Chapter 11

TRAINING ALGORITHMS FOR RECURRENT NEURAL NETS THAT ELIMINATE THE NEED FOR COMPUTATION OF ERROR GRADIENTS WITH APPLICATION TO TRAJECTORY PRODUCTION PROBLEM

Malur K. Sundareshan, Yee Chin Wong and Thomas Condarcure

Department of Electrical and Computer Engineering
University of Arizona, Tucson, AZ 85721-0104

I. INTRODUCTION

The most fundamental characteristic that enables a neural network to serve as a useful computational device is its learning capability. The implementation of an appropriately tailored learning algorithm, *i.e.*, a rule for adaptive adjustment of the network parameters such as the interconnection weights and gains of nonlinear characteristics, can endow the network with the capability for evolving into a structure that performs a desired computation. Designing a computationally efficient and yet simply implemented learning algorithm is hence at the core of successful neural network implementations for practical problems. Although interest in general learning theory and development of systematic training schemes has enjoyed a resurgence in recent times in the context of neural networks applications, they have a much longer history, tracing their origins to machine learning [Nilsson, 1965] and to adaptive learning control systems [Mendel, 1970].

When one narrows the discussion down to the specific context of neural network training, there are two general guiding principles on which many popular algorithms are based. These are Hebbian learning and gradient-descent learning. While Hebbian learning derived its following from the parallels that exist in biological systems, gradient-descent methods have attained a greater importance more recently in spite of the lack of conclusive evidence of whether biological systems employ such a mechanism for global learning of complex behaviors. The reason why gradient-descent methods have become popular is the optimization framework they facilitate not only to tailor specific training algorithms but also to provide estimates of the convergence behavior under these algorithms. A specific approach that has attained a considerable degree of popularity in recent times is the backpropagation rule [Rumelhart, 1986], which employs a gradient descent scheme to adjust the interconnection weights of a

© 2001 by CRC Press LLC

multilayer neural net in order to minimize a measure of the deviation between the actual network output and a reference entity. Alternate ways of specifying this measure, or the "error norm," can be used to develop different algorithms that perform supervised training.

Gradient descent-learning is conceptually very simple. However, in practical implementations it may lead to several problems related to the need for precise computation of gradients of the error function with respect to the network parameters being adjusted for the algorithm to succeed, and the possibility of being trapped at local minima of the error function that prevents the training error to be minimized to its global minimum value. The problems are further exacerbated when recurrent neural networks are attempted to be trained by this approach due to the complexity of implementing the needed updating equations.

Neural networks with recurrent connections and dynamical processing elements are finding increasing applications in diverse areas. While feedforward networks have been recognized to perform excellent pattern recognition even with very complex nonlinear decision surfaces, they are limited to processing stationary patterns (i.e., patterns that are invariant with time). It requires the power of dynamical networks, such as networks with recurrent and feedback connections, to handle the challenges posed in the storage of spatiotemporal patterns and sequences.

The recognition of the importance of training recurrent neural networks has prompted a host of researchers to investigate devising schemes by which gradient methods, and in particular backpropagation learning, could be extended to these networks. Several notable schemes have been developed with some early contributions made by numerous researchers. The backpropagation-through-time approach of Werbos [Werbos, 1990] attempts to approximate the time evolution of a recurrent net in terms of a sequence of static networks to which gradient methods are applied. Lapedes and Farber [Lapedes, 1986] propose a master slave formulation where deployment of a second neural network (master net) is made to perform the required computations in programming the attractors of the original dynamical network (slave net) to be trained. Similarly, Pineda [Pineda, 1987] and Almeida [Almeida, 1987] propose a second neural network, of the same dimension as the original one, for implementing the backward propagation equation in order to avoid a more complex matrix inversion in the weight adjustment process. A direct differentiation of the neural activation dynamics to calculate the error gradients is proposed by Williams and Zipser [Williams, 1989], which, although it provides some benefits of reducing the storage capacity needed, is still computationally very cumbersome and scales poorly to large networks (*i.e.*,networks with large numbers of dynamical processing elements and a large set of adjustable parameters). The algorithm proposed by Sato [Sato, 1990] is based on Lagrange multipliers, while Pearlmutter [Pearlmutter, 1989] gives a variational method that involves solving a set of "adjoint equations". A detailed survey of the various attempts to extend backpropagation learning to recurrent networks is also given by Pearlmutter [Pearlmutter, 1995].

© 2001 by CRC Press LLC

A major problem with the backpropagation approach used for recurrent network training is the computational intensity. For illustration, in the specific formulation given by Pearlmutter [Pearlmutter, 1989] that utilizes variational arguments, the complexity arises in the form of the need to solve a set of differential equations backwards in time and the need to store variables for recall later when the forward solution is implemented. Although this is not a drawback unique to backpropagation methods and is shared by many optimal control methods (such as dynamic programming [Bertsekas, 1987]), it certainly limits the attractiveness of the training scheme. Also limiting the usefulness for practical implementations is the fact that such gradient-based approaches do not scale well for large-sized networks. For a typical trajectory learning problem that involves training a continuous trajectory, defined over a time interval divided into L time steps, to a network with N neurons, some estimates [Toomarian, 1992] indicate that the total number of multiplications and additions required for the implementation of the required updating scales as $O(N^4 L)$. This clearly imposes a significant computational burden and is practically infeasible even for medium-sized networks. For overcoming the computational demands and ensuring a relatively manageable implementation, one is usually forced to making simplifying approximations, such as coarser gradient evaluations and heuristic selections of high gains in the activation functions (instead of allowing the network to find the optimized parameter values) [Sudharsanan, 1991, Sudharsanan, 1994], which in turn lead to reduced training efficiency. In several precision applications, as for instance those encountered in multijointed robot control [Karakasoglu, 1993] and reliable tracking of target maneuvers in severe clutter and noise environments [Wong, 1998], for which neural network-based solutions are becoming very attractive, making such approximations could pose serious limitations and alternate training procedures that bypass the need for computation of gradients of the error function are clearly useful.

The primary focus in this chapter is the design of supervised training schemes for recurrent neural networks that do not require gradient evaluations. In particular, we describe two distinct approaches, one that employs concepts from the theory of learning automata and the other based on the classical simplex optimization approach. Besides the elimination of the need for evaluation of error gradients, these approaches result in simple training algorithms suitable for implementation on low-end platforms such as personal computers. They also offer the flexibility of tailoring a number of specific training schemes based on the selection of linear and nonlinear reinforcement rules for updating automaton action probabilities and specification of different error norms. For demonstrating the training efficiency with these approaches, the illustrative task of spatiotemporal signal production by a trained neural network will be considered. To underscore the complexity involved in this task compared to learning of isolated fixed points, one may note that while a variety of networks, both static and dynamic, can be used for the fixed point learning problem even on arbitrarily high dimensional spaces, the trajectory learning problem requires exploiting the unique capability of recurrent neural networks for approximating

© 2001 by CRC Press LLC

the temporal dynamics. The practical usefulness of this problem can also be appreciated by noting that the ability of a recurrent neural net to be trained to produce desired trajectories and to converge to attractor trajectories from arbitrary starting points can be used effectively in several control applications, particularly where precise repetitive actions are desired to be performed, such as those arising in process control and robotic manipulator control.

The structure of the chapter is as follows. In Section 2, we shall provide a mathematical description of the learning problem in general dynamical systems and specialize this to spatiotemporal training of recurrent neural networks. Some important concepts such as incremental training and teacher forcing that contribute to the efficiency of training are also discussed. In Section 3, some basics on learning automata will be introduced and specific training policies that can be developed utilizing a penalty-reward structure for reinforcement learning will be discussed. Performance of these methods in training a recurrent neural network to produce prespecified periodic trajectory patterns is also established. The use of a nonlinear simplex optimization approach for neural network training will be discussed in Section 4. Some basics on simplex optimization are briefly introduced and a systematic training scheme for recurrent networks is developed. For comparison with the earlier approach, the trajectory production performance resulting from this approach is also established by considering specific benchmark trajectory patterns.

II. DESCRIPTION OF THE LEARNING PROBLEM AND SOME ISSUES IN SPATIOTEMPORAL TRAINING

A. GENERAL FRAMEWORK AND TRAINING GOALS

For a precise description of the learning problem and the training objectives considered in this article, it is useful to adopt the general framework afforded by considering the problem of modifying the behavior of a general nonlinear dynamical system to meet specified objectives. Consider the problem of training an N-dimensional system whose dynamics are described by the nonlinear differential equation

$$\dot{x}(t) = \Im(x, u, \wp) \tag{1}$$

where $x(.): \Re \to \Re^N$ is the N-dimensional vector that describes the evolution of the system state, $u(.): \Re \to \Re^m$ is a vector of external inputs (fixed or time-varying), $\wp \in \Re^M$ is a set of adjustable parameters and $\Im$ is a nonlinear function whose properties can be specified to include different types of dynamical behavior of interest. For instance, one may require $\Im$ to satisfy Lipschitz conditions in all of its arguments to ensure continuity of system trajectories, or to meet appropriate limiting conditions such as saturation limits and limits on the rise time of the trajectories in order to ensure boundedness and

© 2001 by CRC Press LLC

stability properties [Sudharsanan, 1991a]. The problem of interest is to develop an organized procedure for adjusting the parameters in the set $\wp$ such that the dynamical system exhibits desired time-behavior when started at an initial state $x(t_0) = x_0$. The system behavior desired may be specified in different ways depending on the particular application to which the system may be employed, such as: (i) requiring the system to exhibit an "asymptotically stable behavior", *i.e.,* $\|x(t)\|$ bounded for all time $t \geq t_0$ and $\lim_{t \to \infty} x(t) = x_e$, where x_e is a specified equilibrium state of system (1), or (ii) requiring the system to exhibit an acceptable "tracking behavior", *i.e.,* $\|x(t) - x^*(t)\| \leq \varepsilon$ for all $t \geq t_0$, and a specified $\varepsilon > 0$ and a trajectory to be tracked $x^*(t)$.

The specific problem cited above of training the network to ensure stability of the equilibrium points is of importance for fixed point learning, and a variety of applications such as associative memory designs and synthesis of nonlinear input-output mappers can be based on this property. For illustration, in the case of a network which is designed to serve as a reliable associative memory, the information stored corresponds to its stable equilibria. It has been established that by a careful selection of the nonlinear activation functions and of the interconnection weights, the network can be endowed with a number of stable equilibria, each of which corresponds to a to-be-stored memory vector. Furthermore, the size of the basins of attraction for each of these stable equilibria can be tailored in order to ensure desired levels of reliability in the memory recall process. As shown in Sudharsanan [1994], there exists intricate interrelations between the stability properties of the network equilibria and the convergence properties of the training algorithms that can be synthesized for these networks. In particular, one can attempt to utilize analytical stability results for these networks [Sudharsanan, 1991a, Sudharsanan, 1991b] in order to pre-select the shapes of the nonlinear activation functions (selection of the dc gain, for instance), which in turn enables one to develop learning rules that approximate gradient schemes but offer simple implementation possibilities. It must however be appreciated that the *a priori* selection of the nonlinear gains almost always leads to a suboptimal solution to the overall training of the recurrent neural network.

