Gradient Descent Learning Algorithm Overview: A General Dynamical Systems Perspective

Pierre Baldi

Abstract— **We give a unified treatment of gradient descent learning algorithms for neural networks using a general framework of dynamical systems. This general approach organizes and simplifies all the known algorithms and results which have been originally derived for different problems (fixed point/trajectory learning), for different models (discrete/continuous), for different architectures (forward/recurrent), and using different techniques (backpropagation, variational calculus, adjoint methods, etc.). The general approach can also be applied to derive new algorithms. We then briefly examine some of the complexity issues and limitations intrinsic to gradient descent learning. Throughout the paper, we focus on the problem of trajectory learning.**

I. Introduction

LEARNING in biological systems is widely believed to result from progressive modifications occurring at the level of synapses connecting neurons. While the detailed biophysical mechanisms by which a given connection may become stronger or weaker are being intensively investigated, it is clear that ultimately such modifications can depend only on the local electrochemical environment of a given synapse. Obviously, a single synapse does not know anything about the global tasks the organism is trying to learn. Synaptic modifications and global tasks belong to two very distant levels of the brain hierarchy. Therefore, the fundamental question of learning is: What are the principles according to which local synaptic changes are organized in order to yield global learning of complex behaviors for the organism? This puzzling question remains largely unanswered. There are, however, two basic and somewhat complementary ideas in the theoretical literature which have shed some light on this question.

The main contribution to the origin of the first idea is usually attributed to Hebb [16]. Loosely speaking, this is the idea that if the activities of two connected neurons are correlated over time, then the strength of the corresponding synapse should tend to increase. If the activities are not correlated, then the strength should tend to decrease. This formulation is obviously very vague, and various ways exist by which Hebbian learning rules can be implemented in neural networks. It is well known that, although these rules are very simple and local, they can lead to powerful self-organization effects in networks of simple model neurons (see, for instance, Linsker [20]). Yet, it is also clear that a Hebb rule by itself cannot account for the learning of complex behaviors, and that some more global organizing principles are also required.

Gradient descent is the second main idea in the theory of learning, one which precisely attempts to offer a simple guiding principle for the overall organization of synaptic changes across networks. Whether something similar to gradient descent occurs in biological organisms is still very unclear. But the study of gradient descent learning remains worthwhile for at least two reasons: the numerous insights it sheds on the problems attached to learning, and for its applications in artificial neural systems to the solution of many practical problems. This paper is devoted to the study of gradient learning from an algorithmic point of view. In particular, we try to sort out all the different possible ways of actually implementing the basic gradient descent idea, and also examine some of its limitations.

Because gradient descent learning algorithms have been studied in the context of particular neural network models (discrete time, continuous time, additive model, higher order models, etc.), particular problems (fixed point learning, trajectory learning, etc.), and using different techniques (variational calculus, adjoint methods, numerical integration, etc.), the general underlying principles have been somewhat obscured. Thus, numerous and apparently different algorithms have been published in the literature. Some of the most recent examples include the backpropagation and backpropagation through time of Rumelhart *et al.* [26], the recurrent backpropagation of Pineda [25], and the algorithms of Williams and Zipser [34], Pearlmutter [24], and Toomarian and Barhen [30]. This list is not intended to be exhaustive: the same algorithm, including the simple backpropagation, has often been discovered independently by several authors at different times, and gradient descent ideas are, of course, quite ancient (see also Werbos [33]) and Bryson and Denham [7] along these lines). One of our goals here is to give a general and unified treatment of all these approaches in terms of general dynamical systems. It is our hope that such a treatment should illuminate the simple underlying structure of all the known algorithms, which become special cases of the general approach. Another advantage of the general approach is that it can be readily applied to new models or to new situations, for instance, the learning of parameters other than synaptic weights, such as gains, time constants, or delays.

For the purpose of generality, we want to consider the learning problem for an N-dimensional dynamical system of

Manuscript received April 4, 1992; revised October 19, 1992. This work was supported by grants from the AFOSR, the ONR, and the McDonnell–Pew Foundation.

The author is with the Jet Propulsion Laboratory and the Division of Biology, California Institute of Technology, Pasadena, CA 91109 USA.

IEEE Log Number 9205983.

the form

$$\frac{d\boldsymbol{u}}{dt} = \boldsymbol{F}(\boldsymbol{u}, W, \boldsymbol{I}, \boldsymbol{u}^*) \tag{1.1}$$

with initial condition $\boldsymbol{u}(t_0)$. In this relation, the N-dimensional vector $\boldsymbol{u}$ represents the state variables of the system, W an array of adjustable parameters, and $\boldsymbol{I}$ a vector of external inputs. When present, the vector $\boldsymbol{u}^*$ is used to represent particular target values for some of the $\boldsymbol{u}$ variables. In neural network terminology, the $\boldsymbol{u}$'s describe the activity or internal membrane potential of the N neurons comprising the network. The parameters W consist mainly of connection weights between the units, but may also include their time constants, their gains, as well as the time delays existing between them. In some situations, one may require that the initial conditions themselves be adjustable. All the results to be derived can also be presented using the slightly different but equivalent framework of discrete dynamical systems. For completeness, these are discussed in the Appendix, where it is shown how to translate learning algorithms from one formalism to the other.

In this framework, the learning problem consists of adjusting the parameters W so that the trajectories of (1.1) have certain specified properties. Gradient descent learning requires that, at any time, the performance of the dynamical system be assessable through a certain error function E which measures the discrepancy between the trajectories of the dynamical system and the desired behavior. The essence of gradient learning consists of iteratively adjusting the dynamical parameters in the direction opposite to the gradient of E, so as to reduce this discrepancy according to

$$\frac{dw}{dt} = -\eta \frac{\partial E}{\partial w} \tag{1.2}$$

or some approximation to it. η is the usual learning rate or the relaxation time constant of the parameter dynamics. It is also possible to have different learning rates for different parameters, or even learning rates which are themselves adaptable (see, for instance, Fang and Sejnowski [10]). The results to be derived extend immediately to these cases.

In the case of the most commonly used form of supervised gradient descent learning, the target trajectories of the system are supposed to be known and available for training. In particular, the target trajectories $\boldsymbol{u}^*$ can be incorporated in the error function, for instance, by using the usual least mean-squares error function. In the case of supervised fixed-point learning, the dynamical system should converge to a prescribed fixed point $\boldsymbol{u}^*$. In the more general case of supervised trajectory learning, the dynamical system should follow a prescribed trajectory $\boldsymbol{u}^*(t)$ over a certain time interval $[t_0, t_1]$. The target fixed point $\boldsymbol{u}^*$ or the target trajectory $\boldsymbol{u}^*(t)$ is usually not defined for all the N units and at all times, but only on a subset of the units, the visible units, and possibly only at certain times. The error function is computed only at the visible units and at the times where error information is available. In the most general gradient descent setting, however, the target trajectories need not to be known in advance or be explicitly present in the error function E.[1] In practice, it is usually desirable to learn several trajectory patterns at the same time so that $E = \sum_p E_p$, where E_p is the error for the pth pattern. Since $\partial E/\partial w = \sum_p \partial E_p/\partial w$, it is sufficient to study the computation of the gradient for only one pattern. Accordingly, in what follows, we shall assume that the system is learning only one trajectory pattern.[2]

Most of the neural network models discussed in the literature are simple special cases of this general framework. It is helpful, for concreteness, to keep in mind some of the particular models to which the theory can be applied. In general, these can be viewed as simple one-compartment models for the neuron charging equations.

A. Examples of Neural Dynamical Systems

The usual additive model (see, for instance, Amari [1], Sejnowski [27], Grossberg and Cohen [14], and Hopfield [18]) is just a one-compartment model for each neuron and can be described by

$$\frac{du_i}{dt} = -\frac{u_i}{\tau_i} + \sum_j w_{ij} f_j(u_j) + I_i \tag{1.3}$$

where τ_i is the time constant of the ith unit and f_i its input–output transfer function, usually a sigmoid with gain g_i. The form of the transfer function is, however, irrelevant in most of what follows, and the equations to be derived can be applied also to the case of nonsigmoidal transfer functions such as exponential, Gaussian, or logarithmic units and the like.

A different version of the additive model is given by

$$\frac{du_i}{dt} = -\frac{u_i}{\tau_i} + \alpha_i f_i\left(\sum_j w_{ij} u_j\right) + I_i. \tag{1.4}$$

It is easy to check that, provided all the time constants are identical ($\tau_i = \tau$) and the matrix (w_{ij}) is invertible, the two systems (1.3) and (1.4) are equivalent under a simple linear transformation. Indeed if, for instance, in (1.4) we make the transformation $r_i = \sum_j w_{ij} u_j$, we immediately obtain a system for the vector r similar to (1.3). The discretization of (1.4), with $\tau_i = \alpha_i = \Delta t = 1$, leads immediately to the usual connectionist models of Rumelhart *et al.* [26].

In some cases (Williams and Zipser [34], Baldi and Pineda [4]), it is useful to include a teacher signal (or error signal) in the charging equation of some of the units of the additive model, for instance, in the form

$$\frac{du_i}{dt} = -\frac{u_i}{\tau_i} + \sum_j w_{ij} f_j(u_j) + I_i + \beta_i (u_i - u_i)^{\gamma_i}. \tag{1.5}$$

The additive model can also easily be modified to take into account higher order interactions. In the case of certain models of shunting inhibition, for instance, one can have terms of the

[1] Contrastive learning (Baldi and Pineda [4]), which is a generalization of Boltzmann machine learning, is also a special case of gradient descent learning on a particular type of error function. So far, it has mainly been applied only to the case of fixed-point learning, and therefore will not be considered here in detail.

[2] The issue of batch learning versus on-line learning, i.e., whether (1.2) can be approximated by $dw/dt = -\eta \partial E\rho/\partial w$, is outside the focus of this paper.

form $(1 - f_k(u_k))w_{ij}f_j(u_j)$ where the kth unit can control the j to i connection. If the kth unit is active, $f_k(u_k)$ is close to one, and the j to i connection is blocked. The more general additive system with, for instance, only first- and second-order interactions can be written in the form

$$\frac{du_i}{dt} = -\frac{u_i}{\tau_i} + \sum_{j,k} w_{ijk} f_j(u_j) f_k(u_k) + \sum_j w_{ij} f_j(u_j) + I_i. \tag{1.6}$$

An example of a different class of neural dynamical systems consists of networks of coupled oscillators (see, for instance, Baldi and Meir [3]) or directional spins. These are modeled in the form

$$\frac{du_i}{dt} = \omega_i + \sum_j w_{ij} \sin(u_j - u_i) \tag{1.7}$$

where the variables u_i represent the phases of the corresponding oscillators. The general approach can also be applied to more complex models, such as Fitzugh [12] and Hindmarsh and Rose [17], where several coupled differential equations are used to model the behavior of a single neuron and its spiking activity.

