partial^+ \mathbf{Y}_{k'}}{\partial \mathbf{w}(k')} \quad (39)$$

where the diagonal matrix  $\mathbf{F} \in R^{[LN \times LN]}$  is used to perform the exponential weighting. In general,  $\mathbf{F}$  takes the form

$$\mathbf{F} = \begin{bmatrix} \phi & 0 & \dots & \dots & 0 \\ 0 & \phi^2 & 0 & \dots & \dots \\ \dots & 0 & \dots & \dots & \dots \\ \dots & \dots & \dots & \dots & 0 \\ 0 & \dots & \dots & 0 & \phi^L \end{bmatrix}$$

where

$$\phi = \begin{bmatrix} \alpha & 0 & \dots & \dots & 0 \\ 0 & \alpha & 0 & \dots & \dots \\ \dots & 0 & \dots & \dots & \dots \\ \dots & \dots & \dots & \dots & 0 \\ 0 & \dots & \dots & 0 & \alpha \end{bmatrix} \quad (40)$$

with  $\phi \in R^{[N \times N]}$  and  $0.0 < \alpha < 1.0$ .

The on-line recursive algorithm with exponential weighting is easily implemented. The feedback input gradient, which is used in the recursive gradient update equation, is initialize as indicated by Equation 33. The elements of the first weight vector,  $\mathbf{W}(0)$ , are randomly initialized. At each iteration, the following sequence is performed: a forward propagation implemented by Equation 24, a recursive update of the output gradient by Equation 39, a calculation of the error gradient which utilizes Equation 37 and an update of the weights using Equation 7.

The on-line recursive algorithm can be used to adapt the weights of a dynamic neural network or neural controller-plant system. As discussed in the previous section, in these cases, the two terms,  $\partial \mathbf{y}_{k'}/\partial \mathbf{w}_{k'}(k')$  and  $\partial \mathbf{y}_{k'}/\partial \mathbf{Y}_{k'}$ , can be calculated using  $N$  backpropagations. The combination of the on-line recursive algorithm and the backpropagation algorithm to adapt a dynamic network shall be referred to as the on-line recursive backpropagation algorithm. In addition, a neural network which is implemented on a VLSI chip can be adapted on-line using a combination of the on-line recursive algorithm and the MRH algorithm. The on-line recursive algorithm provides an alternative technique to the on-line Euler-Lagrange based algorithm. A comparison of these algorithms is presented in the Section 7. This comparison will allow proper selection of an on-line algorithm for a given problem.

## 6 Comparison of the Algorithms

Although the Euler-Lagrange based algorithm and recursive gradient update algorithm may appear to be quite different, they both perform first-order gradient descent in weight space. In the epochwise case where no approximations are required to calculate the error gradient, the two algorithms are equivalent. In this case, because the selection of algorithm does not affect the convergence rate, the algorithm should be chosen on the basis of computational complexity and storage requirements which are discussed in Section 7.

In the on-line case, the two algorithms result in approximately identical weight updates given the same set of inputs. This can be shown using the formulation of Section 5. The exponentially weighted on-line error gradient of the Euler-Lagrange based algorithm, shown in Equation 26, is repeated here for comparison with the on-line gradient calculated by the recursive algorithm.

$$\frac{\partial^+ E_{k'}}{\partial \mathbf{w}(k')} \approx \sum_{k=k'-T}^{k'} \alpha^{k'-k} \frac{\partial^+ E_{k'}}{\partial \mathbf{w}_k(k)}. \quad (41)$$

An equation similar to Equation 41 can be derived for the on-line recursive algorithm. Using induction, it can be proved that the on-line recursive output gradient calculation of Section 5.4, Equation 39, is approximated as

$$\frac{\partial^+ \mathbf{y}_{k'}}{\partial \mathbf{w}(k')} \approx \frac{\partial \mathbf{y}_{k'}}{\partial \mathbf{w}_{k'}(k')} + \frac{\partial \mathbf{y}_{k'}}{\partial \mathbf{Y}_{k'}} \mathbf{F} \frac{\partial^+ \mathbf{Y}_{k'}}{\partial \mathbf{w}(k')} = \sum_{k=0}^{k'} \alpha^{k'-k} \frac{\partial^+ \mathbf{y}_{k'}}{\partial \mathbf{w}_k(k)}.$$

Substituting this result into the error gradient calculation of the on-line recursive algorithm, Equation 36, the weighted on-line error gradient calculated by the recursive algorithm is

$$\frac{\partial^+ E_{k'}}{\partial \mathbf{w}(k')} \approx \frac{\partial E_{k'}}{\partial \mathbf{y}_{k'}} \sum_{k=0}^{k'} \alpha^{k'-k} \frac{\partial^+ \mathbf{y}_{k'}}{\partial \mathbf{w}_k(k)} = \sum_{k=0}^{k'} \alpha^{k'-k} \frac{\partial^+ E_{k'}}{\partial \mathbf{w}_k(k)}. \quad (42)$$

The two on-line techniques are approximately equivalent when equations 41 and 42 are approximately equal. This occurs when

$$\sum_{k=0}^{k'-T-1} \alpha^{k'-k} \frac{\partial E_{k'}}{\partial w_k(k)} \approx 0 \quad (43)$$

By appropriate selection of  $\alpha$ , one can guarantee that Equation 43 is made arbitrarily close to zero. Thus, we can conclude that the two on-line algorithms are approximately equivalent and that the validity of this approximation depends upon satisfying Equation 43.