The second problem cited above of training the network to track a specified trajectory $x^*(t)$ is a more complex one. It is well known in the literature on nonlinear dynamical systems [Khalil, 1992] that under certain conditions the tracking problem can be reduced through an appropriate transformation to a corresponding problem of ensuring the stability of an equilibrium point of a transformed system. In particular, by defining the vector $y(t)$ as

$$y(t) = x(t) - x^*(t) \tag{2}$$

© 2001 by CRC Press LLC

one can transform the nonlinear system described by (1) into an equivalent system

$$\dot{y}(t) = g(y, u) \tag{3}$$

such that the tracking problem of forcing $x(t)$ to follow $x^*(t)$ in system (1) can be reduced to the problem of ensuring the stability of the equilibrium point $y(t) = 0$ in system (2). However, when the desired objective is one of training system (1) to perform a desired task, *i.e.,* explicitly adjust the parameters in the set $\wp$, such a reformulation of the problem may not be very useful in practice since the transformation given by (2) makes an explicit handling of these parameters almost always impossible. Consequently, any attempts at simplifying training by approximations such as those discussed above for the fixed point learning problem are more difficult to obtain in this case.

It is evident from the above discussion that training a dynamical system to produce state-space trajectories of specified forms constitutes a highly challenging learning problem due to the diversity in the possible spatiotemporal features that may need to be learned. A problem of particular interest is to train the system to exhibit desired limit cycles, which focuses only on the asymptotic behavior of the state-space trajectory to converge to a prespecified periodic temporal behavior. In the context of neural network training, an aspect of particular significance is ensuring the learning of the true spatiotemporal features as opposed to a point-by-point memorization of the terminal trajectory. This capability is provided by training the network to have the desired attractor dynamics such that arbitrary starting motions are forced to converge to the desired terminal periodic behavior. The complexity of implementing gradient-based training methods for these problems makes the development of alternate learning schemes that do not require the evaluation of gradients particularly attractive.

B. RECURRENT NEURAL NETWORK ARCHITECTURES

The training problems described in the previous section are quite general. For the establishment of specific simple rules for parameter adjustment and also to illustrate how well the training objectives are met in practice by different algorithms, it is useful to consider specialized architectures for the nonlinear dynamical system that is being trained. One such model that has been popular with neural network researchers is the continuous-time recurrent network model described by the set of coupled nonlinear differential equations

$$\frac{dv_i}{dt} + \tau_i v_i(t) = \tau_i \tanh\left(g_i \sum_{j=1}^{N} \omega_{ij} v_i(t) \right), \quad i = 1, 2, ..., N \tag{4}$$

© 2001 by CRC Press LLC

where $v_i(\cdot): \Re \rightarrow \Re$ denotes the state of the *i*th neuron, $\tau_i \in \Re$ is a time constant referred to as the relaxation time, $g_i \in \Re$ is a parameter that controls the slope of the sigmoidal activation function, and $\omega_{ij} \in \Re$ denotes the interconnection weight from the *j*th neuron to the *i*th neuron. The inputs to this network come from the initial conditions $v_i(t_0)$ and the outputs are the observations of the behavior of the state trajectories $v_i(t)$, for $t \geq t_0$. The task of training this network to serve as a useful computational device involves the implementation of an algorithm for progressively updating the $N^2 + 2N$ parameters $\{\omega_{ij}, \tau_i \text{ and } g_i\}$ such that when the training is completed, the network trajectories $v_i(t)$ starting from any initial states $v_i(t_0)$ behave in a prescribed manner to perform the desired computation.

In several practical problems, observing only a subset of the state variables may be of particular importance for checking whether the goals of the desired computation are met, and consequently designation of a set ϑ of output neurons (which is a subset of the total set of neurons) may be appropriate. Also, in certain problems where the input-output mapping behavior of the neural network is of interest, the use of externally applied time-dependent forcing signals to alter the activation dynamics of one or more neurons may be necessary. In order to be able to handle such problems, the dynamical framework can be expanded to permit the introduction of external inputs $I_i(t), i = 1, 2, ..., m$, by modifying the dynamical equation (4) into

$$\frac{dv_i}{dt} + \tau_i v_i(t) = \tau_i \tanh\left[g_i\left(\sum_{j=1}^{n} \omega_{ij} v_j(t) + \sum_{j=1}^{m} \tilde{\omega}_{ij} I_j(t) \right)\right], \quad i = 1, 2, ..., N \tag{5}$$

The weights $\tilde{\omega}_{ij}$, some of which could be zero, serve to fan-out the *m* input signals I_i into the individual nodes of the network.[1] The number of weight parameters that need to be trained increases in this case to $N^2 + (m+2)N$.

Evidently, for $I_j(t) = 0$, (5) reduces to (4). In this chapter, we will exclusively consider the specialized network architecture described by the dynamical equation (4), since for trajectory learning problems that will be considered for illustration here no external inputs are needed. A schematic of the general recurrent network architecture described by (5) is shown in Fig. 1.

[1] To conform this architecture to the more familiar multilayer configurations, the input signals I_i can be considered as the input nodes of the network. These nodes, however, are different from the *N* dynamical nodes in that they do not have recurrent or feedback connections, but connect to the N dynamical nodes only in the feedforward direction through the weights $\tilde{\omega}_{ij}$.

© 2001 by CRC Press LLC

C. SOME ISSUES OF INTEREST IN NEURAL NETWORK TRAINING

1. AN OPTIMIZATION FRAMEWORK FOR SPATIOTEMPORAL LEARNING

As noted earlier, a particularly challenging learning problem is that of training a recurrent network to produce a continuous trajectory of a specified form or to ultimately relax to a desired limit cycle behavior. In fact, this is also one of the tasks where the greater capabilities of dynamical networks are brought into a sharp focus. Recurrent network training to learn such trajectories has received some attention in the recent past with the investigation of schemes which use various forms of gradient descent algorithms. These include the real-time recurrent learning (RTRL) scheme of Williams and Zipser [Williams, 1989], the method of directed derivatives of Pearlmutter [Pearlmutter, 1989], and the method of adjoint operators of Toomarian and Barhen [Toomarian, 1992]. These works have shown that a dynamical network can indeed be trained to exhibit desired limit cycle behavior (it may be noted that this behavior is not possible to emulate in a static feedforward network) and have demonstrated the success of their training algorithms by application to the problem of learning certain benchmark trajectories. Some additional refinements to the use of gradient methods for training to produce continuous trajectories have also been made very recently by Lin *et. al.* [Lin, 1995] and by Ruiz *et. al.* [Ruiz, 1998]. While the closeness with which the desired trajectory could be generated varies from one algorithm to another, the required computation of gradients and other implementation considerations for error backpropagation impose considerable burden (in fact, the methods cited above differ from one another mainly in the specific procedure employed for implementing the required gradient computations). The application of alternate training procedures that eliminate the need for gradient computations as will be described in this chapter are of particular interest in the context of this problem.

Two specific benchmark trajectories that have received wide attention in performance evaluations are the "circle trajectory" and the "figure-eight trajectory". A recurrent network can be set up to produce these trajectories by requiring two output nodes in the architecture shown in Fig. 1 to generate oscillatory response of a sinusoidal form with a specified frequency. It is easy to see that requiring the two outputs to oscillate according to the relation

$$o_1(t) = A\sin\omega t \quad \text{and} \quad o_2(t) = A\cos\omega t$$

© 2001 by CRC Press LLC

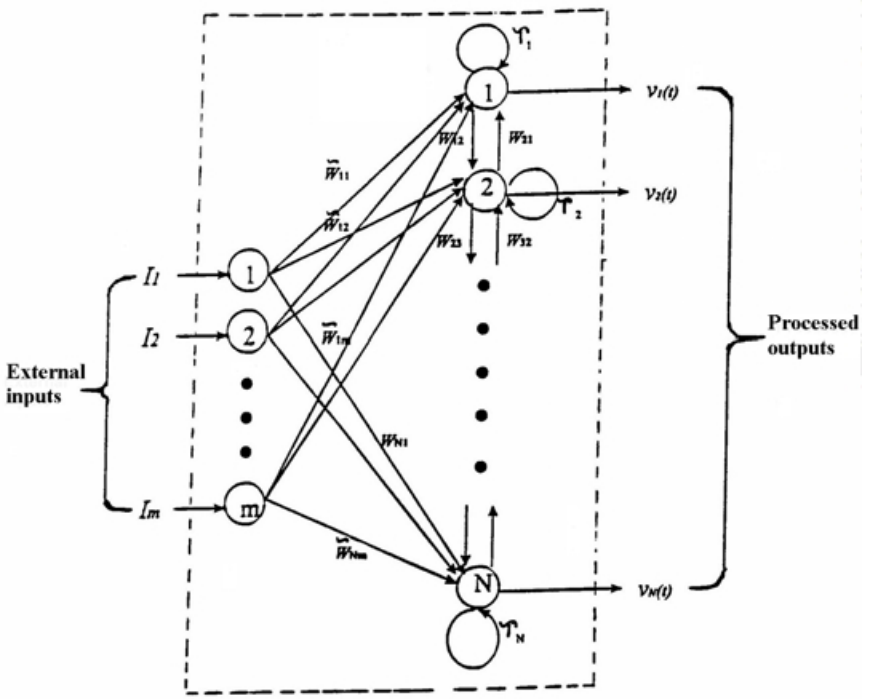

Figure 1. General architecture of an N-node recurrent neural network with m external inputs

with an arbitrary frequency ω would generate a circle with center at the origin and radius A on a two-dimensional plane with $o_1(t)$ and $o_2(t)$ as coordinates, while requiring the two outputs to oscillate according to the relation

$$o_1(t) = A\sin\omega t \quad \text{and} \quad o_2(t) = A\sin 2\omega t$$

would generate a figure-eight pattern passing through the origin of the $(o_1(t), o_2(t))$ plane. It is also easy to see that since the latter trajectory intersects on itself, the training problem is more challenging in this case compared to one of training a non-intersecting trajectory pattern. While these patterns are the ones that have been considered by earlier researchers to demonstrate the training efficiency, more general trajectories can also be formed by specifying the neural network outputs in appropriate forms.

An optimization framework can be developed for such a spatiotemporal learning task extending over a time horizon $\left[t_0, t_f\right]$ by specifying an error functional

$$\varepsilon = \sum_{i\in\vartheta}\int_{t_0}^{t_f} f(v_i(t) - v_i^d(t))dt$$

© 2001 by CRC Press LLC

where ϑ denotes the set of designated output nodes of the network and $v_i^d(t)$, $i \in \vartheta$, denote the desired output signals. The function $f(.,.)$ can be specified in various ways in terms of the L_1-norm or the L_2-norm of the deviation $v_i(t) - v_i^d(t)$ or any other appropriate measure. The training problem then reduces to minimizing this error functional with respect to the set of adjustable network parameters. An issue of some significance for practical applications is the flexibility available in tailoring an appropriate error functional. It may be noted that conventional gradient-based training procedures typically require an L_2-norm of the error, *i.e.*, selection of $f(v_i, v_i^d) = (v_i - v_i^d)^2$, mainly for simplicity in gradient evaluations. However, when an evaluation of error gradients is not needed, as is the case with the training procedures discussed in this article, we have a greater flexibility in formulating the error functional to be minimized.