Time delays seem to play an essential role in natural neural systems (for instance, Carr and Konishi [8]), and can easily be taken into account in any of the previous models by introducing terms of the form $\boldsymbol{u}(t-\tau)$ in the corresponding differential equations. In the most general case, the introduction of time delays leads to somewhat more complicated notations because time delays may not be symmetric and because the current state of a synapse may depend on several events having occurred in the neighboring neurons at different times. For instance, in the case of second-order interactions, one may have a charging equation of the form

$$\frac{du_i}{dt} = -\frac{u_i}{\tau_i} + \sum_{j,k,P} w_{ijkP} f_j(u_j(t - \tau_{ijkP})) \cdot f_k(u_k(t - \tau_{ikjP})) + \sum_{j,Q} w_{ijQ} f_j(u_j(t - \tau_{ijQ})) + I_i \tag{1.8}$$

where the first index of a delay refers to the postsynaptic neuron, the second index to the neuron with corresponding delayed activity in the interaction, the following indexes to the other neurons participating in the synaptic interaction, and finally the last index to the corresponding event in the past. In most cases, only the most recent event in the past plays a role in the current charging equation. A more general formulation of delayed networks in terms of convolution kernels, which does not need discretized delays, is possible (see, for instance, de Vries and Principle [9]). For a first-order additive model, for example, we can write

$$\frac{du_i}{dt} = -\frac{u_i}{\tau_i} + \int_0^t \sum_j w_{ij}(t-s) f_j(u_j(s))\, ds + I_i. \tag{1.9}$$

This, however, requires the introduction of a time-dependent matrix of synaptic interactions, and results have been derived only in the most simple cases which often are essentially reducible to discrete delays.

In Sections II and III, we derive the gradient descent learning algorithms for fixed points and trajectories in the framework of general dynamical systems. In both cases, the computation of the gradient depends crucially on the solution of a linear system with matrix L, where L is the Jacobian matrix of the function $\boldsymbol{F}$ which governs the dynamical system

$$L = (L_{ij}) = \left(\frac{\partial F_i}{\partial u_j}\right). \tag{1.10}$$

In fixed-point learning, it is a linear system of equations evaluated at equilibrium, and basically L needs to be inverted. In trajectory learning, it is a time-dependent linear system of differential equations which needs to be integrated over time. Different algorithms are possible, depending on how these linear systems are solved. In both cases, one of the algorithms is based on the solution of an auxiliary linear system (the adjoint system) governed by the transpose L^t of L. It is this transposition which is the signature of "backpropagation." We briefly examine the connection of these algorithms to the "backpropagation through time" concept of Rumelhart *et al.* [26]. We then extend the algorithms to the case of networks with time delays, and where, in addition to the synaptic weights, the delays themselves and the other temporal parameters may also be adjustable and subject to learning. In Section IV, we examine the complexity of the algorithms and some of the limitations inherent to gradient descent procedures, especially for recurrent networks. Several technical results used in the sections are deferred to the Appendix.

Throughout the paper, $\boldsymbol{v} = (v_i)$ or $M = (M_{ij})$ will denote a vector or a matrix (for readability, vector quantities are in boldface). $\|\boldsymbol{v}\|$ is the Euclidean norm of the vector $\boldsymbol{v}$. All vectors are column vectors, and $\boldsymbol{v}^t$ or M^t represent the transpose of the corresponding vector or matrix. The adjoint of a matrix M is the transpose of the matrix whose entries are the complex conjugate of the entries of M, i.e., the adjoint of M is the matrix $\overline{M}^t = (\overline{m}_{ij})^t$. Because all the matrices considered here have real entries, adjoint and transpose matrices are identical. Unless otherwise specified, all functions (in particular, $\boldsymbol{F}$ and f_i) are supposed to be continuously differentiable in all their variables over the range of interest. This ensures the existence and unicity of the solutions of the differential equations considered, and allows certain manipulations such as interchanging the order of differentiation in mixed partial derivatives. Parameters such as synaptic weights or delays require multiple indexes reflecting the interaction of several neurons. By convention, the first index always characterizes the unique equation in which this parameter appears. Thus, a parameter such as $w_i \cdots$ appears only in the charging equation of the ith unit, i.e., in F_i. Finally, if an interval such as $[t_0, t_1]$ is discretized into intervals of length Δt, then for any function $g(t)$, we shall write $g(m) = g(t_0 + m\Delta t)$.

II. Learning Fixed Points

In this section, we assume that the system defined by (1.1) with fixed initial condition $\boldsymbol{u}(t_0)$ is convergent, and remains so in the course of learning. Accordingly, all the

quantities to be considered will be evaluated at equilibrium. The system converges to a fixed point $\boldsymbol{u}^f$ which is a function of the parameters W, for fixed initial conditions and fixed external input. The goal is to modify the parameters so that $\boldsymbol{u}^f$ approaches a minimum of E. In the special case of supervised learning, $\boldsymbol{u}^f$ should approach a prescribed target value $\boldsymbol{u}^*$ defined for the visible units of the networks. The following derivation is unchanged whether the final state is to be considered a function of the external input or a function of the initial state. The error function is of the form

$$E = E(\boldsymbol{u}^f, \boldsymbol{u}^*). \tag{2.1}$$

For instance, in the usual LMS case,

$$E = \frac{1}{2}(\boldsymbol{u}^f - \boldsymbol{u}^*)^2 \tag{2.2}$$

computed over the visible units. The fixed point $\boldsymbol{u}^f$ satisfies the relation

$$\boldsymbol{F}(\boldsymbol{u}^f, w, \boldsymbol{I}, \boldsymbol{u}^*) = 0. \tag{2.3}$$

To update the weights, we need to compute the gradient

$$\frac{\partial E}{\partial w} = \sum_i \frac{\partial E}{\partial u_i^f}\frac{\partial u_i^f}{\partial w} = \left(\frac{\partial E}{\partial \boldsymbol{u}^f}\right)^t \frac{\partial \boldsymbol{u}^f}{\partial w} = \left(\frac{\partial E}{\partial \boldsymbol{u}^f}\right)^t \boldsymbol{p}_{\cdot w} \tag{2.4}$$

($\partial E/\partial u_i^f = 0$ for nonvisible units) where the coordinates

$$p_{iw} = \frac{\partial u_i^f}{\partial w} \tag{2.5}$$

of the vector $\boldsymbol{p}_{\cdot w}$ represent how the coordinates u_i^f of the fixed point vary with a small change in w. In control theory, the numbers p_{iw} are often called sensitivities. To compute $\boldsymbol{p}_{\cdot w}$, we differentiate the fixed-point equation (2.3) to get

$$\sum_j \frac{\partial F_i}{\partial u_j^f} p_{jw} + \frac{\partial F_i}{\partial w} = 0. \tag{2.6}$$

where the vector $\partial \boldsymbol{F}/\partial w = (\partial F_i/\partial w)$ results from the *explicit* dependence of F_i on w. In vector notation,

$$L\boldsymbol{p} \cdot w + \frac{\partial \boldsymbol{F}}{\partial w} = 0 \tag{2.7}$$

or, with the mild assumption that L be invertible,

$$\boldsymbol{p} \cdot w = -L^{-1}\frac{\partial \boldsymbol{F}}{\partial w} \tag{2.8}$$

where $L = (\partial F_i/\partial u_j^f)$ is the Jacobian matrix at the equilibrium point. Then, from (2.4) and (2.8),

$$\begin{aligned}\frac{\partial E}{\partial w} &= -\left(\frac{\partial E}{\partial \boldsymbol{u}^f}\right)^t L^{-1}\frac{\partial \boldsymbol{F}}{\partial w} \\ &= -\left(\frac{\partial \boldsymbol{F}}{\partial w}\right)^t (L^t)^{-1}\frac{\partial E}{\partial \boldsymbol{u}^f}.\end{aligned} \tag{2.9}$$

Now, a few alternatives exist, depending on how one chooses to deal with the inversion of the matrices on the right-hand side of (2.9), and several observations can be made. First, what is really required is the calculation of the product $L^{-1}\partial \boldsymbol{F}/\partial w$ or $(L^t)^{-1}\partial E/\partial \boldsymbol{u}^f$. The second product $(L^t)^{-1}\partial E/\partial \boldsymbol{u}^f$ is preferable because *it does not depend directly on the parameter* w, and therefore can be calculated once for all, and used in the evolution equation of every parameter throughout the network. If so, we can write

$$\frac{\partial E}{\partial w} = \left(\frac{\partial \boldsymbol{F}}{\partial w}\right)^t \boldsymbol{v} \tag{2.10}$$

with $\boldsymbol{v} = -(L^t)^{-1}\partial E/\partial \boldsymbol{u}^f$ or, using (1.2),

$$\frac{dw}{dt} = -\eta\left(\frac{\partial \boldsymbol{F}}{\partial w}\right)^t \boldsymbol{v}. \tag{2.11}$$

Second, the vector $\partial \boldsymbol{F}/\partial w$ is very sparse because it contains at most one nonzero component because of our convention on the indexes of the neural parameters. (The vector $\partial E/\partial \boldsymbol{u}^f$ can also be sparse if the proportion of visible units is small.) Therefore, if we write $w = w_i \cdots$,

$$\frac{dw_i \cdots}{dt} = -\eta \frac{\partial F_i}{\partial w_i \cdots} v_i \tag{2.12}$$

or

$$\frac{dw_i \cdots}{dt} = \eta \frac{\partial F_i}{\partial w_i \cdots} \sum_j (L^{-1})_{ji} \frac{\partial E}{\partial u_j^f}. \tag{2.13}$$

In this form, the right-hand side of the parameter update equation is the product of two terms: a usually presynaptic activity term $\partial F_i/\partial w_i \cdots$ and an error-propagated term v_i. This learning rule could be considered Hebbian only in a very limited sense since some special machinery must be required to propagate the errors. The definition of the vector $\boldsymbol{v}$ is based on the transpose L^t of L. It is this transposition which is the signature of "backpropagation" and reverts the orientation of the edges in the network. Indeed, the vector $\boldsymbol{v}$ can be computed by a relaxation method as the equilibrium solution of the system

$$\frac{d\boldsymbol{v}}{dt} = L^t\boldsymbol{v} + \frac{\partial E}{\partial \boldsymbol{u}^f}. \tag{2.14}$$

(Notice that since the dynamical system is convergent, all the eigenvalues of L at $\boldsymbol{u}^f$ have negative real parts. Since L and L^t have the same eigenvalues, (2.14) is also convergent.) This transposition also appears in the next section on trajectory learning and adjoint methods. In the case of feedforward networks, the matrices L and L^t are triangular and the calculations are somewhat simpler. In the Appendix, we give a derivation from (2.10) of the usual backpropagation equations for connectionist feedforward networks. One can then see that $\boldsymbol{v}$ is the usual backpropagated error. When (2.12) and (2.14) are applied to the version of the additive model defined by (1.4), one obtains exactly the recurrent backpropagation algorithm discussed in Pineda [25] (see also Farotimi *et al.* [11] for another general derivation based on optimal control theory).