## 7 Computational Complexity and Storage Requirements

If the two algorithms are equivalent in both the epochwise and on-line cases, then how does one choose which algorithm to use? Obviously, the choice should be based upon practical issues such as computational efficiency and storage requirements. In this section, the computational and storage requirements, with the system architecture,  $f$ , implemented by a neural network, are analyzed and compared. We choose to present only the dynamic neural network case for two reasons. First, the dynamic neural network system is the most common form of discrete-time dynamic system encountered in the neural networks field. Second, the computation and storage requirements are easy to calculate for any system once one understands these requirements for the dynamic neural network. Because we assume a neural network structure of the system architecture, the computational and storage requirements of the backpropagation-through-time, on-line backpropagation-through-time, epochwise recursive backpropagation and on-line recursive backpropagation algorithms are discussed below.

All four of these algorithms are based on the backpropagation algorithm. Therefore, an understanding of the computation requirements of this algorithm is necessary before deriving the complexity of the dynamic network algorithms. Epochwise training, using the backpropagation algorithm, consists of a forward propagation, a backward propagation and a weight update. Each of these computations requires on the order of  $Q$  multiplications and additions, where  $Q$  is the number of weights in the network. Therefore, the epochwise computational requirement of the backpropagation algorithm is on the order of  $3Q$  multiplications and additions. Throughout the remainder of this section, we shall use the term operation to refer to a multiplication and addition.

### 7.1 Backpropagation-Through-Time Based Algorithms

The backpropagation-through-time algorithm, which is an epochwise technique of adapting a dynamic system, is based upon repeated forward and backward propagations through the dynamic network. For any given epoch,  $k_f + 1$  forward and backward propagations are required. Because each of these propagations requires on the order of  $Q$  multiplications and additions,  $2(k_f + 1)Q$  operations are needed to calculate the error gradients,  $\partial^+ E / \partial w_k(i)$ , of each epoch. Using Equation 13 to compute the epochwise error gradient and Equation 5 to update the weights requires approximately  $(k_f + 1)Q$  operations. (In calculating the epochwise computational requirements, it is assumed that the weights are updated at each epoch.) The total number of multiplications and additions using the backpropagation-through-time algorithm, is

$$C_{BPTT} \approx 3(k_f + 1)Q. \quad (44)$$

The storage requirement of the backpropagation-through-time algorithm is derived from two primary components. First, the weights and their associated error gradients need to be stored in memory for efficient computation. These terms require  $2Q$  floating point memory slots. Secondly, the output vector,  $y_k$  and external input vector,  $r_k$  of the dynamic system at each iteration of the epoch are required for the calculation of the backward sweep. In order to minimize the storage requirements, only the inputs and outputs need be stored. The internal states of the system, such as the hidden node activation levels, can be recalculated from the input and output vectors. It should be noted, that minimization of the storage requirements may

increase the computation requirements by at most  $(k_f + 1)Q$  because of the need to recalculate internal states. The external input is composed of  $M$  floating point numbers while the output contains  $N$  terms. Thus,  $(k_f + 1)(M + N)$  memory slots are required for the external inputs and outputs of the dynamic system. Adding the two components together, the minimal storage requirement of the backpropagation-through-time algorithm is

$$S_{BPPT} \approx 2Q + (k_f + 1)(M + N).$$

The computational complexity of the on-line backpropagation-through-time algorithm is based upon the number of operations per iteration. These requirements are easily derived from those of the backpropagation-through-time algorithm. At each iteration, the error gradient is calculated using  $T + 1$  backpropagations, and the weights are updated based upon this gradient. The backpropagations and weight updates require  $2(T + 1)Q$  operations. In addition to these computations, one forward propagation of the dynamic system, which requires  $Q$  multiplications and additions, is necessary. Using these calculations, the computation requirements per iteration of the on-line algorithm is

$$C_{OBPTT} \approx (2T + 3)Q.$$

In order to achieve the minimal storage requirements of the on-line backpropagation-through-time algorithm, the output vector,  $\mathbf{y}_k$ , must be defined as shown in Equation 27. In this case, only one weight vector need be stored in memory. In addition to the weights, the associated error gradients of each weight must be stored. Finally, only the output vector and external input vector of the previous  $T + 1$  iteration are needed for calculation of the error gradient. Thus, the minimal storage requirement of the on-line backpropagation-through-time algorithm is

$$S_{OBPTT} \approx 2Q + (T + 1)(M + N).$$

## 7.2 Recursive Backpropagation Algorithms

The epochwise recursive backpropagation algorithm is based upon a forward propagation, a recursive update of the output gradient and a recursive update of the error gradient being performed at each iteration. The weights are updated at the final iteration based upon the error gradient. The computational complexity of the forward propagations is  $(k_f + 1)Q$ . The complexity of calculating the  $k_f + 1$  output gradients is determined by the computational requirements of the recursive gradient update calculation, Equation 32. In the dynamic network case, the two terms  $\partial \mathbf{y}_k / \partial \mathbf{w}_k(i)$  and  $\partial \mathbf{y}_k / \partial \mathbf{Y}_k$  are computed using  $N$  backpropagations. In addition to the calculation of these two terms, a matrix-vector multiplication, which requires  $N^2L$  operations, must be performed to compute the second term of Equation 32. Finally, the addition of the two terms of Equation 32 requires  $N$  operations per weight. Adding all these components together, the computation complexity of the recursive output gradient calculation, Equation 32, is  $(N^2L + 2N)Q$ . The complexity of finding all output gradients is  $(k_f + 1)(N^2L + 2N)Q$ . At each iteration, the error gradient is updated using Equation 34 which requires a minimum of  $N$  operations per weight. The computation requirements of updating the  $Q$  error gradients over the  $k_f + 1$  iterations of the epoch is  $(k_f + 1)NQ$ . Finally,  $Q$  multiplications and additions are needed to update the weights. The computational requirements of the epochwise recursive backpropagation algorithm is

$$C_{RB} \approx (k_f + 1)(N^2L + 3N + 1)Q + Q. \quad (45)$$

As pointed out in Section 5.3.2, the storage requirements are determined by the recursive output gradient calculation, Equation 32. Therefore, the storage requirement of the epochwise recursive backpropagation algorithm is