2. INCREMENTAL LEARNING

When neural networks are trained in a supervised manner, there is a tendency for the training to proceed rapidly reducing the value of the specified error for some time, until a point is reached where no further training becomes possible. This corresponds to the case when the training has proceeded to a *local minimum*. In the present context, this condition may be visualized by considering the error surface in an N^2+2N+1 space (where the N^2+2N axes correspond to the adjustable parameters of the network and the final dimension corresponds to the error function), which indicates that the error has been reduced with respect to these parameters but has fallen into an energy well, from which a recovery with the type of parameter changes already used is not possible. In the specific application to the trajectory learning problem, which is of particular interest in this chapter, this situation corresponds to the neural network learning to generate a trajectory that reduces the error, but the generated trajectory not having the same shape as the desired trajectory.

In order to reduce the occurrence of becoming trapped in a local minimum, some method of controlling the evolution of trajectories during learning could be used. A simple way of overcoming the problem is by a process of *incremental learning*, which generates a set of intermediate learning goals. Let $\xi_o(t)$ denote the trajectory generated by the neural network at the start of training and $\xi_f(t)$ denote the final trajectory. It is desired to establish M learning goals, where the absolute error between one goal and the next is small. This can be accomplished by defining a sequence of learning goals as

$$\xi_n(t) = \xi_0(t) + n \cdot \Delta\xi(t), \qquad n = 0, 1, ..., M \tag{6}$$

where $\Delta\xi(t) = [\xi_f(t) - \xi_0(t)]/M$.

© 2001 by CRC Press LLC

For illustration, suppose it is desired to train a dynamic recurrent neural network of the form (1) to output the trajectory

$$v(t) = \sin(\pi t), \qquad 0 \le t \le 1.0. \tag{7}$$

Let $v_o(t)$ denote the initial trajectory output of the network for some initial set of parameters and initial states of neurons. An arbitrary number, say 100, of learning targets can be selected as

$$\xi_n(t) = v_0(t) + n \cdot \frac{[\sin(\pi t) - v_0(t)]}{100} \quad \text{for } 0 \le t \le 1.0, \quad n = 0, 1, ..., 100. \tag{8}$$

When $v_0(t) \approx \xi_i(t)$ within some predetermined error bound, the next learning target becomes $\xi_{i+1}(t)$. Learning progresses through these increments until the final desired target is reached. Fig. 2 shows a succession of these desired trajectories which represent incremental targets.

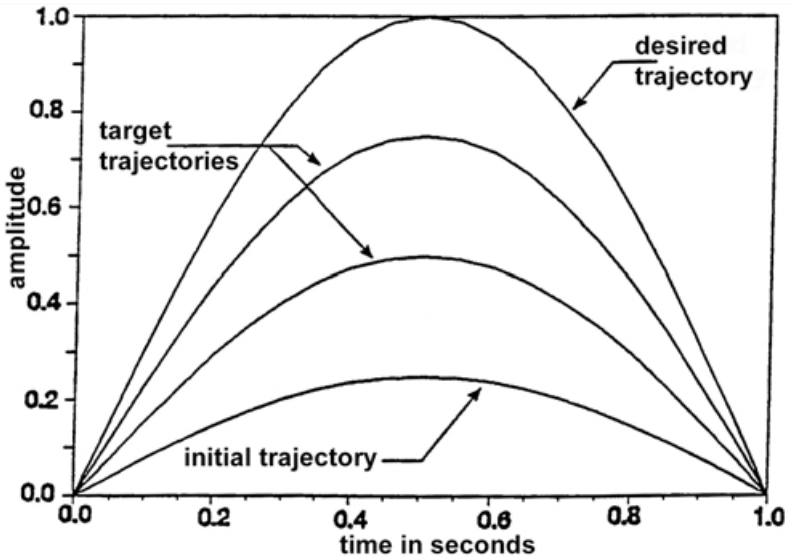

Figure 2. Target trajectories for increment learning

It may be noted that since the neural network being trained is characterized by nonlinear dynamics, the effort in moving from one incremental learning goal to another may not be uniform even when a uniform distance between these learning goals is implicit. This however is of no major consequence insofar as the overall learning performance is concerned since the motivation for modifying the learning goal is to provide a mechanism for perturbation of the error function

© 2001 by CRC Press LLC

during the training process, and the objectives of incremental learning are achieved when an appropriately large number of learning goals *M* is selected for implementation.

3. TEACHER FORCING

In training problems such as trajectory learning, where the desired output is available at every instance of time during the training process, using an appropriate mechanism to directly feed this information to alter the activation dynamics of the neural network provides several benefits. This formalism, referred to as *Teacher Forcing*, has been used by several previous researchers [Williams, 1989, Toomarian, 1992] in one form or another. The idea of including a teaching forcing signal in general supervised learning problems comes from the desire to supply additional instantaneous information from the teacher directly to the activation dynamics during the learning stage. The role of including this signal on the training performance can be understood from the analogy with the use of continuous feedback in reducing the error in closed-loop control systems. A temporal modulation of this signal as learning proceeds is often desirable so that the activation dynamics during learning progressively reduce to the activation dynamics during the recall stage.

In the present work, for improving the trajectory learning performance, a method of teacher forcing similar to the one suggested originally by Williams and Zipser [Williams, 1989] can be employed. In this scheme, the desired network output signals are used in place of the actual network outputs when fed back into the network via the recurrent connections. The actual outputs are still used for computing the error in order to determine whether the parameter updating action at any stage is favorable or not. The teacher forcing drives the network outputs closer to the desired signals as training progresses and the network is trained at each stage as if it were already generating the correct signal. This seems to significantly speed up learning, particularly at the beginning stages.

Upon completion of successful training, *i.e.,* when the error functional becomes zero, the teacher forcing will no longer exist and the network dynamics will revert to the usual dynamics described by (4). As pointed out by Toomarian and Barhen [Toomarian, 1992], there exist training scenarios (particularly arising in trajectory learning problems) where the error functional cannot be reduced to zero and consequently the activation dynamics of the neural network after training is completed, *i.e.,* during the recall phase, will be different from that specified by (4). To avoid this discrepancy, at some point in the training process, when confidence in the shape of generated trajectories is developed, the teacher forcing is disabled and the learning is progressed with the actual outputs of the network. Alternately, a temporally modulated teacher forcing scheme [Toomarian, 1992] that progressively reduces the amount of teacher intervention during the training phase can be employed; a simple mechanism for implementing such modulation is by multiplying the signal by a time-varying gain $\lambda(t) = 1 - e^{\varepsilon(t)/\rho}$, where $\varepsilon(t)$ is the measured error and ρ is an

© 2001 by CRC Press LLC

appropriately selected number sufficiently large (a large value of ρ relative to the expected values of error is recommended to prevent $\lambda(t)$ from becoming negative).

III. TRAINING BY METHODS OF LEARNING AUTOMATA

A. SOME BASICS ON LEARNING AUTOMATA

A *learning automaton* interacts adaptively with the environment it is operating in and updates its actions at each stage based on the response of the environment to these actions [Lakshmivarahan, 1981, Narendra, 1989]. Hence an automaton can be defined by the triple (α, β, T) where α denotes the set of actions $\alpha = \{\alpha_1, \alpha_2, ..., \alpha_r\}$ available to the automaton at any stage, $\beta = \{\beta_1, \beta_2, ..., \beta_m\}$ is the set of observed responses from the environment, which are used by the automaton as inputs, and T is an updating algorithm which the automaton uses for selecting a particular action from the set α at any stage. In the present context of neural network training, a specific action at any stage corresponds to the updating of the values of one or more parameters of the network.

For a *stochastic learning automaton*, the updating algorithm specifies a rule for adjusting the probability $p_i(n)$ of choosing a particular action α_i at stage n. Such a rule may be generally described by a functional relation of the form

$$p_i(n+1) = F(p_i(n), \alpha(n), \beta(n)). \tag{9}$$

The learning procedure at each stage hence consists of two sequential steps. In the first step the automaton chooses a specific action $\alpha(n) = \alpha_i$ from the finite set of actions available, and in the second step, the probabilities of choosing the actions are updated depending on the response of the environment to the action in the first step, which influences the choice of future actions.

An alternative way of specifying the updating algorithm is to define a state vector for the automaton and consider the transition of the state due to a certain action, which enables one to state the updating rule in terms of state transition probabilities. This approach has been quite popular in the development of learning automaton theory [Varshavskii, 1963]. For our application to neural network training, however, the action probability updating approach, with the updating algorithms specified in the form of equation (9), provides a simpler and more convenient framework.

For execution of training, the feedback signal from the environment, which triggers the updating of the action probabilities by the automaton, can be given by specifying an appropriate "error" function. The environmental response set $\beta(n)$ at any stage n can then be selected as the binary set $\beta(n) = \{0,1\}$, with $\beta = 1$ indicating that the selected action α_i is not considered satisfactory by the environment and $\beta = 0$ indicating that the action selected is considered

© 2001 by CRC Press LLC

satisfactory.[2] For a stochastic automaton with r available actions (*i.e.*, $\alpha = \{\alpha_1, ..., \alpha_r\}$), the updating rules can then be specified in a general form as follows:

For the selected action at the nth stage $\alpha(n) = \alpha_i$, if $\beta(n) = 0$ then

$$p_j(n+1) = p_j(n) - \gamma_j(p(n)) \qquad \text{for } j \neq i$$
$$p_i(n+1) = p_i(n) + \sum_{\substack{j=1 \\ j \neq i}}^{r} \gamma_j(p(n)) \tag{10}$$

whereas if $\beta(n) = 1$, then

$$p_i(n+1) = p_i(n) - \delta_i(p(n))$$
$$p_j(n+1) = p_j(n) + \frac{1}{(r-1)}\delta_i(p(n)), \qquad j \neq i \tag{11}$$

The functions $\gamma_j(\cdot)$ and $\delta_i(\cdot)$ are appropriately selected continuous-valued nonnegative functions. The summation $\Sigma\gamma_j$ in (10) and the division by (r-1) in (11) are to ensure preservation of probability measure (*i.e.*, sum of probabilities at (n+1) equals one).

The two sets of equations (10) and (11) specify a *reinforcement learning algorithm*. By tailoring the functions $\gamma_j(\cdot)$ and $\delta_i(\cdot)$ an appropriate degree of reinforcement in the selection of a particular action can be introduced. A scheme where both sets of equations are employed together is termed a *reward-penalty reinforcement scheme*. It is evident that in this scheme an action that is judged favorable is rewarded by having its probability of selection increased while an unfavorable action is penalized by having its probability of selection decreased. Another reinforcement scheme, termed *reward-inaction scheme*, employs the updating only for $\beta(n)$=0, whereas for $\beta(n)$=1 the action probabilities are maintained at the same values as before. These schemes and several other variations of them have been discussed in the literature [Lakshmivarahan, 1981, Narendra 1989]. Due to the stochastic nature of the framework, however, very few analytical results can be developed for these schemes and studies directed to the evaluation of performance (such as convergence, asymptotic behavior) typically employ simulation experiments.