III. Learning Trajectories

In the more general case of trajectory learning, the goal is to modify the parameters so that the output trajectory minimizes some measure of poor performance. In the supervised case, as usual, $\boldsymbol{u}$ should follow a prescribed target trajectory $\boldsymbol{u}^*$ for the

visible units over a time interval $[t_0, t_1]$. This is a well-defined task, although it is a simplification of what may happen in real situations where one may want to learn several trajectories or even a flow. Furthermore, the problems of stability and phase along the trajectory and whether the trajectory is an attractor are left out for the moment and will be partly addressed in Section IV. The error function is of the form

$$\boldsymbol{E} = \int_{t_0}^{t_1} E(\boldsymbol{u}, \boldsymbol{u}^*, t)\, dt. \tag{3.1}$$

Although $\boldsymbol{E}$ is a scalar, a boldface character is used to distinguish it from the instantaneous error $E(\boldsymbol{u}, \boldsymbol{u}^*, t)$. For instance, in the LMS case

$$\boldsymbol{E} = \frac{1}{2}\int_{t_0}^{t_1} (\boldsymbol{u}^*(t) - \boldsymbol{u}(t))^2\, dt \tag{3.2}$$

computed over the visible units and possibly over the subtime intervals at which $\boldsymbol{u}^*(t)$ is defined. To update the weights, we need to compute the gradient

$$\frac{\partial \boldsymbol{E}}{\partial w} = \int_{t_0}^{t_1} \sum_i \frac{\partial E}{\partial u_i}\frac{\partial u_i}{\partial w}\, dt = \int_{t_0}^{t_1} \left(\frac{\partial E}{\partial \boldsymbol{u}}\right)^t \frac{\partial \boldsymbol{u}}{\partial w}\, dt$$
$$= \int_{t_0}^{t_1} \left(\frac{\partial E}{\partial \boldsymbol{u}}\right)^t \boldsymbol{p}_{\cdot w}\, dt \tag{3.3}$$

($\partial E/\partial u_i = 0$ for nonvisible units) where the coordinates

$$p_{iw}(t) = \frac{\partial u_i(t)}{\partial w} \tag{3.4}$$

of the sensitivity vector $\boldsymbol{p}_{\cdot w}(t)$ represent how the coordinates u_i at time t vary with a small change in w. To compute $\boldsymbol{p}_{\cdot w}$, we have

$$p_{iw}(t) = \frac{\partial}{\partial w}\int_{t_0}^{t} \frac{du_i}{ds}\, ds = \int_{t_0}^{t} \frac{\partial F_i}{\partial w}\, ds \tag{3.5}$$

or, by differentiating,

$$\frac{dp_{iw}(t)}{dt} = \frac{\partial F_i(\boldsymbol{u}, W, \boldsymbol{I}, \boldsymbol{u}^*)}{\partial w} = \sum_j \frac{\partial F_i}{\partial u_j}\frac{\partial u_j}{\partial w} + \frac{\partial F_i}{\partial w} \tag{3.6}$$

where $\partial F_i/\partial w$ denotes again the explicit derivative. In vector notation,

$$\frac{d\boldsymbol{p}_{\cdot w}}{dt} = L\boldsymbol{p}_{\cdot w} + \frac{\partial \boldsymbol{F}}{\partial w} \tag{3.7}$$

where L is the time-dependent Jacobian matrix $L = L(t) = (\partial F_i(t)/\partial u_j)$. The similarity of this derivation to the one obtained in the case of fixed-point learning should be stressed. In particular, in the case of fixed-point learning, $dp_{iw}(t)/dt = 0$ at equilibrium and (3.7) yields immediately (2.7). If the choice of the initial condition $\boldsymbol{u}(t_0)$ is independent of the parameters w, as is most often the case, then the initial condition for the system (3.7) is $p_{iw}(t_0) = 0$. Now, to evaluate (3.3), different algorithms are possible, depending on how one deals with the linear system of differential equations (3.7).

A. Numerical Integration of the System

The first possible algorithm is derived by simply integrating (3.7) forward in time and using (3.3). If the interval $[t_0, t_1]$ is discretized into M small steps of length Δt, then $\boldsymbol{p}_{\cdot w}(m)$ can be evaluated recursively through the relation

$$\boldsymbol{p}_{\cdot w}(m+1) = \Delta t\left[L(k)\boldsymbol{p}_{\cdot w}(m) + \frac{\partial \boldsymbol{F}}{\partial w}(m)\right] + \boldsymbol{p}_{\cdot w}(m). \tag{3.8}$$

The parameters can be updated by using (3.3) in the form

$$\frac{\partial \boldsymbol{E}}{\partial w} \approx \sum_{m=1}^{M} \left(\frac{\partial E}{\partial \boldsymbol{u}}(m)\right)^t \boldsymbol{p}_{\cdot w}(m)\Delta t. \tag{3.9}$$

When (3.8) and (3.9) are applied with $\Delta t = 1$ to the discretized version of (1.4), one immediately obtains the algorithm discussed by Williams and Zipser [34] (see also Appendix). This algorithm amounts to a direct computation of each partial derivative using the definition $\partial \boldsymbol{E}/\partial w = \lim(\boldsymbol{E}(w+h) - \boldsymbol{E}(w))/h$ as $h \to 0$, i.e., by slightly perturbing each weight, one at a time, leaving all the others fixed. This method can also be applied to conventional feedforward networks. Its main disadvantage lies in the fact that a lot of computations are required since each weight is dealt with separately. One advantage is that both unit and parameter dynamics can be evolved forward simultaneously. If the learning rate is small enough, it is also possible to approximate the gradient descent process by updating the parameters after each time step, instead of every M steps, resulting in an on-line learning procedure.

B. Explicit Solution of the System

Because (3.7) is linear, its solution can be written explicitly (see Section B in the Appendix). Indeed, if we take, for instance, $\boldsymbol{p}_{\cdot w}(t_0) = 0$, then

$$\boldsymbol{p}_{\cdot w}(t) = \int_{t_0}^{t} K(t, s)\frac{\partial \boldsymbol{F}}{\partial w}(s)\, ds \tag{3.10}$$

where $K(t, s) = H(t)H^{-1}(s)$, and $H(t)$ is an $N \times N$ invertible matrix whose columns form a set of N independent solutions of the homogeneous system associated with (3.7). Thus, from (3.3),

$$\frac{\partial \boldsymbol{E}}{\partial w} = \int_{t_0}^{t_1}\int_{t_0}^{t} \left(\frac{\partial E}{\partial \boldsymbol{u}}(t)\right)^t H(t)H^{-1}(s)\frac{\partial \boldsymbol{F}}{\partial w}(s)\, ds\, dt \tag{3.11}$$

or, by transposing as in (2.9),

$$\frac{\partial \boldsymbol{E}}{\partial w} = \int_{t_0}^{t_1}\int_{t_0}^{t} \left(\frac{\partial F}{\partial \boldsymbol{u}}(s)\right)^t (H^t(s))^{-1}H^t(t)\frac{\partial E}{\partial \boldsymbol{u}}(t)\, ds\, dt. \tag{3.12}$$

By first interchanging the order of integration in (3.11) [or (3.12)] and then exchanging the variables s and t, we successively get

$$\frac{\partial \boldsymbol{E}}{\partial w} = \int_{t_0}^{t_1}\int_{s}^{t_1} \left(\frac{\partial E}{\partial \boldsymbol{u}}(t)\right)^t K(t, s)\frac{\partial \boldsymbol{F}}{\partial w}(s)\, dt\, ds$$
$$= \int_{t_0}^{t_1} (\boldsymbol{x}(t))^t \frac{\partial \boldsymbol{F}}{\partial w}(t)\, dt \tag{3.13}$$

which is the time-dependent version of (2.10) and where

$$\boldsymbol{x}(t) = \int_t^{t_1} K(s, t)^t \frac{\partial E}{\partial \boldsymbol{u}}(s)\, ds. \tag{3.14}$$

As discussed in the Appendix, $K(s, t)^t$ is the transition matrix of a new system, called the adjoint system. Thus, (3.14) suggests that we look at the application of adjoint methods to our problem.

C. Adjoint Methods

The so-called adjoint methods rest on a very simple trick (see Appendix). To solve the system in (3.7), we construct an auxiliary N-dimensional system, the adjoint system, by introducing a new vector variable $\boldsymbol{v}$. The matrix of the auxiliary linear system of differential equations is the opposite of the adjoint of L, which in our case is just the transpose of L. The adjoint system is then defined by

$$\frac{d\boldsymbol{v}}{dt} = -L^t\boldsymbol{v} - \frac{\partial E}{\partial \boldsymbol{u}}. \tag{3.15}$$

The reason for the choice of the nonhomogeneous term will soon become apparent. Notice that (3.15) is essentially the same as (2.14). Now, we multiply, on the left, both sides of (3.7) by $\boldsymbol{v}^t$ and of (3.15) by $\boldsymbol{p}^t.w$ and add the two resulting equations. After simple cancellations, this yields

$$\frac{d(\boldsymbol{v}^t\boldsymbol{p}_{.w})}{dt} = \boldsymbol{v}^t\frac{\partial \boldsymbol{F}}{\partial w} - \boldsymbol{p}^t.w\frac{\partial E}{\partial \boldsymbol{u}}. \tag{3.16}$$

From (3.3) and integrating both sides of (3.16), we get

$$\frac{\partial \boldsymbol{E}}{\partial w} = \int_{t_0}^{t_1} \boldsymbol{p}_{.w}^t \frac{\partial E}{\partial \boldsymbol{u}}\, dt = -(\boldsymbol{v}^t\boldsymbol{p}_{.w})_{t=t_1} + (\boldsymbol{v}^t\boldsymbol{p}_{.w})_{t=t_0} + \int_{t_0}^{t_1} \boldsymbol{v}^t\frac{\partial \boldsymbol{F}}{\partial w}\, dt. \tag{3.17}$$

Except for the boundary terms, we see that in the adjoint methods, the computation of the integral $\int_{t_0}^{t_1} \boldsymbol{p}_{.w}^t(\partial E/\partial \boldsymbol{u})\, dt$ is replaced by $\int_{t_0}^{t_1}(\boldsymbol{v}^t(\partial \boldsymbol{F}/\partial w)\, dt$. Whereas in the direct methods a linear system for $\boldsymbol{p}_{.w}$ must be solved *for each* w, in the adjoint approach, only *one* linear system for the variable $\boldsymbol{v}$ needs to be solved and used in the update of any of the parameters.