	Epochwise Algorithms		On-Line Algorithms	
Requirements	Backpropagation-through time	Recursive Backpropagation	Backpropagation-through-time	Recursive Backpropagation
Computational	$3(k_f + 1)Q$	$(k_f + 1)(N^2L + 3N + 1)Q + Q$	$(2T + 3)Q$	$(N^2L + N(L + 3) + 2)Q$
Storage	$2Q + (k_f + 1)(M + N)$	$(L + 1)NQ$	$2Q + (T + 1)(M + N)$	$(L + 1)NQ$

Table 1: Computation and Storage Requirements.

$$S_{RB} \approx (L + 1)NQ.$$

The complexity of the on-line recursive backpropagation algorithm follows almost immediately from the computational requirements of the epochwise algorithm. At each iteration, the error gradient is calculated using an exponentially weighted recursive update calculation, Equation 39. The only difference between this equation and the one used in the epochwise case is the exponential weighting. Because the weighting constant,  $\mathbf{F} \in R^{[LN \times LN]}$ , is a diagonal matrix,  $LN$  operations per weight are introduced by this matrix multiplication. Therefore, the recursive update calculation of Equation 39 requires  $(N^2L + N(L + 2))Q$  operations. The error gradient, Equation 37, is calculated using at least  $NQ$  operations. Finally, the forward propagation of the system and the weight update both require  $Q$  operations. The computational complexity per iteration of the on-line recursive backpropagation algorithm is

$$C_{ORB} \approx (N^2L + N(L + 3) + 2)Q.$$

The only differences between the on-line and epochwise recursive algorithms are the exponential weighting of the recursive update equation and the weight update at each iteration of the on-line case. These two differences do not account for any difference in storage requirements. Therefore, the storage requirement of the on-line recursive backpropagation is

$$S_{ORB} \approx (L + 1)NQ.$$

which is the same as that required for the epochwise recursive backpropagation algorithm.

### 7.3 Comparison of Algorithms

The computational and storage requirements are outlined in Table 1. Comparing the computational complexity of the two epochwise algorithms, equations 44 and 45, we find the epochwise backpropagation-through-time algorithm to be computationally more efficient than the epochwise recursive backpropagation algorithm. In general, epochwise algorithms based on the Euler-Lagrange equations are more efficient than those based on the recursive update equation. The computational inefficiency of the recursive technique is a result of the output gradient calculation which requires a matrix-vector multiplication for each weight. Although this calculation introduces inefficiency into the recursive backpropagation algorithm, it has the advantage of fixing the storage requirements to  $(L + 1)NQ$ , which is independent of the number of iterations in an epoch. In some cases, where the total number of iterations,  $k_f + 1$ , is large compared to the number of weights,  $Q$ , the epochwise recursive backpropagation algorithm may be advantageous to use for this reason. However, in many cases the total number of iterations is small compared to the number of weights and the storage requirements favor use of the epochwise backpropagation-through-time algorithm.

The ratio of the computational requirements of the two on-line algorithms

$$\frac{C_{OBPTT}}{C_{ORB}} \approx \frac{(2T + 3)Q}{(N^2L + N(L + 3) + 2)Q} \quad (16)$$

can be used to compare the efficiencies of the two on-line algorithms. The most efficient on-line method can be chosen on the basis of the number of outputs,  $N$ , the maximum delay in the feedback loop,  $L$ , and the error gradient window length of the backpropagation-through-time algorithm,  $T$ . Using the ratio of Equation 46, a couple of general statements can be made about selection of an on-line algorithm based on computational efficiency. First, for systems with a small number of outputs,  $N$ , and a maximum delay,  $L$ , less than the window length,  $T$ , the on-line recursive algorithm is the most efficient technique of updating the weights. A general class of systems which meet these conditions are single output IIR adaptive filters. With this in mind, it is not surprising that in the adaptive filter field, the computationally more efficient on-line recursive technique has been well studied [9,14] while we are unaware of any attempts to use the on-line backpropagation-through-time algorithm. Secondly, for dynamic networks with a large number of outputs,  $N$ , or a large number of delays,  $L$ , the backpropagation-through-time algorithm is most efficient. Fully recurrent networks, which have a large number of outputs because each node is regarded as an output, should be adapted using the on-line backpropagation-through-time algorithm. The algorithm should also be used to train multidimension adaptive filters.

In addition to the computational efficiency, the storage requirements of the two on-line algorithms were derived in the previously in this section. On the basis of the storage requirements alone, the on-line backpropagation-through-time algorithm is preferable to the on-line recursive backpropagation algorithm when  $(T + 1)(N + M) < (L + 1)NQ$ . For almost all dynamic neural network systems, this inequality will hold, and the storage requirement will favor the on-line backpropagation-through-time algorithm.

## 8 Reducing On-Line Computational Complexity

Both the on-line backpropagation-through-time and on-line recursive backpropagation algorithm are computationally expensive. In this section, two techniques for reducing the number of computations are briefly presented.