It should be emphasized that (10) and (11) describe a general framework for tailoring a variety of specific training algorithms useful in particular applications by selecting $\gamma_j(\cdot)$ and $\delta_i(\cdot)$ appropriately as linear or nonlinear functions. In fact,

[2] In the literature on learning automata [Lakshmivarahan, 1981, Narendra, 1989], this case of β allowed to take two distinct values only is referred to as the *P*-model. More general models where β can take a number of values within an interval have also been discussed.

© 2001 by CRC Press LLC

a number of heuristic algorithms where $\gamma_j(\cdot)$ and $\delta_i(\cdot)$ may not have an analytical form can also be considered for realizing improved speed and accuracy in training. In certain applications of neural network training such constructions motivated by intuitive reasoning may indeed prove to be more efficient. An illustrative example of this will be demonstrated in a later section for application to the trajectory learning problem.

B. APPLICATION TO TRAINING RECURRENT NETWORKS

A principal advantage of the learning automaton approach is its ability to determine optimal actions among a set of possible actions and this is particularly useful in neural network training where a number of possible actions exist. For training the neural network described by (4), we will employ the learning configuration schematically shown in Fig. 3. The automaton actions are defined as either an increment or a decrement to any of the network parameters ω_{ij}, τ_i and g_i. For an N-neuron network, this corresponds to a set of $2(N^2 + 2N)$ single parameter updating actions. Multiple parameter actions can also be considered, with the number of possible actions in this case increasing to $2(N^2 + 2N)$!

The environment for this learning configuration comprises the neural network itself together with an appropriately specified error functional ε defined over the time interval $[t_0, t_f]$ as discussed earlier. The feedback signal to the automaton can be defined as

$$\begin{aligned} &\beta = 0 \text{ for action that reduces the error} \\ &\beta = 1 \text{ for action that does not reduce error.} \end{aligned} \tag{12}$$

As noted earlier, function $f(\cdot,\cdot)$ for computing the training error can be specified in various ways; for the examples that will be discussed later, we employed $f(v_i, v_i^d) = \left| v_i - v_i^d \right|$ to define the error functional ε. Subsequent to the determination that an action has reduced the error, the corresponding changes to the neural network parameters are retained. However, if the action increases the error, the corresponding parameter changes are not kept. Thus, the only modifications to the neural network structure come from those actions that reduce the value of the specified error.

© 2001 by CRC Press LLC

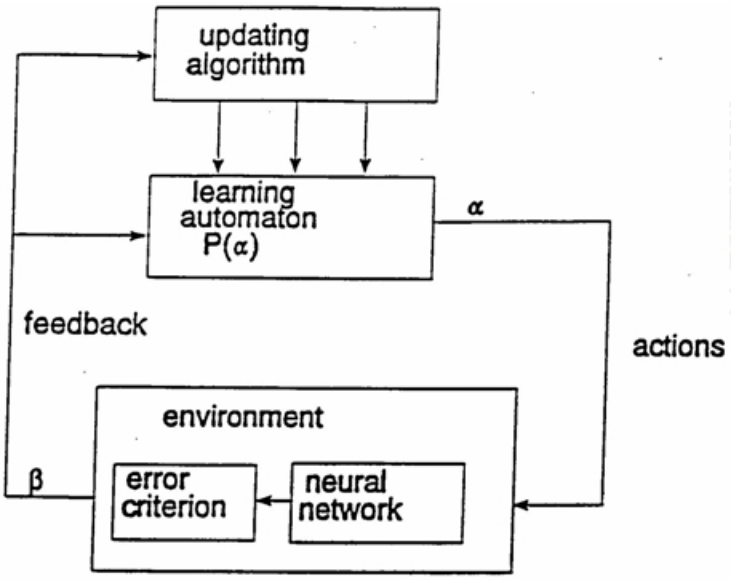

Figure 3. Learning configuration

A probability of selection is initially assigned to each action. Since no *a priori* knowledge generally exists as to which of the network parameters has the greatest influence in reducing the specified error, the entropy in learning is maximum at the beginning of training. Hence, a uniform distribution is used at the beginning for the action probabilities. As learning progresses, the probability associated with each action is changed. This probability determines the relative frequency with which a particular action will be selected. Thus, the more successful a particular action is at reducing the error, the more likely its selection will be in the future stages.

Any available prior knowledge on the qualitative behavior of the network being trained can be utilized in the process of initializing the training algorithm. The network described by (4) is one whose dynamics and equilibrium behavior have been extensively studied in the past [Sudharsanan, 1994, Sudharsanan, 1991a, Sudharsanan, 1991b] and the correlations of these results with the training performance can be exploited for the initial setting of parameter values. For illustration, some past results that underscore the role of high gain sigmoidal nonlinearities in ensuring desirable stability properties for the network equilibria [Sudharsanan, 1991a] and the observed correlation between selection of high gains and improvement in learning rates [Sudharsanan, 1991, Behrens 1991] could be usefully employed in the initial selection of g_i parameters for improving the efficiency of the training process.

In discussing the time-behavior of the training process, two types of convergence come into the picture: convergence of the training error and convergence of the automaton to some optimal action. Convergence of the error is assured by the nature of the learning algorithm. Since changes to the neural

© 2001 by CRC Press LLC

network structure come only from those actions that result in a reduction of the error, starting from any finite positive initial error, a monotonic decreasing sequence of positive real numbers is generated. This sequence is bounded and, from the monotone convergence theorem [Bartle, 1992, Condarcure, 1991], is convergent.

Under certain conditions, the learning automaton will converge toward some optimal action depending on the type of reinforcement rule used. By associating with each action a penalty probability, it has been shown in the literature [Narendra, 1989] that if the penalty probabilities are stationary, then the action probabilities will converge to an optimal action. In particular, for the linear reward-inaction scheme (*i.e.*, for $\gamma_j(\cdot)$ a linear function of the argument and $\delta_i(\cdot) = 0$ in the updating rules (10) and (11)), convergence is assured in this sense. It should however be noted that convergence of this type may not be desirable in the present context of neural network training. The penalty probabilities are not known at the start of training and their distribution may not be stationary since the structure of the neural network is constantly changing during the training process. An action that may produce a favorable response at some point in the training process may not yield a favorable response at a later time. Furthermore, the gains g_i and the time constants τ_j are constrained to be nonnegative and hence cannot be continually decremented to take on negative values. Therefore, convergence of the learning automaton to an optimal action is not desirable and will not occur when the reward-penalty reinforcement rules are used (since the probability of any action approaching 1 is not possible with this reinforcement scheme for a nonstationary environment [Narendra, 1989]).

C. TRAJECTORY GENERATION PERFORMANCE

The performance of the training approach described in the last section has been tested in the task of learning continuous trajectories. We shall give the results for a circle trajectory of specified radius 0.5.

Simulation experiments were conducted using a fourth-order Runge-Kutta algorithm for studying the temporal dynamical behavior of the neural network. A time increment of $0.02T$ was selected as the integration time constant, where T is approximately the period of the trajectory to be generated. For implementing the actions of the learning automaton, it is necessary to generate an output function $\alpha(n)$, which maps the stage number n into a selection of the appropriate action to take in a probabilistic fashion. Since these action probabilities are unknown at the start of the experiment, they are initialized to a uniform distribution. Then, as the experiment progresses and successful actions are found, a discrete probability density function is built up, with the probability for a particular action $\alpha_i(n)$ being increased or decreased according to the specific reinforcement in the form of (10) or (11). As the density function is being generated, it is used for the selection of actions by an inverse distribution method. This is done by generating uniformly distributed random numbers (by a standard procedure such as the Lewis-Payne method [Lewis, 1973]) and then summing the numbers in the density function to create a distribution function

© 2001 by CRC Press LLC

until the generated random number is greater than the sum. The action is then selected at the point where the sum of the densities is greater than the uniform random number.

A six-node network (*i.e.* $N = 6$) with two nodes designated as the output nodes $\{o1, o2\}$ and with no externally applied inputs was trained to generate the desired circle trajectory. In order to attempt to better control the trajectory rise time, rather than try to force the network to generate the circle with an unknown rise time, a parameter η was introduced to modify the desired outputs in the form

$$\begin{aligned} v_{o1}^d &= 0.5(1-e^{-\eta t})\sin \pi t \\ v_{o2}^d &= 0.5(1-e^{-\eta t})\cos \pi t \end{aligned} \tag{13}$$

For initializing the network, the weights w_{ij} were set to 0.0, the gains g_i were set to 10.0, and the time constants τ_i were set to numbers randomly distributed around 6.0. The initial states of the neurons $v_i(0)$ were chosen to be small random numbers centered around zero. Incremental learning was used with 100 intermediate learning targets established as discussed in Section 2.C.

A brief explanation on the role of parameter η seems useful. Observe that with the selection of $v_{o1}^d(t)$ and $v_{o2}^d(t)$ as in (13), we have

$$v_{o1}^{d^2}(t) + v_{o2}^{d^2}(t) = 0.25 + 0.25(e^{-2\eta t} - 2e^{-2\eta t})$$

and hence as t becomes progressively larger, $v_{o1}^d(t)$ and $v_{o2}^d(t)$ approach the desired signals $0.5\sin(\omega t)$ and $0.5\cos(\omega t)$ respectively for any selection of $\eta > 0$. However, by selection of a sufficiently large η, a scaling of time can be achieved thus accelerating the convergence to desired final values. It may also be noted that the use of $v_{o1}^d(t)$ and $v_{o2}^d(t)$ as in (10) is motivated by our desire to generate the desired circle trajectory from the starting values of $v_{o1}^d(t) = 0$ and $v_{o2}^d(t) = 0$, which corresponds to a more challenging learning task than the case when the initial point is selected to lie on the desired circle. Selection of η hence offers a mechanism for controlling the trajectory rise time which is a highly desirable feature. In the experiments that will be reported later, a representative value of $\eta = 10$ was used.

To test the effects of selecting alternate reinforcement rules and parameter updating actions on the training performance, several experiments [Condarcure, 1991] were conducted. For the sake of brevity, only two illustrative cases will be described in the following.

© 2001 by CRC Press LLC

1. EXPERIMENT 1

In this experiment, a simple linear reward-penalty reinforcement scheme obtained by defining $\gamma_j(\cdot)$ and $\delta_i(\cdot)$ in (10) and (11) as linear functions was used. The reinforcement rules in this case will take the following form:

For an automaton with r available actions, with the selected action at the nth stage $\alpha(n) = \alpha_i$, if $\beta(n) = 0$, then

$$p_j(n+1) = (1-\gamma)\, p_j(n)\,, \qquad j \neq i$$

and

$$p_i(n+1) = \gamma + (1-\gamma)\, p_i(n) \tag{14}$$

whereas if $\beta(n) = 1$, then

$$p_i(n+1) = (1-\delta)\, p_i(n)$$

and

$$p_j(n+1) = p_j(n) + \frac{\delta}{r-1}(1-\gamma)\, p_i(n)\,, \qquad j \neq i \tag{15}$$

In (14) and (15), γ and δ are constants that may be selected appropriately in the ranges $0 < \gamma < 1$ and $0 < \delta < 1$. Also, from (14) it is evident that an action α_i considered favorable will result in a reduction of the probabilities p_j (for $j \neq i$) by a percentage γ while increasing the probability p_i by an amount such that the sum of the probabilities at stage $(n+1)$ is 1. Similarly, when action α_i is unfavorable, the probability p_i is reduced by a percentage δ while the remaining probabilities p_j (for $j \neq i$) are correspondingly increased such that the sum of the probabilities remains at 1, as reflected by the form of the updating rules in (15).