To deal with the boundary terms, it must be noticed that the boundary conditions for $\boldsymbol{v}$ in (3.15) can be chosen arbitrarily. Since, usually, as we have seen, we can choose $\boldsymbol{p}_{.w}(t_0) = 0$, one possible approach is to fix $\boldsymbol{v}(t_1) = 0$. In this case,

$$\frac{\partial \boldsymbol{E}}{\partial w} = \int_{t_0}^{t_1} \boldsymbol{v}^t\frac{\partial \boldsymbol{F}}{\partial w}\, dt \tag{3.18}$$

or, similar to (2.12),

$$\frac{dw_i \cdots}{dt} = -\eta\int_{t_0}^{t_1} v_i\frac{\partial F_i}{\partial w_i \cdots}\, dt. \tag{3.19}$$

The application of (3.15) and (3.19) to the version of the additive model defined by (1.4) yields immediately the algorithm discussed by Pearlmutter [24]. This approach requires that (3.15) be solved backwards in time with final condition $\boldsymbol{v}(t_1) = 0$. In the Appendix, it is shown that $K(s, t)^t$ is the transition matrix of the adjoint system. Therefore, by solving (3.15) backwards in time with final condition $\boldsymbol{v}(t_1) = 0$, one gets

$$\boldsymbol{v}(t) = -\int_{t_1}^{t} K(s, t)^t \partial E/\partial \boldsymbol{u}(s)\, ds.$$

Comparing to (3.14), we see that $\boldsymbol{x}(t) = \boldsymbol{v}(t)$, and therefore the algorithm derived by explicitly solving the sensitivity system is, in fact, identical to the algorithm we just derived using adjoint methods. As we shall see, the discretized version of this algorithm is also identical to the backpropagation through time of Rumelhart *et al.* [26], and the variable $\boldsymbol{x}(t) = \boldsymbol{v}(t)$ can be interpreted as a vector of backpropagated error signals. Because the integration is backwards, this algorithm, which computationally is the fastest, has been considered as being "noncausal," and thus probably not ideally suited for a collective implementation in a physical system. However, provided one is willing to pay the price of a matrix inversion, the solution of (3.15) with final boundary condition can still be obtained through a forward calculation in time. This is because the solution of a linear system depends linearly on the initial condition, and therefore the effect of the initial condition can be factored out. To be more precise, by using the theorem on the solution of linear systems, we have

$$\boldsymbol{v}(t) = G(t, t_0)\boldsymbol{v}(t_0) + \int_{t_0}^{t} G(t, s)\frac{\partial \boldsymbol{E}}{\partial \boldsymbol{u}}\, ds \tag{3.20}$$

where G is the usual forward state transition matrix [as already pointed out, $G(t, s) = K(s, t)^t$]. Thus, from (3.18), we can write

$$\frac{\partial \boldsymbol{E}}{\partial w} = \boldsymbol{v}^t(t_0)\int_{t_0}^{t_1} G^t(t, t_0)\frac{\partial \boldsymbol{F}}{\partial w}\, dt + \int_{t_0}^{t_1}\int_{t_0}^{t}\left(\frac{\partial E}{\partial \boldsymbol{u}}(s)\right)^t G^t(t, s)\frac{\partial \boldsymbol{F}}{\partial w}(t)\, ds\, dt. \tag{3.21}$$

Because $\boldsymbol{v}(t_0)$ factors out in (3.20), the integral terms in (3.19) can be computed in the forward mode. At the end of the process, we can use (3.20) and the final condition $\boldsymbol{v}(t_1) = 0$ to solve for $\boldsymbol{v}(t_0)$:

$$\boldsymbol{v}(t_0) = -G^{-1}(t_1, t_0)\int_{t_0}^{t_1} G(t, s)\frac{\partial \boldsymbol{E}}{\partial \boldsymbol{u}}\, ds. \tag{3.22}$$

For the sake of completeness, we can derive the discretized version of this algorithm. From (3.18),

$$\frac{\partial \boldsymbol{E}}{\partial w} \approx \sum_{m=1}^{M} \Delta t\boldsymbol{v}^t(m)\frac{\partial \boldsymbol{F}}{\partial w}(m). \tag{3.23}$$

Using (3.15), the vector $\boldsymbol{v}(m)$ satisfies the recurrence relation

$$\boldsymbol{v}(m+1) = A(m)\boldsymbol{v}(m) + \boldsymbol{b}(m) \tag{3.24}$$

where $A(m)$ is the $N \times N$ matrix

$$A(m) = (I - \Delta tL^t(m)) \tag{3.25}$$

and $\boldsymbol{b}(m)$ is the N-dimensional vector

$$\boldsymbol{b}(m) = -\Delta t\frac{\partial E}{\partial \boldsymbol{u}}(m). \tag{3.26}$$

Then, by induction,

$$\boldsymbol{v}(m) = C(m)\boldsymbol{v}(0) + \boldsymbol{d}(m) \tag{3.27}$$

where the matrix $C(m)$ is given by

$$C(m) = \prod_{j=m-1}^{0} A(j) \tag{3.28}$$

and the vector $\boldsymbol{d}(m)$ by

$$\boldsymbol{d}(m) = \boldsymbol{b}(m-1) + \sum_{l=0}^{m-2} \prod_{j=m-1}^{l+1} A(j)\boldsymbol{b}(l) \tag{3.29}$$

(the order of the terms in the products appearing in (3.28) and (3.29) is important since, in general, the matrices $A(j)$ do not commute). Therefore,

$$\frac{\partial \boldsymbol{E}}{\partial w} \approx \boldsymbol{v}^t(0) \sum_{m=1}^{M} \Delta t C^t(m) \frac{\partial \boldsymbol{F}}{\partial w}(m) + \sum_{m=1}^{M} \Delta t \boldsymbol{d}^t(m) \frac{\partial \boldsymbol{F}}{\partial w}(m). \tag{3.30}$$

Again, the sums in (3.30) can be computed entirely in feedforward mode and, at the end, we can find $\boldsymbol{v}(0)$ by inverting the matrix $C(M)$:

$$\boldsymbol{v}(0) = C^{-1}(M)(\boldsymbol{v}(M) - \boldsymbol{d}(M)) = -C^{-1}(M)\boldsymbol{d}(M). \tag{3.31}$$

Provided the parameter dynamics is slow with respect to the network dynamics, this algorithm can be implemented on-line in a way similar, but more economical, to the implementation of the algorithm derived above from the numerical integration of the system.

D. Backpropagation through Time

In their original paper on backpropagation, Rumelhart *et al.* [26] considered the problem of learning in discrete-time recurrent networks in terms of backpropagation through time. Their analysis was based on the observation that the time evolution of a recurrent network can be represented by a stack of snapshots of the same network taken at successive time intervals with the proper feedforward connections. (Notice that with the proper time discretization, time delays in this representation result in feedforward connections between distant elements of the stack.) Error signals can then be injected in the stack at different levels, both in space and time, and their effect on the update of each weight assessed through the usual backpropagation through space in the stack (i.e., layer by layer). Conversely, the usual backpropagation algorithm for feedforward networks can be seen as a backpropagation through time as soon as the time taken by each unit in the network to compute its value is taken into consideration. In other words, there is no real distinction between backpropagation through space and through time. What is then the relation of the different algorithms described above to backpropagation through time? All the different algorithms result from different ways of organizing the gradient calculation, i.e., the propagation of errors throughout the previous stack. If the propagation is started at the top of the stack as in the usual backpropagation, then backpropagation through time is identical, as shown in the Appendix, to the discretized version of the algorithms derived in the previous sections by explicitly solving the sensitivity system or by using adjoint methods. This is also the algorithm that can be derived by variational methods (see Bryson and Denham [7] and Appendix) or which has been suggested by Pearlmutter [24]. If, on the other hand, each weight is perturbed in isolation and the effect of each perturbation is propagated forward in the stack starting from the bottom, then one obtains the algorithm of Williams and Zipser [34] which corresponds to a forward numerical integration of the sensitivity equations. Thus, there seem to be only two basic ways of calculating the gradient in the literature we have reviewed. Several variations, however, seem possible, and an example has been given at the end of the section on adjoint methods. Clearly, there are tradeoffs between the amount of computation, the amount of memory required, and the time direction (forward or backward in time) followed by the different learning algorithms. Whether these tradeoffs and the theoretical limits of any algorithm can be formalized with precision, and whether other algorithms exist which can perform better or achieve these theoretical limits remain to be seen.

E. Gradient Learning of Time Constants, Gains, and Delays

To extend gradient descent learning for fixed points or trajectories to include time constants or gains in any of the models listed in the Introduction is entirely straightforward. Time constants and gains can be treated as any other parameter such as synaptic weights, and all the learning algorithms derived above can be applied without any modification. The treatment of delays requires a slightly more careful analysis (see also Baldi and Atiya [6]).

The presence of delays in a network can affect learning algorithms in two different ways. First, they affect the learning of other parameters such as synaptic weights. Second, delays may themselves be part of the adaptable parameters. Although there is no conclusive evidence for continuously adjustable delays in biology, there exist numerous examples of fine-tuned delays and delay lines (Carr and Konishi [8]; in this context, see also Lisberger and Sejnowski [21]). Moreover, many delays are subject to variations, for instance, during the growth of an organism. Certainly, several biophysical mechanisms can be envisioned to achieve adaptable delays. Whether in natural or artificial systems, delays could then be regarded as beneficial rather than detrimental for they could enlarge the family of means through which neural networks can tailor their own dynamics.

First, in the case of fixed-point learning, it is easy to see that delays have no effect on the gradient descent learning equations. This is because all the relevant expressions such as $\partial E/\partial \boldsymbol{u}^f$ or $\partial \boldsymbol{u}^f/\partial w$ are computed at equilibrium when the vector $d\boldsymbol{u}/dt$ is constantly 0. Thus, the delays do not appear in the learning equations at all. Of course, they affect the activation dynamics and can change the fixed-point associated with a given initial state, thus still influencing the learning dynamics in an undirect way.

For the case of trajectories, the learning equations need to be modified. To simplify the notation, we shall assume that neuron j can influence the charging equation of neuron i only through one term with corresponding delay τ_{ij}. This would, for instance, be the case for the simple model

$$\frac{du_i}{dt} = -\frac{u_i}{\tau_i} + \sum_j w_{ij} f_j(u_j(t - \tau_{ij})). \tag{3.32}$$

The derivation for the case of more general delays is similar. There are two ways in which delays can affect the previous learning algorithms. First, the equations for the sensitivities of the usual parameters need to be slightly modified to take the delays into account. Then, in the case where the delays themselves are learnable, new sensitivity equations need to be added for the adjustment of the delays. More precisely, when delays are present, (3.1)–(3.5) remain unchanged. However, in the case of (3.32), (3.6) needs to be modified in the form

$$\frac{dp_{iw}(t)}{dt} = -\frac{1}{\tau_i} p_{iw}(t) + \sum_j w_{ij} f_j'(u_j(t - \tau_{ij})) \cdot p_{jw}(t - \tau_{ij}) + \frac{\partial F_i}{\partial w} \tag{3.33}$$

where $\partial F_i/\partial w$ is the explicit derivative of F_i with respect to w, i.e., $\partial F_i/\partial w_{jk} = \delta_{ij} f_k(u_k(t - \tau_{ik}))$. In vector notation,

$$\frac{d\boldsymbol{p}_{\cdot w}(t)}{dt} = L(t)\boldsymbol{p}_{\cdot w}(t) + L(t - \tau)\boldsymbol{p}_{\cdot w}(t - \tau) + \frac{\partial \boldsymbol{F}}{\partial w} \tag{3.34}$$

where $L_{ij}(t) = \partial F_i/\partial u_j(t) = -\delta_{ij}/\tau_i$, $L_{ij}(t - \tau) = \partial F_i/\partial u_j(t - \tau_{ij})$, $\boldsymbol{p}_{\cdot w}(t - \tau)$ is the vector of components $p_{jw}(t - \tau_{\cdot j})$. The adjoint methods or backpropagation through time algorithms can easily be extended to this case. To see this, it suffices to remark that in the discretized and unfolded through time network, delays only introduce additional feedforward connections which may skip certain layers. Thus, the overall connectivity matrix of the unfolded network remains triangular, and therefore the usual backpropagation equations can be applied (see also Appendix).