### 8.1 Feeding Back the Desired Response

We have already stated that in the on-line case, it is common to minimize the square error at each iteration. In this case, a desired response vector,  $\mathbf{d}_k$ , must be available at each iteration. One method of speeding-up on-line learning, is to feed back the desired response instead of the output vector. Using this technique, the system equation is

$$\mathbf{y}_k = f(\mathbf{R}_k, \mathbf{D}_k, \mathbf{W}_k(k)) \quad (47)$$

where  $\mathbf{D}_k = [d_{k-1}^T, d_{k-2}^T, \dots, d_{k-L}^T] \in R^{(NL \times 1)}$ . Because the system defined by Equation 47 is independent of previous states of the system and therefore static, the error gradient can be calculated using a single backpropagation. Obviously, this technique is computationally less expensive than the two on-line algorithms of Section 5.

However, a price is to be paid for using this method. An approximate error gradient is calculated, whose validity depends upon the magnitude of the difference between the output and desire response vectors of the previous iterations. If these vectors are significantly different, a poor approximation of the error gradient is used to update the weights. Thus, even though the calculations per iteration are reduced, the number of iteration required to reach convergence will probably increase. Despite the increase in the number of iterations, feeding back the desired response is a method which can greatly decrease the computationally complexity per iteration.

Feeding back the desired response has been extensively studied in the field of adaptive signal processing [9]. This technique, which has been used to adapt single output linear filters, is known as the output-error formulation. A detailed analysis of the advantages and disadvantages of using this method for an adaptive linear filter can be found in Shynk, 1989 [14].

## 8.2 Redefining the On-Line Error Function

A second technique, which reduces the computational requirements while slightly increasing the time to convergence when using the on-line backpropagation-through-time algorithm, is based upon redefining the on-line error function. In the on-line case, the error function is commonly the squared error. In order to calculate the error gradient using the on-line backpropagation-through-time algorithm,  $T$  backpropagations through the system are required at each iteration. However, by changing the error function, the computational complexity of the algorithm can be significantly reduced. Instead of using the squared error, the error function can be redefined as

$$\frac{\partial^+ E_{k'}}{\partial w(k')} = \begin{cases} \sum_{j=k'-C+1}^{k'} \frac{\partial^+ E_j}{\partial w(j)} & \text{if } (k' \bmod C) = 0 \\ 0 & \text{otherwise} \end{cases}$$

where  $C$  is an integer constant greater than 1, and  $k'$  denotes the forward iteration count. Using this definition, the error is nonzero every  $C$  iterations. Thus, the error gradient need only be calculated every  $C$  iterations instead of every iteration. For example, if  $C = 10$ , the number of computations is approximately reduced by a factor of 10, assuming the window length  $T$  is not drastically increased. In general, the window length should be increased to  $T + C$ , and the learning rate,  $\mu$ , should be multiplied  $C$ . The reduction in computational requirements is accomplished by grouping the square error gradient calculations. Even though the number of computations is reduced, the number of iterations to convergence may increase because it may not be possible to multiply  $\mu$  by a factor of  $C$  for stability reasons. A more detailed explanation of this technique of reducing the on-line computations can be found in Williams and Peng, 1989 [15].

This method of speeding-up on-line learning can only be used for the on-line Euler-Lagrange based algorithm. Because the on-line recursive algorithm calculates the output gradients at each iteration, redefining the error gradient as shown in Equation 48 does not significantly change the computational complexity of the algorithm.

## 9 Examples

In this section, two examples which illustrate the uses of the dynamic system training algorithms are presented. The first example demonstrates the use of the algorithms for nonlinear controller design. A neural network is trained using the Euler-Lagrange based algorithm to provide the steering angle of a boat which is placed in a river with a nonlinear current. By providing the proper steering angle, the neural network guides the boat across the river to a designated dock position. The second example illustrates the use of the on-line recursive algorithm for adaptive filtering. In this example, an adaptive noise cancelling system is trained to eliminate filtered noise from a corrupted signal.

### 9.1 Nonlinear Control Example

In this example, a boat is initially placed in a river, which is 200 feet wide, within a region 100 feet upstream or downstream of a dock. The boat is powered by a constant thrust motor which is also used to point the boat in any desired direction. Starting from the initial position, it is desired to maneuver the boat to a dock, which is located on one shore of the river. Maneuvering the boat to the dock is made difficult by the stream's nonlinear current.

Let  $x_k$  denote the distance from the center of the boat to the shore with the dock at iteration  $k$ . Let  $y_k$  denote the distance of the center of the boat upstream or downstream of the dock. Assuming the current only to be a function of the distance from the shore,  $x_k$ , the equations of motion for the boat are

$$\begin{aligned} x_{k+1} &= x_k + 10\cos(u_k) \\ y_{k+1} &= y_k + 10\sin(u_k) + f_c(x_k). \end{aligned} \quad (48)$$

where  $u_k$ , the orientation of the boat given in radians, is the control signal, and  $f_c(x_k)$ , the influence of the current on the boat, is given in feet per iteration. The current, which is parabolic in nature with the greatest force in the middle of the stream at  $x = 100$ , is given by the following equation

$$f_c(x_k) = 7.5 \left( \frac{x_k}{50} - \left( \frac{x_k}{100} \right)^2 \right).$$

The control signal is supplied by the output of a three layer neural network. The first layer contains the two inputs,  $x_k$  and  $y_k$ , which are the states of the system. The hidden layer contains ten sigmoidal neurons which are fully connected to the inputs and a bias. The output layer, which is linear, is fully connected to the hidden layer and the bias.