For the numerical simulations we used the values $\gamma = 0.02$ (corresponding to 2% change in the case of a favorable action) and $\delta = 0.01$ (corresponding to 1% change in the case of an unfavorable action); these values were determined from experimentation to give good results.[3] A single parameter action (increment or decrement), defined as an incremental change to one network parameter that is continued until it is no longer successful for a given trial, was employed. The error functional discussed earlier (*viz.* Eq. (5) with $f(v_i\,,\, v_i^d) = |v_i - v_i^d|$) was used and it was required that the value of the error be reduced to 0.06 before moving from one learning goal to the next. Teacher forcing was used to help accelerate the learning process at the start and was disabled at the 50th learning increment when the shape of the actual output trajectory was sufficiently close to the desired trajectory.

---

[3] In earlier work on learning automata [Lakshmivarahan, 1981, Narendra, 1989], it is observed that a certain degree of asymmetry between the reward and the penalty parameter results in general in a desirable training behavior, *i.e.* rewarding a favorable response more than penalizing an unfavorable response is generally preferable.

© 2001 by CRC Press LLC

Fig. 4a depicts the parameter changes or actions that were attempted by the automaton for each learning increment. It may be noted that learning was very easy when teacher forcing was active, which agrees well with intuition. After the 50th step, when teacher forcing was disabled, learning became more difficult, as the network must meet the learning goals on its own. This continued until about step 82, when the automaton developed enough experience in making better selections. The results of this experiment with the network trained for 4 cycles (each cycle corresponding to one period of the sinusoidal waveforms) and then continued to run for another 8 cycles is shown in Fig. 4b, which clearly indicates the stability of the generated limit cycle. It may be noted that only the first three cycles during which the trajectory evolves into the limit cycle are distinguishable while the rest overlap.

2. EXPERIMENT 2

In this experiment the primary goal was to study the effects of allowing multiple parameter actions, *i.e.,* sets of parameters to be updated simultaneously. It is to be noted that since the neural network is nonlinear, the effect of changing more than one parameter at a time is not the same as the combined effect resulting from changing them one after another. For a 6-node network ($N = 6$), the number of possible actions now increases to 48! (*i.e.,* $(N^2+2N)!$). Consequently, to reduce the memory requirements, two options were exercised. The first is to limit the actions to those that update 10 parameters or less at a time. The second is to store the successful actions in a repertoire for a preferential selection at the later stages. An action is added to the repertoire if it is used successfully to reduce the error, which is the reward. If an action in the repertoire does not successfully reduce the error, it is penalized by being removed from the repertoire. Once all actions existing in the repertoire are used at any stage, new actions are selected randomly from the remaining set of available actions based on a uniform distribution. The following learning reinforcement is also used. When rewarded, the probability for an action in the repertoire stays at its previous value, whereas for a successful action not in the repertoire it is increased from its value in the uniform distribution to a higher value. When penalized, the action probability is reduced to its value in the uniform distribution.

In the framework of the reinforcement rules discussed earlier, the present updating mechanism corresponds to a nonlinear reinforcement scheme, more general than the linear reinforcement rules used in Experiment 1. An analytical modeling of the updating rules is however more difficult to obtain in this case.

© 2001 by CRC Press LLC

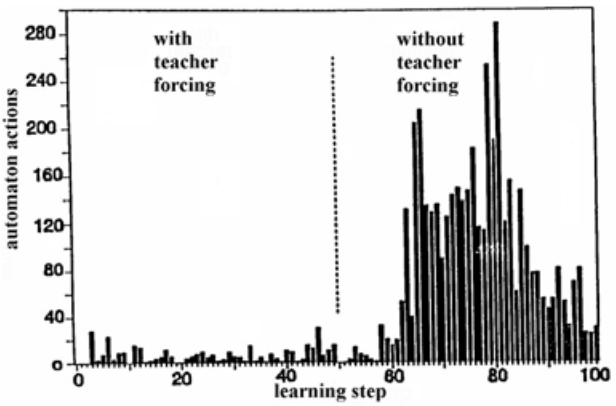

Figure 4a. Automaton actions per learning increment

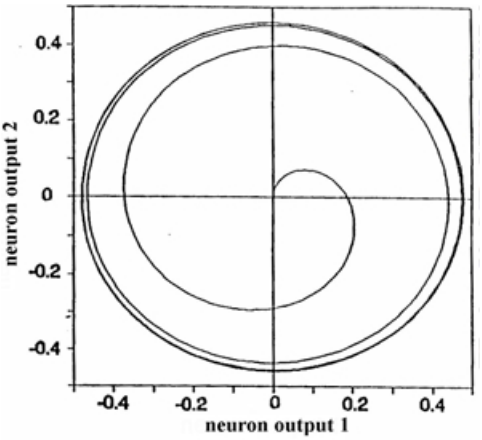

Figure 4b. Neural net output trajectory in Experiment 1

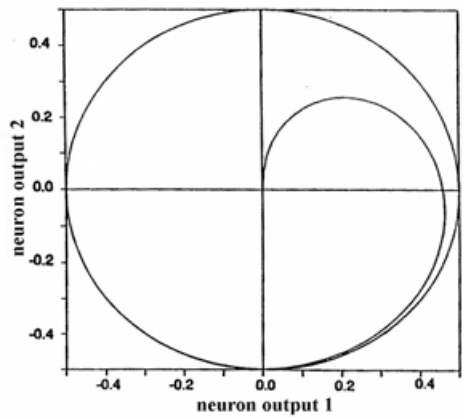

Figure 4c. Trajectory generated in Experiment 2

© 2001 by CRC Press LLC

To provide a greater ease of implementation, in this experiment the activation gains g_i were permanently set to the value 10.0 and learning was restricted to changes in the other parameters (w_{ij} and τ_i). Incremental learning was used as before with 100 learning steps. The result of this experiment is shown in Fig. 4c, which indicates a substantial improvement in the achieved performance over the linear reinforcement-single parameter action case considered in Experiment 1. As can be observed, the trajectory rise time is also significantly reduced in this case (to about 0.2 sec) and the evolution into the final orbit is almost complete within half a cycle. Fig. 4 shows the results of the experiment with the network trained for 4 cycles and then continued to run for another 8 cycles. The remarkable accuracy with which the recall cycles overlap is worthy of emphasis and this represents a level of performance significantly better than that provided by any of the existing training procedures.

As a further note, in the two experiments described above, the training took approximately 2500 attempted actions to reach the final learning goal. It must be emphasized that the computations required at each step are extremely simple (involving updating of probability vectors) and are almost negligible compared to the evaluation of gradients required by existing methods, which makes the present scheme more attractive to implement. Also, comparing the performance depicted in Figs. 4b and 4c with the other available results for the trajectory learning problem, it may be noted that this level of accuracy in generating the circle trajectory could only be achieved in Toomarian [1992] when the learning was started with the initial values of the neuron states adjusted such that the initial point is already on the desired circle (specifically, case 3 in Toomarian [1992]). In contrast, in our case the learning was started with the initial states set at arbitrary small random values. It must also be noted that this level of performance was achieved even when the learning was restricted to only the weights w_{ij} and the time constants τ_i. It is conceivable that even better performance levels can be realized by permitting the activation gains also to be updated, although at the cost of increased memory requirements. What is particularly noteworthy, however, is the significant reduction in computational requirements compared to the conventional gradient-based algorithms.

IV. TRAINING BY SIMPLEX OPTIMIZATION METHOD

A. SOME BASICS ON SIMPLEX OPTIMIZATION

In order to facilitate some understanding on the basics and motivation for the Simplex algorithm, consider the following simple example. Suppose a simple guessing game is being played between a player and a computer. Suppose that the computer has selected an arbitrary nonlinear function, for example $y = f(x)$, and that the player has to guess the value of the variable x that when substituted into the above nonlinear equation (unknown to him) would result in the global minimum of the function. The player can guess the variable value by keying into the computer a number and observing the corresponding function output provided by the computer (if the player somehow manages to guess the

© 2001 by CRC Press LLC

correct value of the parameter, the computer would inform him that he has achieved minimality).

There are several ways by which the player can obtain the parameter that would result in the minimum value of the function. First, he can keep guessing the parameter randomly until he found the correct one. This method, however, could take an infinitely long period of time, especially when presented with a highly nonlinear and complex multivariable function (*i.e., x* becomes a vector variable). Second, he can try to compute the gradient of the function and use it to guide him to the correct parameter. However, with this method if the function is nonlinear, complex, and multivariate, its gradient may be difficult and expensive to compute. Third, he can make use of the knowledge given to him by the computer, *i.e.,* use the returned value of *y* to strategically locate the desired value of *x*. Consider the following simplified example. Suppose that the function selected by the computer is as shown in Fig. 5.

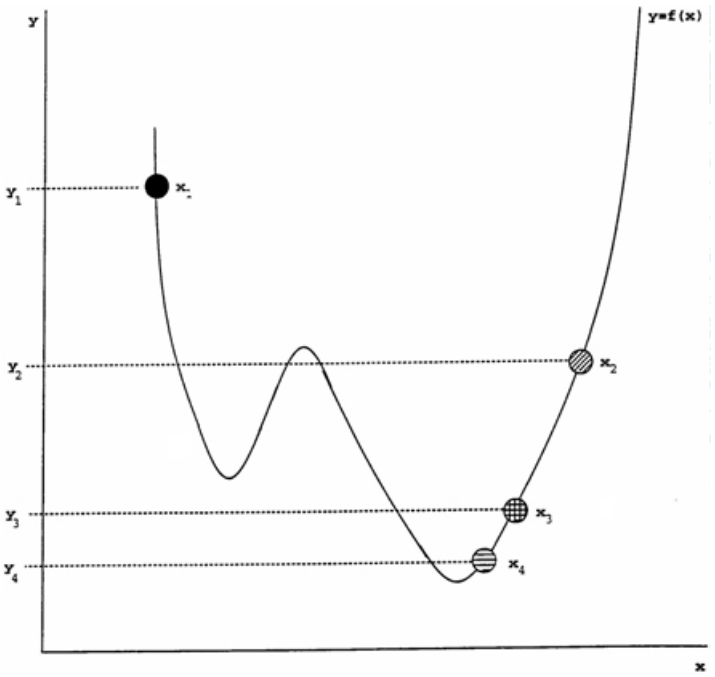

Figure 5. Illustration of the structured search approach of the Simplex optimization algorithm

Also suppose that the first two guesses are x_1 and x_2, and the corresponding results are y_1 and y_2 respectively (see Fig. 5). Further suppose that with the knowledge obtained, *i.e.*, the values of y_1 and y_2, an estimate of the variable x_3 resulting in a lower functional value than those given by x_1 and x_2 can be obtained and that this process can be repeated until some degree of optimality is

© 2001 by CRC Press LLC

reached. It is easy to see that if such a structured iterative optimization method can be implemented and applied to this example, the parameter x_4 whose corresponding functional value is the lowest among the four guesses can be obtained. Indeed, the described process is that offered by the Simplex optimization algorithm. Hence the Simplex algorithm may be viewed as a method that strategically searches for the optimal solution based on the information obtained, without needing to know the mathematical expression for the function itself or calculate its gradient at every iteration. The fact that function gradients need not be computed with this method makes it an attractive optimization method especially when applied to complex multivariate functions or to systems such as a recurrent net. Another characteristic of this method that is of significance is its ability to escape the local minima of a function even though it is a simple downhill direct search method. This characteristic is also illustrated in Fig. 5. Before describing the series of steps involved in the simplex iteration, it is appropriate at this point to give a brief discussion on the development of the present algorithm.