When, in addition, delays are learnable, we must add, for each delay τ, a new sensitivity equation

$$\frac{dp_{i\tau}(t)}{dt} = -\frac{1}{\tau_i} p_{i\tau}(t) + \sum_j w_{ij} f_j'(u_j(t - \tau_{ij})) \cdot p_{j\tau}(t - \tau_{ij}) + \frac{\partial F_i}{\partial \tau} \tag{3.35}$$

where, as usual, $\partial F_i/\partial \tau$ denotes the explicit derivative, i.e., $\partial F_i/\partial \tau_{jk} = -\delta_{ij} w_{ik} f'(u_k(t - \tau_{ik}))\, du_k(t - \tau_{ik})/dt = -\delta_{ij} w_{ik} f'(u_k(t - \tau_{ik})) F_k(t - \tau_{ik})$. In vector notation,

$$\frac{d\boldsymbol{p}_{\cdot \tau}(t)}{dt} = L(t)\boldsymbol{p}_{\cdot \tau}(t) + L(t - \tau)\boldsymbol{p}_{\cdot \tau}(t - \tau) + \frac{\partial \boldsymbol{F}}{\partial \tau} \tag{3.36}$$

where $\boldsymbol{p}_{\cdot \tau}(t - \tau)$ is the vector of components $p_{j\tau}(t - \tau_{\cdot j})$. To evolve the dynamics forward over the interval $[t_0, t_1]$, the initial conditions on any $u_i(t)$ must be defined over the internal $[t_0 - \rho_i, t_0]$ where $\rho_i = \max_j \tau_{ji}$. If, as is often the case, these conditions are chosen independently of the relevant parameters and kept constant, then we can set $p_{iw}(t) = 0$ for $t_0 - \rho_i \le t \le t_0$. In any case, once these initial conditions are determined, the sensitivity equations (3.33) and possibly (3.35) can be evolved forward in time, as in (3.8), although this is computationally expensive.

As in the case of systems without delays, similar alternative approaches can be used to adjust the delays, including backpropagation through time or adjoint methods for delayed differential equations (see also Hale [15]). Intuitively, this is very clear using the same remark already made above. If we represent a general delayed system as in (1.9), then in the discretized and unfolded through time version of the network, forward connections with weight $w_{ij}(t - s)$, for all possible t and s, must be introduced between all pairs of layers. The weights $w_{ij}(t - s)$ can then be adjusted using the usual form of backpropagation. Formally, in this case, it is easy to see that the derivation (3.3)–(3.7), the adjoint system (3.15), and the update equation (3.18) remain valid provided $\boldsymbol{p}_{\cdot w}(t)$ is replaced by $\boldsymbol{p}_{\cdot w(t-s)}(t)$.

Especially if implemented on a computer, the algorithm described here is not meant to be used for undiscriminate learning of delays throughout large, amorphous, and densely interconnected networks with random initial conditions. Rather, it should be considered only for the fine tuning of a small set of parameters in architectures already endowed of a certain degree of structure. In this respect, it is remarkable that in some of the best studied examples of useful delays, both in natural (Carr and Konishi [8]) and artificial (Unnikrishnan *et al.* [32]) neural systems, the delays are arranged in orderly arrays of delay lines. These delays lines are essentially part of a feedforward network for which the learning task is much more simple. In these examples, different delays are used in order to bring together the effects of temporally separated events onto a coincidence detector type of neuron. For instance, in Unnikrishnan *et al.*, the successive parts of a spoken word are delayed differentially in order to arrive simultaneously onto a unit assigned to the recognition of that particular word. For a given input $I(t)$, the output the ith delay line is given by the convolution

$$o_i(t) = \int_0^{+\infty} K(i, t') I(t - t')\, dt' \tag{3.37}$$

where, here, $K(i, t')$ is the Gaussian kernel

$$K(i, t') = \frac{1}{\sqrt{2\pi}\sigma} e^{-(t' - iT)^2/2\sigma^2} \tag{3.38}$$

and T is a parameter used to discretize the possible delays. It is clear that in this approach based on fixed delays, delays could easily be made adjustable by allowing both the width σ and the center $t_i = iT$ of the delay kernel to vary. Both types of parameters could be adjusted by gradient descent since it is easy to derive $\partial o_i/\partial \sigma$ or $\partial o_i/\partial t_i$ from (3.37) and (3.38).

IV. Complexity and Limitations

A. Complexity

For completeness, the purpose here is to give rough known estimates of the computational and storage requirement of the previous algorithms. We also want to keep in consideration

whether the algorithm is causal or not, that is, whether it requires any integration backwards in time, for noncausal algorithms cannot be directly implemented in hardware. The complexity estimates one can derive depend on the type of machine on which the algorithms are implemented. We shall consider the case of a conventional computer implementation, but different estimates could result from a parallel implementation. This complexity analysis has several additional limitations. First of all, it assumes a fixed number of bits of precision for all the quantities involved, and no distinctions are made between the costs of different operations such as additions, multiplications, or input/output transfer functions f. Similarly, the costs of storing or reading from memory are treated as neglectable. These assumptions are easily violated in a physical implementation where different costs and durations may be attached to different operations, and where precision on the connection weights may often be limited to just a few bits. They are also, strictly speaking, insufficient from a theoretical standpoint since it is well known (see, for instance, Minsky and Papert [22] that for certain problems, the precision required on the parameters of a solution may scale with the dimension of the problem. However, the scaling of the precision with N may just behave, to a first approximation, as an amplification factor which increases the complexity order of the various algorithms, but without significantly affecting their ratios. Finally, these assumptions should be valid in the range of many simulations carried on current digital computers where high precision on all variables is easily achieved and where the number of neurons in the simulated networks is still often relatively small. It is also good to keep in mind that the complexity estimates to be derived correspond to rough asymptotic behavior. We shall assume that the number of adjustable parameters is n. In the usual case of densely interconnected networks, with first-order interactions only, $n = O(N^2)$. Complexity estimates based on the number of parameters rather than the number of neurons are more general since they extend to other models, such as higher order networks.

Fixed-Point Learning by Direct Integration: This corresponds to a direct calculation of the sensitivities and weight changes using (2.8) and (2.9). For each pattern presentation and each modification cycle, it requires the following.

1) Integration of the N equation (1.1) for the forward evolution of the system. Typically, in the first-order additive model or in the usual connectionist networks, the main calculation during each integration step is the multiplication of the weight matrix by the vector of activations. This gives a number of operations which is on the order of $O(n)$, and if there are M time steps in the forward integration, this yields on the order of $O(nM)$ operations. The factor M should not matter very much in normal conditions since the convergence to fixed points is usually fast [for instance, in the case of Hopfield associative memories, corresponding essentially to symmetrically interconnected additive models, and under the proper assumptions, the convergence to fixed points occurs in fewer than $O(\log \log N)$ steps (Komlós and Paturi [19])].

2) Inversion of the $N \times N$ matrix L in (2.8). The theoretical minimal cost of such an inversion on a serial machine is known to be of the form $O(N^{2+\alpha})$ where $0 < \alpha < 0.496$ (Pan [23]). The precise value of α is not known, but explicit algorithms are available which can compute the inverse in fewer than $O(N^3)$ computations.

3) Evaluation of the sensitivities $\boldsymbol{p}_{\cdot w}$ for each w according to (2.8). The matrix vector multiplications $L^{-1}\partial \boldsymbol{F}/\partial_w$ requires, in general, $O(N^2)$ operations, and there are n parameters so that the overall complexity scales like $O(nN^2)$. In general, however (unless there is weight sharing), the vector of explicit derivatives $\partial \boldsymbol{F}/\partial w$ is very sparse and has only one nonzero component for each w. So the overall complexity of this steps scales more like $O(nN)$.

4) Evaluation of the weights adjustments, according to (2.9). For each parameter w, the scalar product $(\partial E/\partial \boldsymbol{u}^f)^t \boldsymbol{p}_{\cdot w}$ requires $O(N)$ operations, so that the complexity of this step is $O(nN)$.

Thus, in typical cases, fixed-point learning by gradient descent using direct integration takes $O(nN)$ operations. As far as memory requirement is concerned, if one needs to store all the sensitivities together, then it is easy to see that this is the most demanding requirement and requires a storage capacity of $O(nN)$. If the implementation is such that the sensitivity to only one parameter at a time needs to be stored in memory, then one still needs to store the matrix L and the weights (in addition to evolving the dynamics forward), and it is easy to see that the storage requirement is on the order of $\max(O(n), O(N^2))$.

Fixed-Point Learning by Back-propagation: This corresponds to the calculation of the backpropagated errors $\boldsymbol{v}$ from the relaxation of (2.14) and the parameters updates from (2.11). This requires the following.

1) Same as 1) for the direct algorithm for fixed-point learning above.

2) Integration of the N equations (2.14) for the forward evolution of the backpropagated errors $\boldsymbol{v}$. This, in general, should require $O(N^2)$ operations at each time step for the vector/matrix multiplication. However, in typical cases, the matrix L contains roughly the same number of nonzero entries as W, so that the total cost for the calculation of the equilibrium value of $\boldsymbol{v}$ should scale like $O(nM')$, where M' is the number of time steps, which should not be too relevant in typical situations (the exact duration depends on the negative real parts of the eigenvalues of the matrix L, which are the time constants of the time decaying exponentials governing the relaxation of the linear system).

3) Evaluation of the weight adjustments according to (2.11). For each parameter w, the scalar product $(\partial \boldsymbol{F}/\partial w)^t \boldsymbol{v}$ normally requires $O(N)$ operations. Again, because of the sparsity in general of $\partial \boldsymbol{F}/\partial w$, the total cost of this step for all parameters is $O(n)$.

In the usual case of connectionist feedforward architectures with a small number of hidden layers, the convergence can be considered instantaneous and the factors M or M' entirely neglected [otherwise, one can introduce a factor of $\max(O(M), O(M'))$], so that the total number of operations is roughly equal to three times the number of weights (one time for each step above). It therefore scales like $O(n)$, with a saving of a factor of N over the direct integration. It is easy

to see that in usual conditions, the main memory requirement in the backpropagation algorithm comes from the storage of the weights, and thus also scales like $O(n)$.