The boat system operates in an epochwise manner with the initial position determined randomly and the final position specified as the iteration prior to the boat hitting the dock's shore. For this reason, one of the two epochwise algorithms should be used to train the neural controller. Because of the computational efficiency of the epochwise Euler-Lagrange based algorithm, it was selected for training the controller. In order to make the boat come near to the dock at the final iteration, the following error function was used

$$E = (x_d - x_{k_f})^2 + (y_d - y_{k_f})^2$$

where  $x_d$  is the  $x$  position of the dock and  $y_d$  is the  $y$  position of the dock.

In order to train the neural controller, 4000 thousand training epochs were required with a learning rate,  $\mu = 0.0001$ . After training, four demonstration epochs, which are shown in Figure 8, were run. In the lower portion of Figure 8, the current is shown as a function of  $x$ . In order to show the boat graphically, it was necessary to move the two shores outward a distance equal to half the boat length. For this reason, the current near both shores is shown as zero. The four demonstration epochs show that by using the Euler-Lagrange based algorithm, it is possible to design a neural controller for the boat system.

## 9.2 Adaptive Filtering Example

In this example, an adaptive noise cancelling system was used to reduce additive noise from a corrupted signal. Before getting into the details of this example, the adaptive noise cancelling concept is introduced. Whenever an adaptive noise cancelling system is to be used, it is assumed that it is possible to detect a noise source,  $r_k$ , which corrupts the original signal,  $s_k$ . Furthermore, it is assumed that a filter version of the noise,  $n_k$ , corrupts the original signal. Finally, it is assumed that the noise signal and the original signal are uncorrelated. The adaptive noise cancelling system receives as input the noise signal,  $r_k$ , and the corrupted signal,  $s_k + n_k$ . In order to eliminate the filtered noise from the corrupted signal, the noise signal is adaptively filtered and the result,  $y_k$ , is subtracted from the corrupted signal. If the adaptive filter is appropriately trained so that  $y_k = n_k$ , this subtraction will result in the output of the noise cancelling system,  $\epsilon_k$ , being equal to the original signal. Figure 9 shows an illustration of the basic noise cancelling system.

We have stated earlier that an on-line error function of the form  $(d_k - y_k)^2$  minimizes  $E[(d_k - y_k)^2]$ . For the adaptive noise cancelling system, we select an error function of the form  $\epsilon_k^2$ . Therefore, on-line adaptation of the system results in the minimization of  $E[\epsilon_k^2]$ . We can find this quantity by expanding the expected values of  $\epsilon_k$  as follows

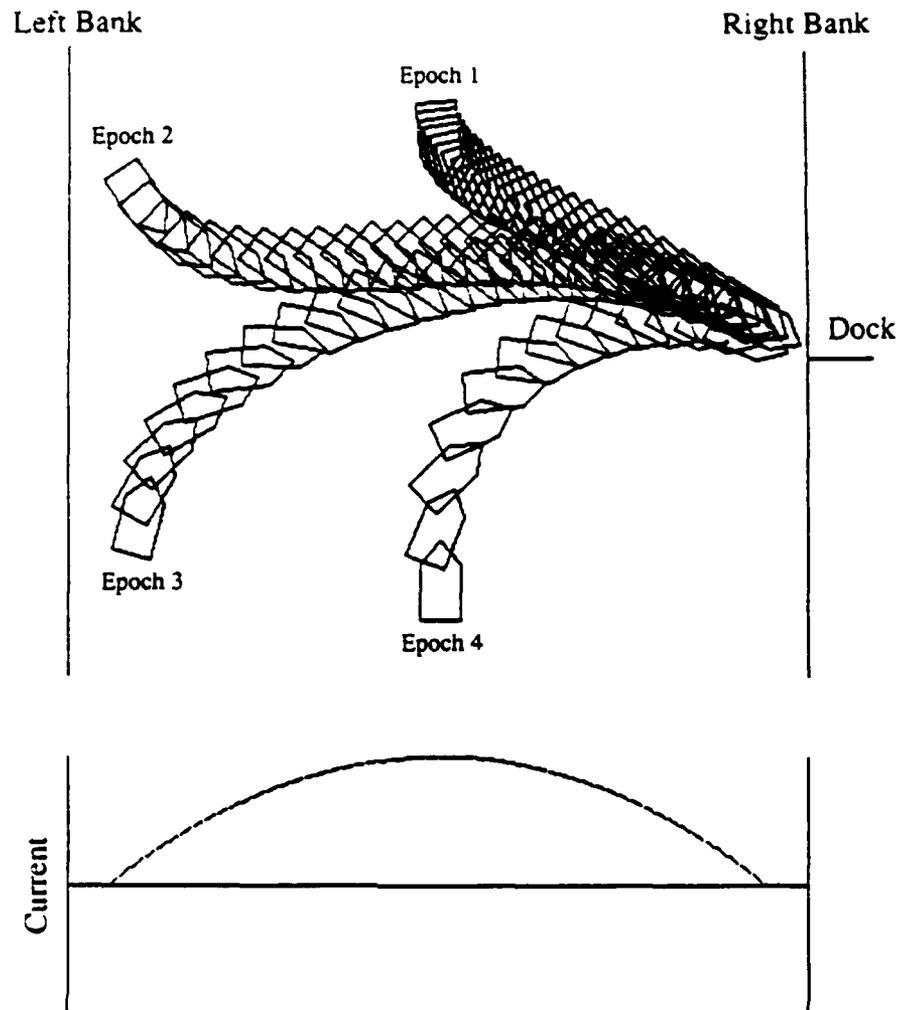


Figure 8: Nonlinear Control Example.