A simplex is a geometrical figure consisting of *N*+1 points (or vertices) in an *N*-dimensional space. In a two-dimensional space, a simplex is a triangle and in a three-dimensional space, it is a tetrahedron. The Simplex algorithm described here is due to Nelder and Mead [Nelder, 1965] and is not to be confused with the Simplex method associated with linear programming. It is a direct downhill search method applicable to any multidimensional problem that requires only function evaluations and not the derivatives. This method, though extremely robust, can be slow in converging especially for problems of high dimensionality. However, in regard to neural network training, the inefficiency of this method, *i.e.,* its slow convergence in high dimensional spaces, can be reduced significantly as will be discussed in a later section. The storage requirement of this method is approximately N^2.

The reason for requiring *N*+1 simplex vertices for an *N*-dimensional optimization problem can be readily shown. Consider for illustration the one-dimensional function, $y = f(x)$. In order to search a region for *x*, some sort of boundary must be defined. In the one-dimensional case where the region is bounded by lines or curves, only two points are needed to enclose a region as illustrated in Fig. 6. With these two points the entire region of the function can be searched, if necessary, using the basic operations of expansion and contraction associated with the Simplex algorithm. These are implemented by keeping the better of the two points fixed, and by either expanding or contracting the other point (worse point) with respect to the fixed point the entire region bounded by the lines or curves can be searched if necessary. Similarly, for a two-dimensional function, such as $z = 5x + 3y$, a region (within a plane) can be uniquely defined, enclosed, and searched by three points following the same expansion and contraction operations. Extending the argument to an *N*-dimensional function, it is clear that *N*+1 vertices are required to bound and search a *N*-dimensional region.

© 2001 by CRC Press LLC

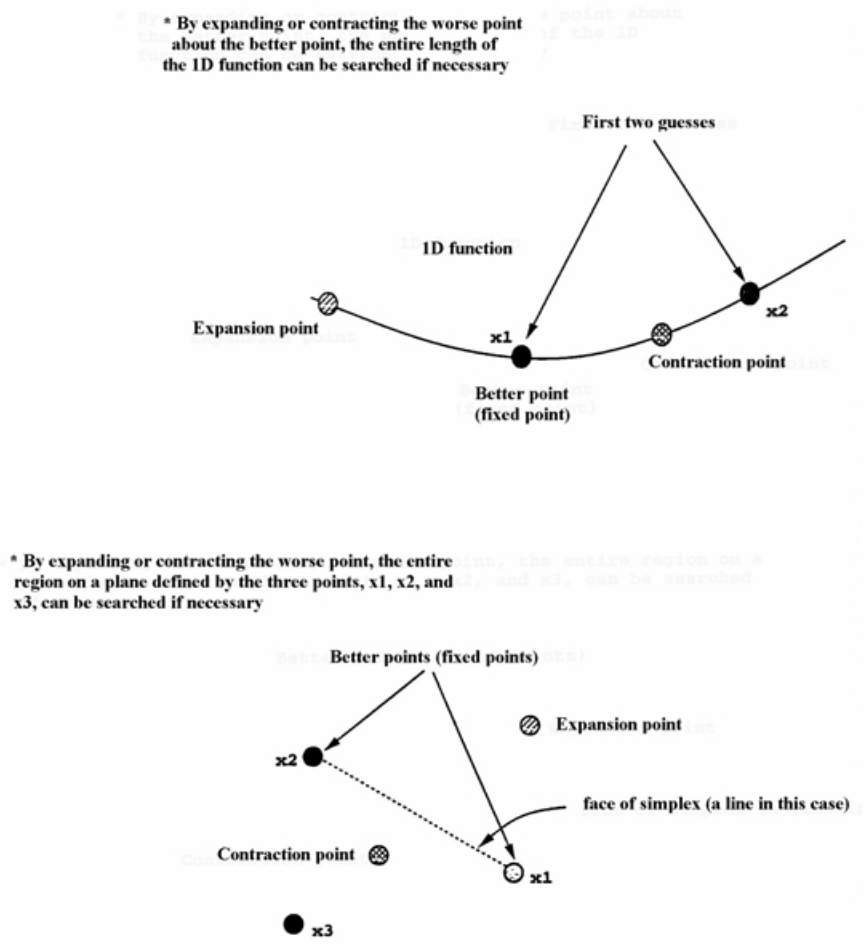

Figure 6. Searching in the N-dimensional space with N+1 simplex points

The simplex algorithm starts with *N*+1 points that can be either arbitrarily chosen or strategically obtained. The algorithm then moves the set of simplex points downhill in the function space, however complex it may be, through a series of steps. Most of the steps executed involve moving the point corresponding to the highest functional value (or lowest functional value in a maximization problem) through the opposite face of the simplex to a point with a lower functional value. This process is illustrated in Fig. 7, which shows 4 simplex points in a 3-dimensional space. In Fig. 7, it can be seen that the simplex point with the highest functional value is moved across the face of the simplex formed by the remaining 3 simplex points, the face being a plane defined by the 3 points in this case, to a location with a lower functional value. This step is generally called a *reflection* operation. If allowed to do so, the method expands the simplex in steps in one direction or another (a precise

© 2001 by CRC Press LLC

mathematical description of the expansion operation will be given in the next section). Contraction of a simplex point occurs when neither the reflection operation nor the expansion operations yield a better simplex point. In the event that neither the contraction, expansion, nor reflection operation yields a lower functional value, the method shrinks itself around the best point.

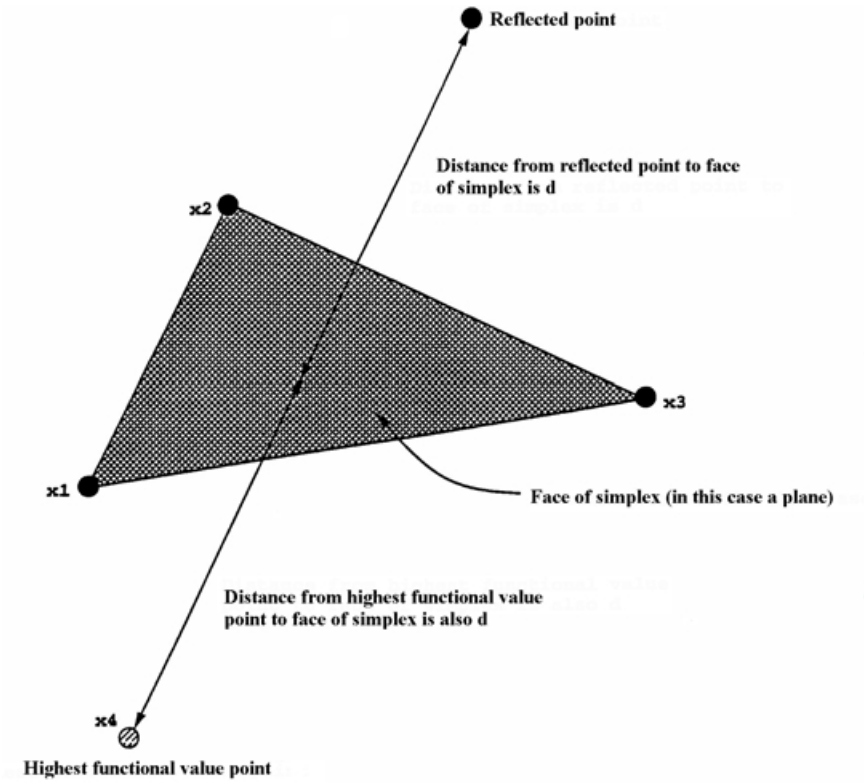

Figure 7. Illustration of reflexion point

The series of steps mentioned above can be mathematically represented by two basic expressions. Let us first define the various parameters that will be needed. Let H_{sp} denote the simplex point with the highest functional value, N_{sp} denote the new simplex point that will replace H_{sp} (that needs to be computed), R_{sp} denote the remaining simplex points (all points excluding H_{sp}), L_{sp} denote the simplex point with the lowest functional value, and S_{sp} denote the simplex point to be shrunk. Also, let α denote the parameter that controls the amount of expansion or contraction, and φ denote the parameter that controls the amount of shrinking. Let N_D denote the dimensionality of the problem (*i.e.,* number of points in the simplex).

© 2001 by CRC Press LLC

The two equations that summarize the various steps encountered in the Simplex algorithm are

$$N_{sp} = \frac{R_{sp}}{N_D}(1-\alpha) + H_{sp}\alpha \tag{16}$$

$$N_{sp} = L_{sp}(1-\varphi) + S_{sp}\varphi \tag{17}$$

Eq. (16) is used for reflecting, expanding, or contracting a simplex point with the parameter α controlling the amount of expansion or contraction. Note that the reflection operation is similar to the expansion operation. The difference between them is the amount by which they are moved across the simplex face. More specifically, in the reflection operation the simplex point is moved to a location across the simplex face that is exactly the same distance away from the face before it is moved; hence the term reflection (see Fig. 7 for clearer illustration). The expansion operation on the other hand moves the simplex point across the face of the simplex to a distance farther away as illustrated in Fig. 8. Since α controls the amount of expansion and contraction, it is clear that α must take on specific values for executing the three operations. Specifically, reflection across the simplex face is achieved with α = -1, expansion across the simplex face is achieved by a value of α < -1, while contraction is achieved with a value of α satisfying 0 < α < 1.

The flow of the Simplex algorithm, *i.e.,* the order in which the abovementioned operations are performed, will be discussed in a later section. Although (16) is used for reflecting, expanding, and contracting a simplex point depending on the value of α, the three different operations will be differentiated for clarity from here on. In particular, for reflection operation, (16) is kept unchanged with the parameter α, whereas for the expansion and contraction operations, the parameter α in (16) will be replaced by β and γ, respectively. Note that the fraction R_{sp}/N_D in (16) is a point on the simplex face. In fact, it can be readily shown that R_{sp}/N_D is the center-of-mass of the simplex face and hence is termed the centroid in the later discussion. Eq. (17) is used when neither reflection, expansion, nor contraction of the simplex point yields a lower functional value point. Eq. (17) is in fact a contraction operation around the simplex point with the lowest functional value L_{sp}. The parameter φ in (17) controls the amount of shrinking and can only take on values between 0 and 1. The detailed implementation strategy for neural network training is discussed in the next section.