Trajectory Learning by Direct Integration: This corresponds to a direct forward integration of the sensitivity equations (3.7) and using the result to estimate the parameter adjustments according to (3.3). For each pattern presentation and each modification cycle, it requires the following.

1) Integration of the N equations (1.1) for the forward evolution of the system. As in the similar cases above, with M time steps, the total complexity typically scales like $O(nM)$. Here, however, the system does not necessarily converge to a fixed point and M can be an important large factor.

2) Evolve the sensitivity system (3.7) forward in time using (3.8). In general, the matrix vector multiplication in (3.8) should take $O(N^2)$ operations, which should yield with M time steps and n parameters a total complexity of $O(nMN^2)$. With the same remark on the matrix L already made above, this complexity can be lowered to $O(n^2M)$. Of course, in the case of dense networks where $n = O(N^2)$, there is no difference between the two estimates.

3) Evaluation of the weight adjustments according to (3.9). For each parameter w, the scalar product $(\partial E/\partial \boldsymbol{u})^t \boldsymbol{p}_{.w}$ requires $O(N)$ operations. With M steps and n parameters, the total complexity scales like $O(nMN)$.

Thus, in the direct algorithm, all integrations are forward in time. The total cost of an update for one patterns is likely to scale like $O(n^2M)$, which is $O(MN^4)$ in dense networks. As far as memory requirement is concerned, for each adjustable parameter, one must store the corresponding sensitivity vector so that the minimal memory requirement is roughly $O(nN)$. It is important to remark that unlike the fixed-point case, it is not possible to work with only one $\boldsymbol{p}_{.w}$ at a time since all the sensitivities need to be available to compute their evolution from one time step to the next.

Trajectory Learning by Backpropagation : This corresponds to backpropagation through time or adjoint methods and relies on the backpropagation of the errors or, equivalently, the solution of the adjoint system (3.15) backwards in time. This requires the following.

1) Same as 1) for the direct algorithm for trajectory learning above.

2) Integration of the N equations of the adjoint system (3.15) backwards in time. Again, the vector matrix multiplication scales, in general, like $O(N^2)$, but this estimate can be reduced to $O(n)$ by using the fact that the number of nonzero entries in L is of the same order as the number of parameters in the most commonly used models. The complexity of solving the adjoint system backwards in time thus scales like $O(nM)$ [rather than $O(MN^2)$].

3) Evaluation of the weight adjustments according to (3.18). For each parameter w, the scalar product $(\partial \boldsymbol{F}/\partial w)^t \boldsymbol{v}$ normally requires $O(N)$ operations. Again, because of the sparsity, in general, of $\partial \boldsymbol{F}/\partial w$, the total cost of this step for all parameters is $O(nM)$ [rather than $O(nMN)$].

Thus, learning trajectories by backpropagation typically requires $O(nM)$ operations. This can also be seen immediately from the backpropagation through time approach since the unfolded network has nM forward connections, and the number of operations for the usual backpropagation is roughly equal to three times the number of weights (one for the forward pass, one for the backward pass, and one for the updates). Backpropagation leads to a reduction in computational costs by a factor of n over direct integration, but requires that the error or adjoint system be evolved backwards in time. As far as memory requirements are concerned, one must evolve the network dynamics forward and store the corresponding NM activities into memory before (3.15) can be evolved backwards in time. Since $O(n)$ parameters are in memory anyway, the memory requirement can be seen to be max $(O(NM), O(n))$. If one still wants to integrate (3.15) forward in time as described in (3.22)–(3.31), the price to pay is that of inverting an $N \times N$ matrix. As we have seen, this has a cost of the form $O(N^{2+\alpha})$ where $0 < \alpha < 0.496$. Thus, if M is larger than $O(N^\alpha)$, the matrix inversion is not relevant, at least asymptotically.

B. Limitations

In spite of its success, there are several problems with gradient descent learning. The problems are slightly different, depending on the perspective adopted: hardware design, biological realism, numerical algorithm for digital computers, or as a general approach to learning. First of all, most of the time it has been used as a supervised mode of learning where the correct target patterns need to be known. It requires a fairly precise calculation of the gradient which, in many cases, is expensive. In biological systems, evidence for neural circuitry dedicated to the computation of gradients for learning is lacking, and the existence of pathways for the precise backpropagation of error signals is very debatable, to say the least. More fundamentally, perhaps, it seems intuitively clear that gradient descent applied to large amorphous networks cannot be successful on complex problems (see also Geman *et al.* [13]). For instance, nobody has tried to take a fully interconnected random network and train it to play chess by gradient descent, although we can reasonably expect that there exists a relatively small circuit, with at most a few thousand threshold gates, which outperforms any existing program or player. Even in the case of a simpler game such as backgammon, which has been successfully tackled by gradient descent learning (Tesauro [29]), considerable additional structure had to be introduced in the system before the start of the learning procedure.

The exact reason for the limitations inherent to gradient descent learning applied to random starting networks is far from clear. Gradient descent can fail in several ways: by reaching a poor local minima, by reaching a global minima which is unsatisfactory, by encountering numerical precision problems and/or getting stuck in long plateaus, by becoming prohibitively slow for the given hardware, or by failing to generalize properly. These reasons are not independent or well understood. For instance, are local minima a characteristic of complex problems? Intuitively, it seems that for a given problem, as the number of units in a network is increased, the number of local minima in the LMS error should tend

to decrease. Thus, either complex problems are characterized by error landscapes plagued with local minima irrespective of network size or, more likely, local minima tend to vanish, but for infinite or prohibitively large networks. In fact, there is some partial evidence (Baldi [5]) that if we allow for a continuum of hidden units, the problem of local minima disappears, at least in the case of normalized inputs. If this is true, then one may expect, by approximation, that for every problem, there is a finite (sufficiently large network) which has an error function devoid of local minima. Alternatively, if gradient descent is considered to fail only because of speed, is it because the numerical precision required to exit from long plateaus or narrow saddle points becomes excessive or because the network becomes to large? As fundamental, if not more so, is the problem of the evolution of the generalization, i.e., of the performance of the network on points which are not part of the training set. Of course, these questions are intimately related to other difficult problems in complexity theory, such as the $P \neq NP$ problem. Is it the case that, for reasonable problems, gradient descent cannot find, in polynomial time, polynomial size networks which execute the task sufficiently well?

There is an additional set of difficulties with gradient descent learning of fixed points or trajectories, which is particular to *recurrent* networks, and which has to do with the bifurcations of the system being considered. In the case of a recurrent[3] network, as the parameters of 1) are varied, the system may or may not undergo a series of bifurcations, i.e., of abrupt changes in the structure of its trajectories and, in particular, of its attractors (fixed points, limit cycles, etc.). This, in turn, may translate into abrupt discontinuities, oscillations, or nonconvergence in the corresponding learning curve. At each bifurcation, the error function is usually discontinuous, and therefore the gradient is not properly defined. Learning can be disrupted in two ways: when unwanted abrupt changes occur in the trajectory space of the dynamical system, or when desirable bifurcations are prevented from occurring. A classical example of the second type is the case of a neural network with very small initial weights being trained to oscillate, in a symmetric and stable fashion, around the origin. With small initial weights, the network, in general, coverges to its unique fixed point at the origin, with a large error. If we slightly perturb the weights, remaining away from any bifurcation, the network continues to converge to its unique fixed point, which may be slightly displaced from the origin and yield an even greater error, so that learning by gradient descent becomes impossible (the starting configuration of weight is a local minimum of the error function).

V. Conclusion

Gradient descent learning is a simple but powerful principle for the global coordination of local synaptic changes in neural networks. In spite of its conceptual simplicity, it has led to the introduction of many different algorithms for different models and different problems. The general framework of dynamical systems enables one to give a unified treatment of all the known procedures, to reveal and stress the common underlying computational structures, and provides a general flexible tool which can rapidly be adapted to new situations. It clarifies the relations between different approaches, the similarities between the static and time-dependent cases, and easily yields new algorithms for new models, for instance, in the case of networks with adjustable delays. The analysis shows that to this date, essentially two distinct algorithms have been used to evaluate gradients: one is based on a direct solution of the sensitivity equations, and the other on the solution of the adjoint system equivalent to backpropagation. The second algorithm is computationally superior, but requires, in the time-dependent case, integrating a time-dependent linear system backwards in time. In spite of several fundamental limitations, gradient descent learning remains an important tool in the study of learning and the design of artificial intelligent machine. Understanding and overcoming these limitations while uncovering the secrets of learning in biological systems remains one of the main challenges in the field.

Appendix

A. Discrete Dynamical Systems and Backpropagation

Gradient descent learning has been studied in a framework of continuous-time dynamical systems. Equivalently, one could use discrete-time dynamical systems of the form

$$\boldsymbol{u}(m+1) = \boldsymbol{G}(\boldsymbol{u}(m), W, \boldsymbol{I}(m), \boldsymbol{u}^*(m)). \tag{A.1}$$

There is a one-to-one correspondence between systems satisfying (1.1) and (A.1). Namely, if we discretize (1.1) according to $d\boldsymbol{u}/dt \approx [\boldsymbol{u}(t+\Delta t) - \boldsymbol{u}(t)]/\Delta t = [\boldsymbol{u}(m+1) - \boldsymbol{u}(m)]/\Delta t$ and use $\Delta t = 1$, we get $\boldsymbol{u}(m+1) = \boldsymbol{u}(m) + \boldsymbol{F}(\boldsymbol{u}, W, \boldsymbol{I}, \boldsymbol{u}^*)$, so that the correspondence between (1.1) and (A.1) is established by having

$$\boldsymbol{G} = \boldsymbol{u} - \boldsymbol{F}. \tag{A.2}$$

It is important to notice that the correspondence between a continuous system and its discretized version is formal: the two can behave very differently, unless the unit of time for which $\Delta t = 1$ is small enough, so that discrete differences yield a good approximation to derivatives. Gradient descent learning in discrete systems can be expressed as

$$w(m+1) = w(m) + \eta \frac{\partial E}{\partial w}(m). \tag{A.3}$$

It is straightforward to check that all the steps carried in the derivation of $\partial E/\partial w$ for continuous systems remain valid in the case of a discrete system. (In the case of trajectory learning, one must, of course, replace $\boldsymbol{u}(t)$ by $\boldsymbol{u}(m)$ and the integrals by the corresponding sums.) Thus, the learning rule for a discrete system such as (A.2) can be derived by calculating $\partial E/\partial w$ in the corresponding continuous system defined by (1.1) and (A.2) and substituting in (A.3) or, equivalently, by directly discretizing the learning rule of the associated continuous system, with $\Delta t = 1$. Again, for the evolution of the parameters to parallel each other in the discrete and

[3] In a feedforward network where the transfer functions of the units are continuous, the output is a continuous function of the parameters, and therefore there are no bifurcations.

continuous systems, two conditions are necessary: both the activation and the parameter units of time corresponding to $\Delta t = 1$ must be sufficiently small to ensure, first, that both systems have similar activation dynamics, and then, similar parameter dynamics.