$$\mathbf{E}\{\epsilon_k^2\} = \mathbf{E}\{(s_k + n_k - y_k)^2\} \quad (49)$$

$$= \mathbf{E}\{s_k^2 + 2s_k(n_k - y_k) + (n_k - y_k)^2\}. \quad (50)$$

Assuming the original signal is uncorrelated with the noise signal and the adaptive filter output, Equation 50 can be written as

$$\mathbf{E}\{\epsilon_k^2\} = \mathbf{E}\{s_k^2\} + \mathbf{E}\{(n_k - y_k)^2\}. \quad (51)$$

Minimization of Equation 51 requires that  $n_k = y_k$ . Therefore, by using an on-line error function of the form  $E_k = \epsilon_k^2$ , the noise is adaptively eliminated from the corrupted signal by the adaptive noise cancelling system. For a more detail discussion of the adaptive noise cancelling concept, see Widrow and Stearns [9].

In our example, the original signal was

$$s_k = .25\cos(.4k). \quad (52)$$

The noise signal,  $r_k$ , is selected randomly from a uniform distribution between -1.0 and 1.0. The filtered noise,  $n_k$ , is calculated using the following nonlinear difference equation

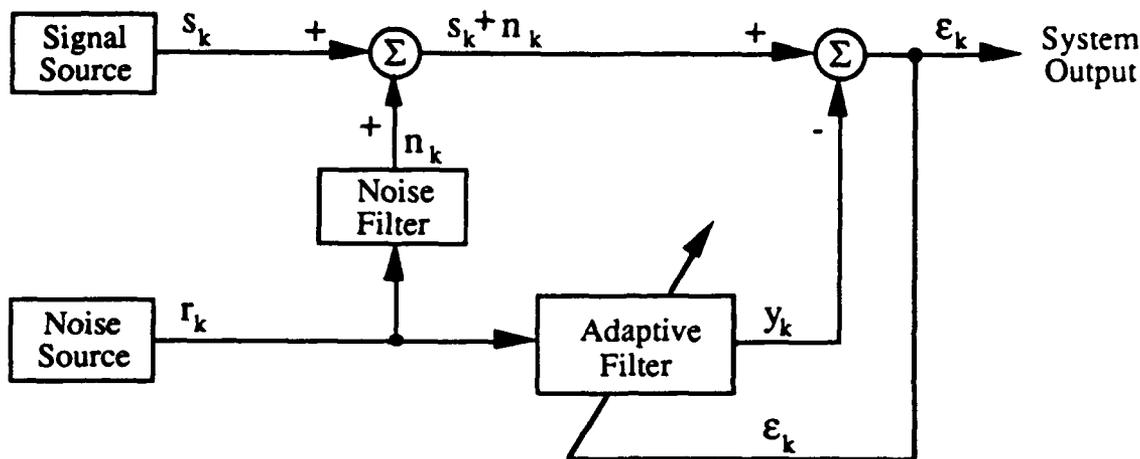


Figure 9: Adaptive Noise Cancelling System.

$$n_k = r_k + f_n(n_{k-1}) \quad (53)$$

where

$$f_n(n_{k-1}) = .5exp\left(\frac{-(n_{k-1} - 1.0)^2}{0.67}\right) + .5exp\left(\frac{-(n_{k-1} + 1.0)^2}{0.67}\right) \quad (54)$$

It should be noted that the noise filter contained nonlinear feedback.

The adaptive filter was implemented by a three layer feedforward neural network. The input layer was composed of two components, the noise signal,  $r_k$ , and the previous output of the adaptive filter,  $y_{k-1}$ . The hidden layer was composed of 17 hidden units each of which were squashed by the sigmoidal function. The first five nodes were connected through five different weights to the noise signal. The remaining ten nodes were connected to the feedback signal,  $y_{k-1}$ . In addition, each of the hidden units were connect to a bias. The output layer contained one linear unit which was connected to the hidden nodes and the bias through 18 separate weights.

One of the primary reasons for selecting the adaptive noise cancelling system as an example is that the feedback adaptive filter described above can only be trained using one of the on-line learning algorithms discussed in Section 5. The speed-up technique of feeding back the desired response cannot be used for this example because a desired response does not exist. The on-line recursive gradient update algorithm was selected for training the adaptive filter because it is computationally more efficient than the on-line Euler-Lagrange based algorithm when the number of outputs,  $N$ , and the number of delays,  $L$ , are both equal to 1.

A learning curve for the system, with the learn rate,  $\mu = .005$ , and the forgetting factor,  $\alpha = .95$ , is shown in Figure 10. The initial decrease in the mean squared error over the first couple hundred iterations is due to learning the feedforward component of the filter. The slow learning, which lasts for several thousand iterations, is due to learning the feedback component. The corrupted signal,  $s_k + n_k$ , and the original signal,  $s_k$ , for iterations 5900-6000 are shown in Figure 11. Notice that it is impossible to determine the characteristics of the original signal from the corrupted signal. The output signal,  $\epsilon_k$ , and original signal,  $s_k$ , for these same iterations are shown in Figure 12. Although the output signal is not perfect, the noise has been significantly reduced.