© 2001 by CRC Press LLC

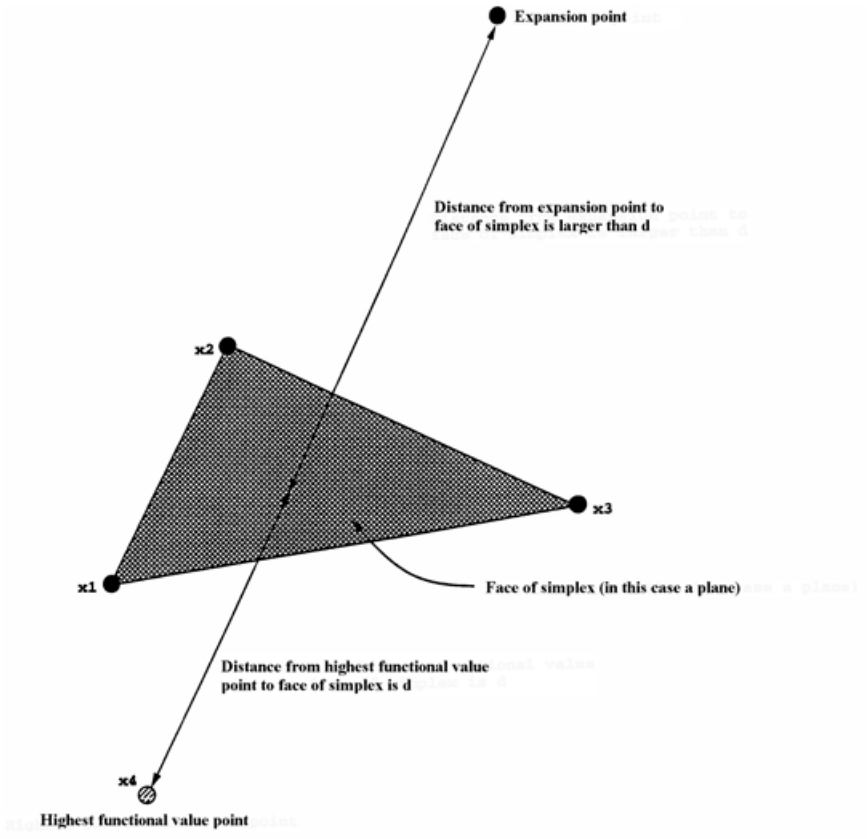

Figure 8. Illustration of expansion point

B. APPLICATION TO TRAINING RECURRENT NETWORKS

The simplex approach is a powerful optimization tool and has been used quite successfully in handling a variety of optimization problems [Wong, 1998, Duan, 1992] with nonlinear objective functions. The motivation for employing this approach in the present work of training a neural network, particularly in preference to the error backpropagation methods (and also to the more general steepest descent optimization approaches), can be explained from the following simple analogy.

The backpropagation approach can be regarded as similar to physically placing a person in a mountainous terrain with his objective being to move to the lowest elevation in that particular terrain (the mountainous terrain symbolizes, in the context of optimization, the peaks and valleys of the object function to be minimized). Having no additional information, other than the knowledge of his own initial elevation, his wisest option is to go down the steepest slope he can find and hope that it will lead him to the lowest elevation. Obviously his ending point will depend on where he starts. If he had been placed right above the

© 2001 by CRC Press LLC

global minimum elevation, then he will easily fulfill his objective with the selected strategy. However such a situation could indeed be very rare. Furthermore, how will he know that he has reached the lowest elevation if at all he does? It is more likely that he will stop at the first valley he reaches (a local minimum) and assume that he has found the global lowest elevation when clearly he has not (this illustrates the reason why backpropagation almost always ends up with a sub-optimal solution). Of course if he has enough energy left after the descent, he can always climb out of the valley he has found and try to find a lower elevation (similar to the operations of some modified backpropagation algorithms with a momentum term). However the question still remains unanswered - How will he know that he has indeed reached the global minimum?

With the simplex approach however, it is like randomly placing a group of people, instead of one person, at various selected initial points on the mountainous terrain. Now each person within this group knows his own elevation and spatial position but not the elevations and positions of the others. What would they do to meet the combined objective of finding the lowest elevation point? The wisest thing is to share their information, which is their elevation and spatial position, and have the person with the highest elevation move to a new position calculated from the rest of the group's elevations and positions on the terrain. Once this person has reached his new calculated position he would then report back his new elevation and spatial position to the group and the whole process starts again. With enough iterations, the group must finally converge to a point that will be close to the lowest elevation. One can see that the Simplex algorithm logically and efficiently overcomes settling into a sub-optimal solution as in the backpropagation algorithm. By a repetitive implementation with different sets of initial starting locations, the outcome of the simplex search can be made even more efficient in seeking out the true global minimum elevation. Observe that if the group of people were to record the spatial position and elevation of the point at which they converge, randomly re-position themselves around the terrain, and start the process all over again, they may eventually converge to an elevation that is closer to the true minimum. By repeating this process an arbitrarily large number of times, the group is bound to find the global lowest elevation with probability approaching 1. However the only drawback of this implementation is that if there are too many people in the group, the amount of computation needed to find the new position will increase correspondingly since there is now more information to process.

An implementation of this strategy for a supervised training of the neural network in order to minimize the training error

$$\varepsilon = \frac{1}{K} \sum_{i \in O} \sum\nolimits_{k=1}^{K} | o_i(k) - \hat{o}_i(k) | \tag{18}$$

will now be described. In the error criterion formulated above $\hat{o}_i(k)$, i=1,2,…,n, denotes the neural network outputs which are the estimates of the

© 2001 by CRC Press LLC

desired outputs denoted by $o_i(k)$, i=1, 2, …, n, where n denotes the total number of neural network outputs and K is the total number of training vectors used. The simplex is initialized by selecting an arbitrary set of N+1 points in the N-dimensional weight space, where each point corresponds to a selection of weight values (*i.e.* a vector of dimension N). This selection is made by randomly assigning all weight values within certain chosen bounds W_{max} and W_{min}. With respect to a neural network, the dimension of the weight space, N in this case, is determined by the size of the neural network (*i.e.* N is the total number of interconnections). Fig. 9 shows an illustrative case of 4 simplex points (for a problem with 3-dimensional weight vectors). The simplex evolution strategy [Nelder, 1965] is then executed, which involves determining the point where ε has the largest value and computing the centroid of the remaining simplex points. ε is a function of the neural network's output $\hat{o}_i(k)$ and the desired output $o_i(k)$ (for i=1, 2, …, n). For a recurrent neural network, such as that shown in Fig. 4, the neural network output is given by

$$\hat{o}_j(n) = \sum_{i=0}^{p} w_{ji}(n)\, y_{ji}(n) \tag{19}$$

where $w_{ji}(n)$ is the synaptic weight connecting the output of neuron i (in the hidden layer in this case) to the input of neuron j (in the output layer in this case) at iteration n, and $y_{ji}(n)$ - is the output signal of neuron i going into the input of neuron j at iteration n.

Note that there are p+1 neurons in the hidden layer as formulated in (19). The centroid of the simplex points, excluding the highest ε, is calculated by averaging the sum of the corresponding elements of each of the simplex points. For example, to illustrate the calculation the centroid of the remaining three simplex points, s_1, s_2, and s_3, in Fig. 9, let the weight values associated with them be

$$s_1 = \begin{bmatrix} w_{11} \\ w_{12} \\ w_{13} \end{bmatrix}', \quad s_2 = \begin{bmatrix} w_{21} \\ w_{22} \\ w_{23} \end{bmatrix}, \quad s_3 = \begin{bmatrix} w_{31} \\ w_{32} \\ w_{33} \end{bmatrix}. \tag{20}$$

© 2001 by CRC Press LLC

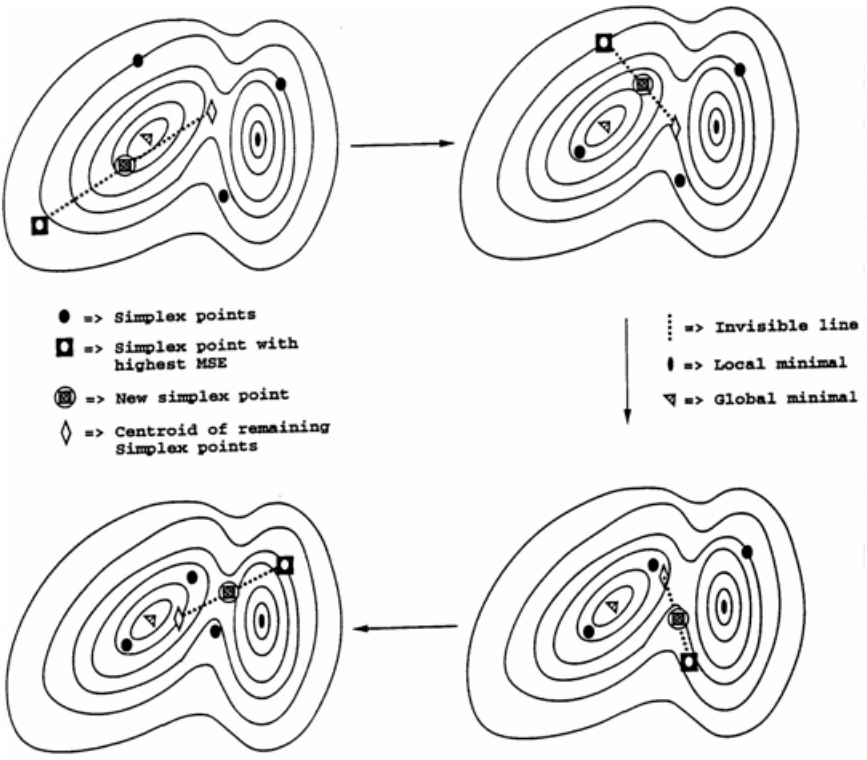

Figure 9. Convergence of Simplex algorithm to a global solution

The centroid, c, is

$$c = \begin{bmatrix} (w_{11} + w_{21} + w_{31})/3 \\ (w_{12} + w_{22} + w_{32})/3 \\ (w_{13} + w_{23} + w_{33})/3 \end{bmatrix}. \tag{21}$$

In general, for an N-dimensional weight space, the centroid may be calculated as

$$c_i = \frac{1}{N} \sum_{j=1}^{N} w_{ji} , \quad \forall i . \tag{22}$$

After the centroid is calculated, a new simplex point is then created by a reflection, expansion, or contraction which involves an operation that consists of joining the centroid computed to the simplex point with the highest ε by an invisible line and locating an expansion point or a contraction point on this line as shown in Fig. 10. The highest ε point is then replaced by the newly generated

© 2001 by CRC Press LLC

point to form the new simplex on which the set of operations is repeated. The reflection, expansion, and contraction points are new points obtained using the centroid and the highest ε point via operations similar to extrapolation and interpolation between these two points. First the ε value corresponding to the centroid is found with (18) and (19).