As an example, we can easily derive the usual (Rumelhart *et al.* [26]) backpropagation equations for feedforward discrete-time neural networks trained on fixed points. To begin with, let us give a description of backpropagation which is as general as possible. Thus, we assume that we have a discrete dynamical system which has a layered feedforward structure, and which is defined by

$$u_i^l(m+1) = f_i^l(u^{l-1}(m), \cdots, u^1(m), u^0), \tag{A.4}$$

$l = 1, \cdots, p$, where u_i^l can be viewed as the activity of the ith unit in the lth layer, which depends on the activities of the units in previous layers. More generally and throughout this section, upper indexes will always refer to layers. The input and output layers correspond to layers 0 and p, respectively. $\boldsymbol{u}^0$ can be viewed as the input which is held fixed so that the system converges in at most p steps to a vector $\boldsymbol{u}$ of stable activities. Only these stable activities are of importance, and therefore the temporal indexes in (A.4) can be neglected. The transfer functions f_i^l may depend on adjustable parameters w. In order to cover the case of "weight sharing," we assume that a given parameter w may appear in several different transfer functions. Thus, let $\mathcal{F}(w)$ be the set of indexes (i, l) such that w appears in the expression of f_i^l. If the performance of the system is measured by an error function E, then

$$\frac{\partial E}{\partial w} = \sum_{(i,l)\in\mathcal{F}(w)} \frac{\partial E}{\partial u_i^l}\frac{\partial u_i^l}{\partial w}. \tag{A.5}$$

Starting from the output layer p, the quantities $\partial E/\partial u_i^l$ can be calculated using the chain rule backwards through the network. Thus, backpropagation is given by the recurrence relations

$$\frac{\partial E}{\partial u_i^l} = \sum_{(j,k)k>l} \frac{\partial E}{\partial u_j^k}\frac{\partial u_j^k}{\partial u_i^l} = \sum_{(j,k)k>l} \frac{\partial E}{\partial u_j^k}\frac{\partial f_j^k}{\partial u_i^l}. \tag{A.6}$$

The quantities $\partial E/\partial u_i^p$ are, of course, given.

Connectionist neural networks are special cases of this framework, and can be viewed as discretized versions of the additive model (1.4) defined by

$$u_i^l(m+1) = f_i^l\left(\sum_{k<l}\sum_j w_{ij}^{lk} u_j^k(m)\right) \tag{A.7}$$

for $l = 1, \cdots, p$. The weight matrix W is a lower triangular block matrix. The $l-k$ block $(l > k)$ represents the forward connections from layer k to layer l. A constant bias or threshold can easily be incorporated inside the sum appearing on the right-hand side in the usual way by introducing a special unit with constant activity 1 connected to all the other units by a bias weight w_{i0}^{l0}. The usual LMS error function is given, for one pattern, by $E = \sum_i (u_i^p - u_i^*)^2/2$. From what we have seen, we can use (2.10)

$$\Delta w = \eta\left(\frac{\partial \boldsymbol{F}}{\partial w}\right)^t (L^t)^{-1}\frac{\partial E}{\partial \boldsymbol{u}^f} \tag{A.8}$$

to update the weights. Here, L is the Jacobian matrix of $I - G$. Thus, L is a lower triangular block matrix with

$$L = (l_{ij}^{lk}) = \begin{cases} -1 & \text{if } l = k \text{ and } i = j \\ (f_i^l)' w_{ij}^{lk} & \text{if } l > k \\ 0 & \text{otherwise.} \end{cases} \tag{A.9}$$

The matrix L^t is an upper triangular block matrix with

$$L^t = (m_{ij}^{lk}) = (l_{ji}^{kl}) = \begin{cases} -1 & \text{if } l = k \text{ and } i = j \\ (f_j^k)' w_{ji}^{kl} & \text{if } l < k \\ 0 & \text{otherwise.} \end{cases} \tag{A.10}$$

Thus, for $l > k$, we finally have

$$\Delta w_{ij}^{lk} = \eta\left(\frac{\partial F_i^l}{\partial w_{ij}^{lk}}\right) v_i^l = \eta (f_i^l)' u_j^k v_i^l \tag{A.11}$$

where the solution $\boldsymbol{v}$ of $L^t\boldsymbol{v} = \partial E/\partial \boldsymbol{u}$ can be computed recursively, layer by layer, starting from the output layer. Indeed, one has immediately $v_i^p = u_i^p - u_i^*$ for the output layer, and then, using the fact that L^t is upper triangular with $a - 1$ diagonal,

$$v_i^l = \sum_{k>l}\sum_j (f_j^k) w_{ji}^{kl} v_j^k \tag{A.12}$$

which is the usual formula for the backpropagation of errors.

As a second example, we can derive the algorithm described in Williams and Zipser [34]. This time, we consider a recurrent network of units satisfying

$$u_i(m+1) = f_i\left(\sum_j w_{ij} u_j(m)\right). \tag{A.13}$$

The usual LMS error function is written as

$$\boldsymbol{E} = \frac{1}{2}\sum_m\sum_i (u_i^*(m) - u_i(m))^2 = \sum_m\sum_i e_i(m) \tag{A.14}$$

with $e_i(m) = 0$ for the units i and the times m at which a target value is not defined. The corresponding continuous model is given by (1.4). It is easy to check that all the steps carried in the derivation of the learning algorithm for continuous systems remain valid in the case of discrete systems. Thus, by using (3.8) and (3.9) with $\Delta t = 1$ and $L_{ij}(m) = \partial F_i(m)/\partial u_j = -\delta_{ij} + f_i'(\sum_l w_{il} u_l(m)) w_{ij}$, we get

$$\frac{\partial \boldsymbol{E}}{\partial w} = -\sum_{m=1}^{M}\sum_i e_i(m) p_{iw}(m) \tag{A.15}$$

where the sensitivities $p_{iw}(m)$ satisfy the recurrence relation

$$p_{kw_{ij}}(m) = f'\left(\sum_l w_{kl} u_l(m)\right)\left[\sum_l w_{kl} p_{lw_{ij}}(m) + \delta_{ik} u_j(m)\right] \tag{A.16}$$

and, usually, the initial conditions $p_{iw}(0) = 0$ for any i and w. This is exactly the algorithm in Williams and Zipser [34].

As a third and last example, we can derive the explicit solution or adjoint algorithm of Section III from the discrete

backpropagation through time. Indeed, if we discretize a continuous system satisfying (1.1), we get

$$u_i(t + \Delta t) = u_i(t) + \Delta t F_i(u, W, I, u^*). \tag{A.17}$$

In backpropagation through time, the recurrent network and its time evolution are replaced by a layered feedforward network, with one layer for each interval Δt. Therefore, using the same notation as in (A.4), we can write at equilibrium

$$u_i^{m+1} = u_i^m + \Delta t F_i(u, W, I, u^*) = f_i^{m+1}(u, W, I, u^*). \tag{A.18}$$

Here, each layer contains all the units of the original network, and there are as many layers as time steps. For simplicity we can assume that there are no delays, and therefore all connections in the unfolded network run from one layer to the next. The total error $\boldsymbol{E}$ is the sum of all the instantaneous errors $\boldsymbol{E} = \sum_1^M E(l)\Delta t = \sum E^l \Delta t$. Each instantaneous error $E^l \Delta t$ occurs at the lth level of the unfolded network, and therefore gives rise to a correction signal of the form $\Delta t\, \partial E^l / \partial u^m$ which can be backpropagated in all the previous layers with index $m < l$ as in (A.5) and (A.6). Thus, we have

$$\frac{\partial \boldsymbol{E}}{\partial w} = \sum_{m=1}^{M}\sum_{i=1}^{N}\sum_{l \geq m} \frac{\partial E^l}{\partial u_i^m}\frac{\partial u_i^m}{\partial w}\Delta t = \sum_{m=1}^{M}\sum_{i=1}^{N} z_i(m)\frac{\partial u_i^m}{\partial w} \tag{A.19}$$

where

$$z_i(m) = \sum_{l \geq m}^{M} \frac{\partial E^l}{\partial u_i^m}\Delta t \tag{A.20}$$

is the sum of all the error signals arriving on layer m from all the other upper layers and from layer m itself. Using (A.6),

$$z_i(m) = \frac{\partial E^m}{\partial u_i^m}\Delta t + \Delta t \sum_{l=m+1}^{M}\sum_{j=1}^{N} \frac{\partial E^l}{\partial u_j^{m+1}}\frac{\partial f_j^{m+1}}{\partial u_i^m}. \tag{A.21}$$

Using the definition of f_i^{m+1} given in (A.18), we get

$$\frac{z_i(m+1) - z_i(m)}{\Delta t} = -\Delta t \sum_{l=m+1}^{M}\sum_{j+1}^{N} \frac{\partial E^l}{\partial u_j^{m+1}}\frac{\partial F}{\partial u_i^m} - \frac{\partial E^m}{\partial u_i^m} \tag{A.22}$$

or, in continuous form,

$$\frac{dz}{dt} = -L^t z - \frac{\partial \boldsymbol{E}}{\partial \boldsymbol{u}} \tag{A.23}$$

which, for instance, is exactly the same as (3.15). Accordingly, updating the parameters using the continuous version of (A.19) immediately yields (3.18). From (A.20), the boundary condition on z is $z(t_1) = z(M) = 0$, identical to the condition on v in the adjoint methods algorithm.

B. Solution of Linear Systems of Differential Equations and Adoint Systems

The following is a classical theorem which gives in explicit form the solution of a general linear system of differential equations (see, for instance, Apostol [2]).