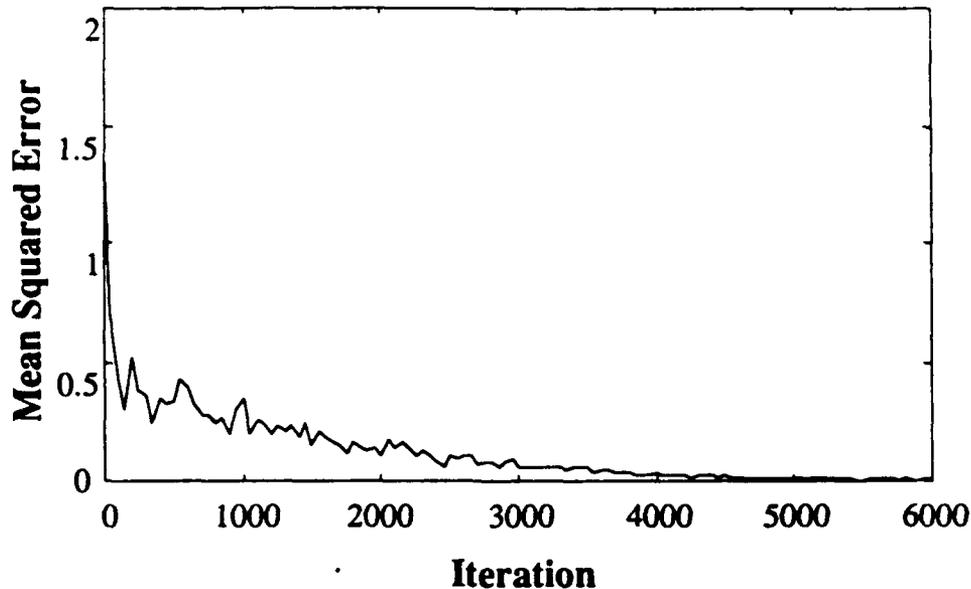


Figure 10: Learning Curve of the Noise Cancelling System.

## 10 Conclusion

The training of discrete-time dynamic systems using first-order gradient descent can be accomplished using either the Euler-Lagrange based algorithm or the recursive gradient update algorithm. Both these algorithms have been derived in this paper using the notation of the standard representation. Epochwise training can be accomplished using either of the two epochwise training algorithms which have been shown to produce identical weight updates. In general, because of both computational and storage requirements, the Euler-Lagrange based algorithm is preferable for epochwise training. However, the epochwise recursive algorithm may be desirable in cases where constant memory size is required. The two on-line algorithms produce approximately the same weight updates at each iteration. In general, the selection of an on-line algorithm is determined by the number of outputs,  $N$ , of the dynamic system. As this number increases, it becomes increasingly computationally expensive to use the recursive algorithm. Therefore, for large  $N$ , the Euler-Lagrange based algorithm is preferable for on-line training. Both on-line algorithms are computationally expensive. One method of reducing the computations is to feedback the desired responses, if they are available. Another method, which is applicable only to the Euler-Lagrange based technique, is to redefine the error function. Finally, two examples which illustrate the usefulness of the algorithms are presented. The first demonstrates the use of the Euler-Lagrange based algorithm for designing nonlinear state feedback controllers. The second illustrates the necessity of on-line algorithms in certain adaptive filtering problems.

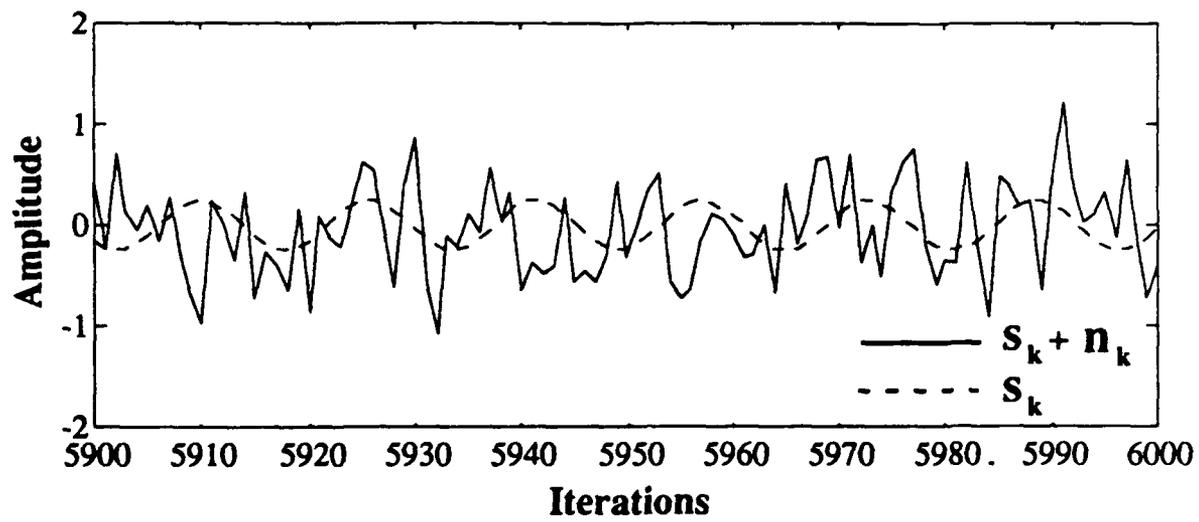


Figure 11: Corrupted Signal and Original Signal.