Theorem : Given an $n \times n$ matrix $A(t)$ and an n-dimensional vector function $\boldsymbol{v}(t)$, both continuous on an open interval containing t_0, the solution of the initial value problem

$$\frac{d\boldsymbol{y}}{dt} = A(t)\boldsymbol{y} + \boldsymbol{b}(t) \qquad \boldsymbol{y}(t_0) = \boldsymbol{c} \tag{B.1}$$

exists and is unique and given by

$$\boldsymbol{y}(t) = H(t)H^{-1}(t_0)\boldsymbol{c} + H(t)\int_{t_0}^{t} H^{-1}(s)\boldsymbol{b}(s)\, ds \tag{B.2}$$

where $H(t)$ is an $n \times n$ invertible matrix whose columns form n independent solutions of the homogeneous systems $d\boldsymbol{y}/dt = A(t)\boldsymbol{y}$ associated with (B.1). Alternatively,

$$\boldsymbol{y}(t) = K(t, t_0)\boldsymbol{c} + \int_{t_0}^{t} K(t, s)\boldsymbol{b}(s)\, ds \tag{B.3}$$

where $K(t, s) = H(t)H^{-1}(s)$ is the transition matrix of the system. Obviously, for any t, $K(t, t) = I$ and K also satisfies the differential equation

$$\frac{d}{dt}K(t, s) = A(t)K(t, s). \tag{B.4}$$

For instance, in the one-dimensional case,

$$H(t) = e^{\int_{t_0}^{t} A(s)\, ds}, \qquad K(t, s) = e^{\int_s^t A(r)\, dr} \tag{B.5}$$

and in the n-dimensional case, *with constant* $A(t) = A$,

$$H(t) = e^{tA}, \qquad K(t, s) = e^{(t-s)A}. \tag{B.6}$$

Notice that it is easy from (B.2) to express the value of $\boldsymbol{y}(t_0)$ as a function of $\boldsymbol{y}(t)$. In other words, if (B.1) is to be solved backwards in time from a final condition $\boldsymbol{y}(t_1)$, then

$$\boldsymbol{y}(t) = K(t, t_1)\boldsymbol{y}(t_1) + \int_{t_1}^{t} K(t, s)\boldsymbol{b}(s)\, ds. \tag{B.7}$$

Thus, the same equation (B.2) describes the evolution of the system forward or backwards in time and $K(t, s)^{-1} = K(s, t)$. By definition, the adjoint system associated with a homogeneous system governed by a matrix A is the homogeneous system governed by the opposite of the adjoint of A. Thus, in the case of real variables, the adjoint of $d\boldsymbol{y}/dt = A(t)\boldsymbol{y}$ is $d\boldsymbol{v}/dt = -A^t(t)\boldsymbol{v}$. Clearly, $d(\boldsymbol{y}^t\boldsymbol{v})/dt = \boldsymbol{y}^t\, d\boldsymbol{v}/dt + \boldsymbol{v}^t\, d\boldsymbol{y}/dt = 0$. Thus, for given initial conditions, the solutions $\boldsymbol{y}(t)$ and $\boldsymbol{v}(t)$ of the two systems maintain a constant angle. Let $L(t, s) = J(t)J^{-1}(s)$ be the transition matrix of the adjoint system. Then $\boldsymbol{y}(t) = H(t)H^{-1}(s)\boldsymbol{y}(t_0)$ and $\boldsymbol{v}(t) = J(t)J^{-1}(s)\boldsymbol{v}(t_0)$. By using the constancy of the angle, we see that the transition matrix of the adjoint system is given by $L(t, s) = K(s, t)^t$ (i.e., $J^{-1} = H^t$).

C. Variational Methods

Alternatively, to derive the learning algorithms, one can use a variational approach (see Troutman [31] for more mathematical background). To minimize the error (3.1) under

the constraint that $\boldsymbol{u}$ be a trajectory of (1.1), we can extremize the functional

$$\mathcal{L} = \int_{t_0}^{t_1} \left[E(\boldsymbol{u}, \boldsymbol{u}^*, t) + \sum_i v_i \left(F_i(\boldsymbol{u}, W, \boldsymbol{I}, \boldsymbol{u}^*) - \frac{du_i}{dt} \right) \right] dt = \int_{t_0}^{t_1} \mathcal{E}\, dt \tag{C.1}$$

which incorporates the constraints imposed by (1.1) using the Lagrange multipliers $\boldsymbol{v}$. Taking a variation of (C.1), i.e., by looking at how $\mathcal{L}$ varies, to a first order, with a small change in $\boldsymbol{u}$ or W, one first obtains the Euler equation:

$$\frac{\partial \mathcal{E}}{\partial \boldsymbol{u}} - \frac{d}{dt} \frac{\partial \mathcal{E}}{\partial d\boldsymbol{u}/dt} = 0. \tag{C.2}$$

This immediately gives

$$\frac{dv_i}{dt} = -\sum_j \frac{\partial F_j}{\partial u_i} v_j - \frac{\partial E}{\partial u_i} \tag{C.3}$$

which is exactly (3.15), with the Lagrange multipliers being the backpropagated error. The boundary condition resulting from the final time t_1 being fixed is given by $\partial \mathcal{E}/\partial(d\boldsymbol{u}/dt)(t_1) = 0$, which yields $\boldsymbol{v}(t_1) = 0$. Finally the variation $\delta\mathcal{L}$ is related to the variation δW by $\delta\mathcal{L} = [\int_{t_0}^{t_1} \boldsymbol{v}^t (\partial \boldsymbol{F}/\partial \boldsymbol{w})\, dt] \delta W$, which gives (3.18).

ACKNOWLEDGMENT

I would like to thank G. Hinton, B. Pearlmutter, F. Pineda, and the reviewers for their useful comments.

REFERENCES

[1] S. Amari, "Characteristics of random nets of analog neuron-like elements," *IEEE Trans. Syst., Man, Cybern.*, vol. SMC-2, no. 5, pp. 643–657, 1972.
[2] T. Apostol, *Calculus*, 2nd ed. New York: Wiley, 1967.
[3] P. Baldi, and R. Meir, "Computing with arrays of coupled oscillators: An application to preattentive texture discrimination," *Neural Computation*, vol. 2, pp. 458–471, 1990.
[4] P. Baldi, and F. Pineda, "Contrastive learning and neural oscillations," *Neural Computation*, vol. 3, no. 4, pp. 526–545, 1991.
[5] P. Baldi, "Computing with arrays of bell-shaped and sigmoid functions," in *Neural Information Processing Systems*, vol. 3, Los Altos, CA: Morgan Kaufmann, 1991.
[6] P. Baldi, and A. Atiya, "How delays affect neural dynamics and learning," *IEEE Trans. Neural Networks*,vol. 5, no. 4, pp. 426–635, 1994.
[7] A. E. Bryson, and W. Denham, "A steepest-ascent method for solving optimum programming problems," *J. Appl. Mech.*, vol. 29, no. 2, pp. 247–257, 1962.
[8] C. E. Carr, and M. Konishi, "Axonal delay lines for time measurement in the owl's brainstem," *PNAS USA*, vol. 85, pp. 8311–8315, 1988.
[9] B. de Vries, and J. C. Principe, "A theory for neural networks with time delays," in *Advances in Neural Information Processing Systems*, vol. 3, R. P. Lippmann, J. E. Moddy, and D. S. Touretzky, Eds. Los Altos, CA: Morgan Kaufmann, 1991, pp. 162–168.
[10] Y. Fang, and T. Sejnowski, "Faster learning for dynamic recurrent back-propagation," *Neural Computation*, vol. 2, pp. 270–273, 1990.
[11] O. Farotimi, A. Dembo, and T. Kailath, "A general weight matrix formulation using optimal control," *IEEE Trans. Neural Networks*, vol. 2, no. 3, pp. 378–394, 1991.
[12] R. Fitzhugh, "Impulses and physiological states in theoretical models of nerve membrane," *Biophys. J.*, vol. 1, pp. 445–466, 1961.
[13] S. Geman, E. Bienenstock, and R. Doursat, "Neural networks and the bias/variance dilemma," *Neural Computation*, vol. 4, pp. 1–58, 1992.
[14] S. Grossberg, and M. Cohen, "Absolute stability of global pattern formation and parallel memory storage by competitive neural networks," *IEEE Trans. Syst., Man, Cybern.*, vol. SMC-13, pp. 815–826, 1983.
[15] J. Hale, *Theory of Functional Differential Equations*. New York: Springer-Verlag, 1977.
[16] D. O. Hebb, *The Organization of Behavior*. New York: Wiley, 1949.
[17] J. L. Hindmarsh, and R. M. Rose, "A model of neuronal bursting using three coupled first order differential equations," *Proc. Roy. Soc. London*, vol. B 221, pp. 87–102, 1984.
[18] J. J. Hopfield, "Neurons with graded response have collective computational properties like those of two-state neurons," *PNAS USA*, vol. 81, pp. 3088–3092, 1984.
[19] J. Komlós, and R. Paturi, "Convergence results in an associative memory model," *Neural Networks*, vol. 1, no. 3, pp. 239–250, 1988.
[20] R. Linsker, "Self-organization in a perceptual network," *Computer*, vol. 21, pp. 105–117, 1988.
[21] S. G. Lisberger, and T. J. Sejnowski, "Computational analysis predicts sites of motor learning in the vestibulo-ocular reflex," Tech. Rep. 9201, UCSD Institute for Neural Computation, 1992.
[22] M. Minsky, and S. Papert, *Perceptrons*. Cambridge, MA: M.I.T. Press, 1969.
[23] V. Pan, "How can we speed up matrix multiplication?" *SIAM Rev.*, vol. 26, pp. 393–415, 1984.
[24] B. Pearlmutter, "Learning state space trajectories in recurrent neural networks," *Neural Computation*, vol. 1, 2, 263–269.
[25] F. J. Pineda, "Generalization of back-propagation to recurrent neural networks," *Phys. Rev. Lett.*, vol. 59, no. 19, pp. 2229–2232, 1987.
[26] D. E. Rumelhart, G. E. Hinton, and R. J. Williams, "Learning internal representations by error propagation," in *Parallel Distributed Processing*, D. E. Rumelhart and J. L. McClelland, Eds. Cambridge, MA: M.I.T. Press, 1986.
[27] T. J. Sejnowski, "On the stochastic dynamics of neuronal interactions," *Biol. Cybern.*, vol. 22, pp. 203–211, 1976.
[28] P. Y. Simard, J. P. Raysz, and B. Victorri, "Shaping the state space landscape in recurrent networks," in *Advances in Neural Information Processing Systems*, vol. 3, R. P. Lippmann, J. E. Moody, and D. S. Touretzky, Eds. Los Altos, CA: Morgan Kaufmann, 1991, pp. 105–112.
[29] G. Tesauro, "Neurogammon wins computer Olympiad," *Neural Computation*, vol. 1, pp. 321–323, 1989.
[30] N. Toomarian, and J. Barhen, "Adjoint-functions and temporal learning algorithms in neural networks," in *Advances in Neural Information Processing Systems, Vol. 3*, R. P. Lippman, J. E. Moody, and D. S. Touretsky, Eds. Los Altos, CA: Morgan Kaufmann, 1991.
[31] J. L. Troutman, *Variational Calculus with Elementary Convexity*. New York: Springer-Verlay, 1983.
[32] K. P. Unnikrishnan, J. J. Hopfield, and D. W. Tank, "Connected-digit speaker dependent speech recognition using a neural network with time-delayed connections," *IEEE Trans. Signal Processing*, vol. 39, no. 3, pp. 698–713, 1991.
[33] P. Werbos, "Beyond regression: New tools for prediction and analysis in the behavioral sciences," Ph.D. dissertation, Harvard Univ., Cambridge, MA, 1974.
[34] R. J. Williams, and D. Zipser, "A learning algorithm for continually running fully recurrent neural networks," *Neural Computation*, vol. 1, no. 2, pp. 270–280, 1989.

Pierre Baldi was born in Rome, Italy. He received the B.S. and M.S. degrees from the University of Paris and the Ph.D. degree from Caltech in 1986.

After spending two years as a Lecturer at the University of California, San Diego, he returned to Caltech in 1988 in the Jet Propulsion Laboratory and the Division of Biology. His main research interests are in the study of computations in both natural and artificial neural systems. He is also one of the founders of Net-ID, Inc., a neural network start-up.