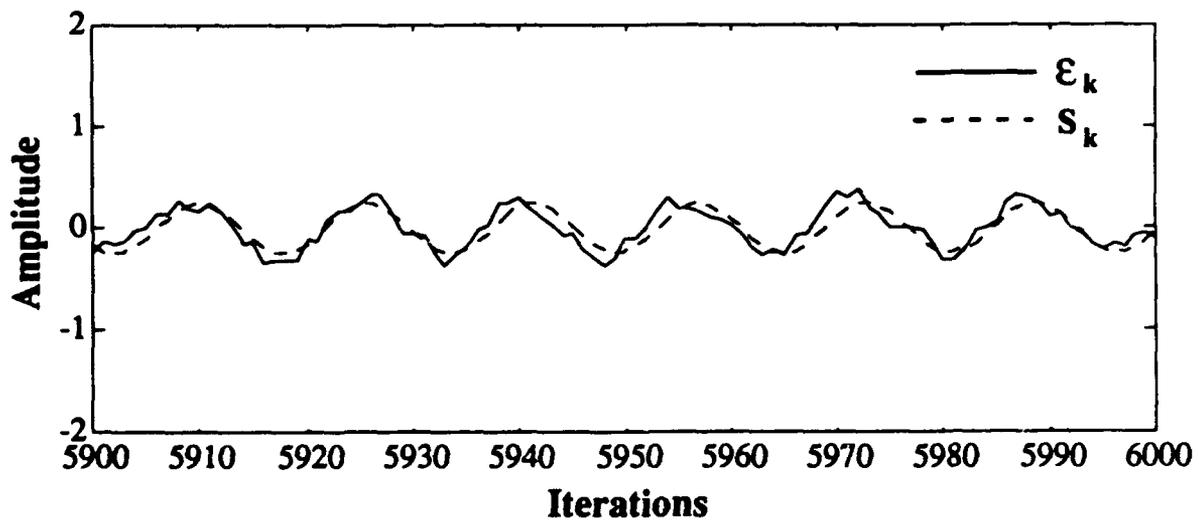


Figure 12: Output Signal and Original Signal.

## References

- [1] R. J. Williams and D. Zipser. A learning algorithm for continually running fully recurrent neural networks. *Neural Computation*, pages 270-277, Summer 1989.
- [2] B. Pearlmutter. Learning state space trajectories in recurrent neural networks. In *Proceedings of the International Joint Conference on Neural Networks*, volume II, pages 365-372, Washington, DC, June 1989.
- [3] D. Nguyen and B. Widrow. Neural networks for self-learning control systems. *IEEE Control Systems Magazine*, April 1990.
- [4] D. E. Rumelhart, G. E. Hinton, and R. J. Williams. Learning internal representations by error propagation. In D. E. Rumelhart and J. L. McClelland, editors, *Parallel Distributed Processing*, volume 1, chapter 8. The MIT Press, Cambridge, MA, 1986.
- [5] K. S. Narendra and K. Parthasarathy. Identification and control of dynamic systems using neural networks. *IEEE Transactions on Neural Networks*, pages 4-27, March 1990.
- [6] P. Werbos. *Beyond Regression: New Tools for Prediction and Analysis in the Behavioral Sciences*. PhD thesis, Harvard University, Cambridge, MA, August 1974.
- [7] R. J. Williams and D. Zipser. Gradient-based learning algorithms for recurrent connectionist networks. Technical Report NU-CCS-90-9, College of Computer Science, Northeastern University, Boston, MA 02115, April 1990.
- [8] M. Jordan. Generic constraints on underspecified target trajectories. In *Proceedings of the International Joint Conference on Neural Networks*, volume I, pages 217-225, Washington, DC, June 1989.
- [9] B. Widrow and S. D. Stearns. *Adaptive Signal Processing*. Prentice-Hall, Englewood Cliffs, NJ, 1985.
- [10] D. Andes, B. Widrow, M. Lehr, and E. Wan. MR3: A robust algorithm for training analog neural networks. In *Proceedings of the International Joint Conference on Neural Networks*, volume I, pages 533-536, Washington, DC, January 1990.
- [11] M. Holler, *et al.* An electrically trainable artificial neural network (etann) with 10240 "floating gate" synapses. In *Proceedings of the International Joint Conference on Neural Networks*, volume II, pages 191-196, Washington, DC, June 1989.
- [12] A. E. Bryson, Jr. and Y. Ho. *Applied Optimal Control*. Blaisdell Publishing Co., New York, 1969.
- [13] S. A. White. An adaptive recursive digital filter. In *Proc. 9th Asilomar Conf. Circuits Syst. Compt.*, page 21, Nov. 1975.
- [14] John J. Shynk. Adaptive IIR filtering. *IEEE ASSP Magazine*, April 1989.
- [15] R. J. Williams and J. Peng. An efficient gradient-based algorithm for on-line training of recurrent network trajectories. Technical Report NU-CCS-90-10, College of Computer Science, Northeastern University, Boston, MA 02115 1990.

**MISSION  
OF  
ROME LABORATORY**

*Rome Laboratory plans and executes an interdisciplinary program in research, development, test, and technology transition in support of Air Force Command, Control, Communications and Intelligence (C<sup>3</sup>I) activities for all Air Force platforms. It also executes selected acquisition programs in several areas of expertise. Technical and engineering support within areas of competence is provided to ESD Program Offices (POs) and other ESD elements to perform effective acquisition of C<sup>3</sup>I systems. In addition, Rome Laboratory's technology supports other AFSC Product Divisions, the Air Force user community, and other DOD and non-DOD agencies. Rome Laboratory maintains technical competence and research programs in areas including, but not limited to, communications, command and control, battle management, intelligence information processing, computational sciences and software producibility, wide area surveillance/sensors, signal processing, solid state sciences, photonics, electromagnetic technology, superconductivity, and electronic reliability/maintainability and testability.*