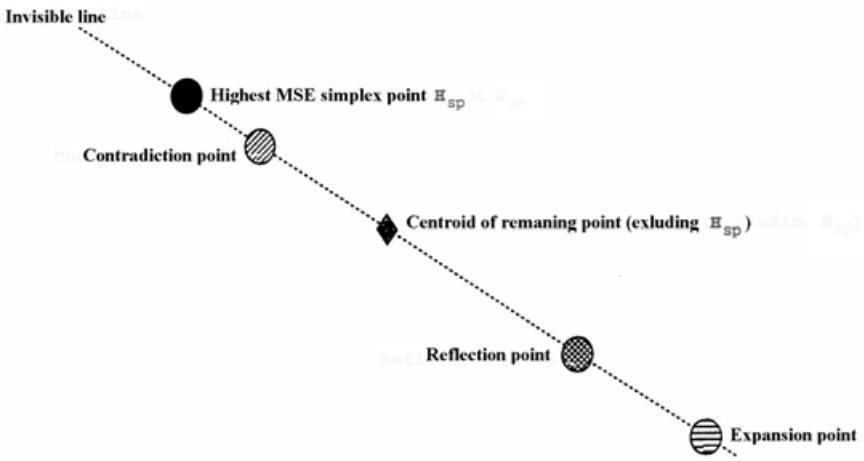

Figure 10. Illustration of reflection, expansion, and contraction operations in the Simplex algorithm

Next, the reflection point along with its ε is calculated. The reflection point is calculated by

$$Rf_{sp} = \frac{R_{sp}}{N}(1-\alpha) + H_{sp}\alpha \tag{23}$$

where Rf_{sp} is the reflection point, H_{sp} is the simplex point with the highest ε, N is the dimension of the weight space (in this case the total number of interconnections in the recurrent network), R_{sp} denotes the remaining simplex points excluding H_{sp}, and α, a parameter that controls the scale of reflection, is selected to be –1 as discussed earlier. If the ε associated with this reflection point is less than the highest ε point, the reflection point is further expanded via the following equation

$$Ex_{sp} = \frac{R_{sp}}{N}(1-\beta) + H_{sp}\beta \tag{24}$$

© 2001 by CRC Press LLC

where Ex_{sp} is the expanded simplex point and β, a parameter that controls the amount of expansion, is selected to be less than -1 (all other parameters in (24) are those defined in (23)). The specific value of β to be selected needs determination through conducting simulation experiments. In the present work, a value of β = -3 was found to yield the best results. In general β can take other values for different applications. If the expanded point of (24) is still less than the highest ε point, it becomes the new simplex point, else the reflected point found in (23) becomes the new simplex point.

If however ε for the extrapolated point is greater than the highest ε value (*i.e.*, H_{sp}), the centroid becomes the new simplex point. If the ε value corresponding to the centroid is greater than the highest ε value, the contraction point, along with its ε, is calculated via an operation similar to interpolation. That is, the contraction point is calculated by

$$Cn_{sp} = \frac{R_{sp}}{N}(1-\gamma) + H_{sp}\gamma \tag{25}$$

where Cn_{sp} is the contracted simplex point and γ, a parameter that controls the amount of contraction, is selected to be less than 1 (all other parameters in (25) are, again, those defined in (23)). Once again, an appropriate value of γ needs to be determined from experimentation, and it was determined for the present application that 0.5 yielded the best results. If the ε of the contraction point is less than the highest ε point, the contraction point then becomes the new simplex point. If however the ε of the contraction point is higher than the highest ε point, another action would have to be taken. At this point, it is apparent that the set of simplex points is located in an adverse situation. In such scenarios, the simplex points are contracted relative to the best simplex point in all directions thereby shrinking the size of the simplex. Consequently the new set of simplex points is obtained with

$$N_{sp} = L_{sp}(1-\varphi) + S_{sp}\varphi \tag{26}$$

where N_{sp} is the shrunk simplex point, L_{sp} denotes the simplex point with the lowest ε, S_{sp} denotes the simplex point to be shrunk, and φ, a parameter that controls the amount of shrinkage, is selected to be 0.75 (once again after several simulation exercises).

For implementation in the present context, the algorithm can be designed with two distinct stopping criteria. The search for the weights of a specified network structure can be terminated when either the maximum spread of the simplex points is smaller than a prespecified threshold (with the centroid being selected as the optimal one in this case), or the number of iterations performed exceeds a

© 2001 by CRC Press LLC

preset threshold. Other criteria can be used to terminate the evolution of the simplex, one such criterion being when the difference in error falls below a preset threshold.

As noted earlier, the only undesirable feature of this training scheme is that as the size of the simplex (number of simplex points) increases, the computational burden correspondingly increases. This however is not unique to the present training scheme since the size of the simplex, *viz.* (N+1), depends on the size of the weight vector, which in turn is a function of the total number of interconnections in the neural net, and it is rather well known that the training complexity increases with the size of the neural network. In an attempt to reduce the training complexity, one may place arbitrary limits on the number of interconnections, which however is not attractive. Some reduction in the overall training complexity without arbitrarily limiting the network size can be achieved by partitioning the neural network into a linear and a nonlinear portion, with the nonlinear portion comprising the connections between the input nodes and the hidden nodes while the linear portion consists of the connections between the hidden nodes and the output nodes (an example of which is to have the network outputs formed as a weighted sum of the outputs of the hidden nodes). The simplex optimization is then performed to find the optimal weights in the nonlinear portion, while a linear least squares minimization is used to determine the optimal weights in the linear portion of the network.

A factor of particular significance in the use of the simplex optimization approach to neural network training is the possibility of approaching the true global minimum by a reinitialization of the simplex, as outlined earlier in the discussion of the analogy. It is well known that implementing the simplex algorithm with multiple restart operation (*i.e.*, reinitializing the simplex and executing the algorithm on the new simplex points) has global search property and hence prevents the training procedure from being trapped by local minima of the error function. Furthermore, it is argued in the literature that multiple restarts of the simplex search each time a convergence to a small cluster is attained, has the effect of moving the procedure towards finding a globally optimal solution with probability approaching 1.0. An aspect that deserves some emphasis in regard to practical implementation is that these multiple restarts can be executed in parallel, thus reducing the training time considerably. The flowchart shown in Fig. 11 summarizes the above discussion on the evolution of the simplex points.

© 2001 by CRC Press LLC

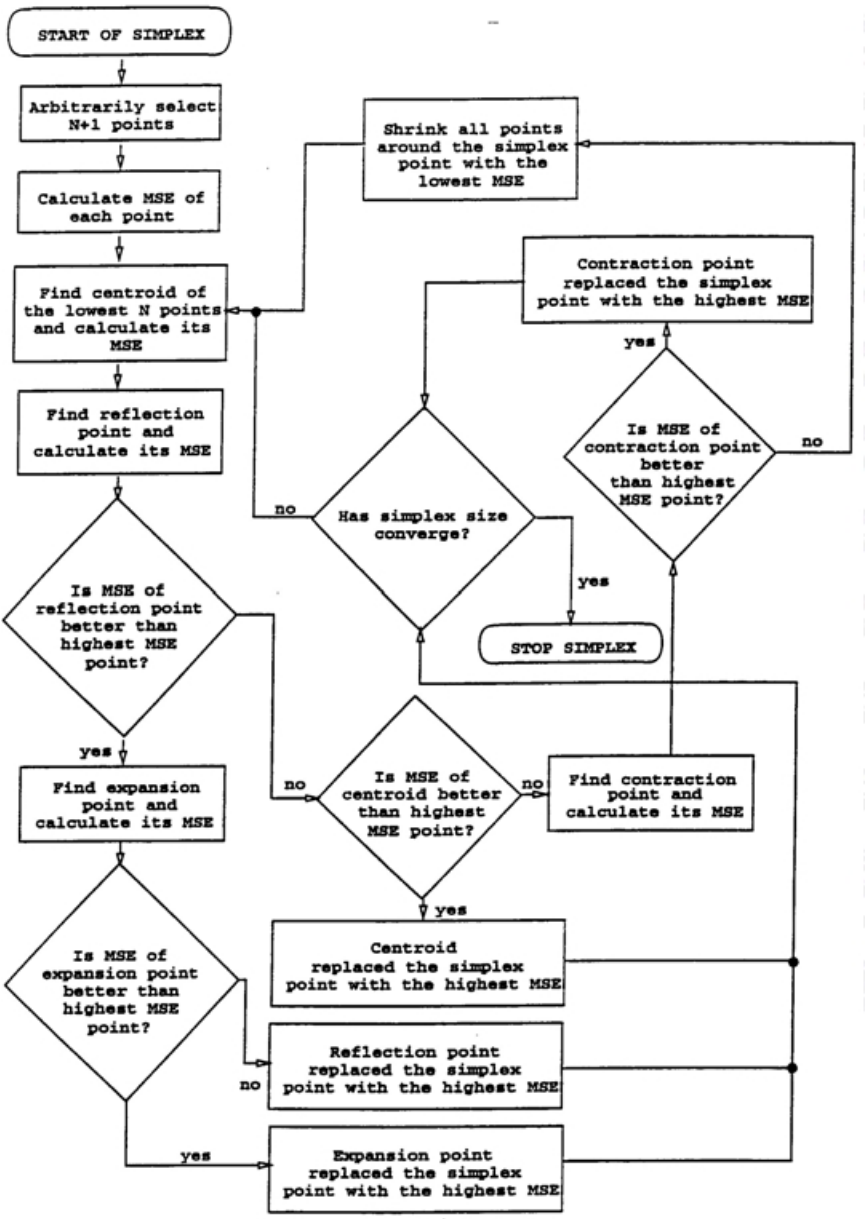

Figure 11. The evolution of the Simplex algorithm

© 2001 by CRC Press LLC

C. TRAJECTORY GENERATION PERFORMANCE

1. EXPERIMENT 1

In this experiment, the recurrent network was trained to generate a circular trajectory centered at (0.5, 0.5) in the Cartesian coordinate space. The x and y components of the trajectory can be mathematically represented as

$$x(t) = A\cos(\omega t) + b \tag{27a}$$

$$y(t) = A\sin(\omega t) + b \tag{27b}$$

where b is introduced to shift the center of the trajectory. In this experiment b in both (27a) and (27b) is set to 0.5 so as to shift trajectory to center at the point (0.5, 0.5). As discussed earlier, parameter A in Eqs. (27a) and (27b) specifies the radius of the circular trajectory and the parameter ω denotes the angular frequency. For this experiment, and also for all the other experiments in this section, A is selected to be 0.2 and ω is selected to be 0.02π.

A recurrent network with the architecture shown in Fig. 4 with five neurons was trained with the Simplex optimization algorithm to produce the desired trajectory. Two of the five neurons were arbitrarily chosen to be the outputs of the recurrent net (giving the x and y components). As with the experiments in the previous section, the recurrent network was driven only by the initial state of its neurons and hence no external input into the system is required. In this experiment, the initial states of the two output neurons were selected to be on the trajectory while the initial states of the other neurons were randomly chosen about zero according to the following distribution, $N(0, 0.00001)$ (*i.e.,* a normal or gaussian distribution with zero mean and a variance of 0.00001).

The training was conducted with teacher forcing, which was maintained until the average absolute errors of the estimates, in both the x and y components, were less than a preset value, δ (δ was chosen to be 0.06 in all of the experiments performed in this section). That is

$$\varepsilon_x = \frac{1}{K}\sum_{i=1}^{K}\left|o_x(i) - \hat{o}_x(i)\right| < \delta \tag{28a}$$

$$\varepsilon_y = \frac{1}{K}\sum_{i=1}^{K}\left|o_y(i) - \hat{o}_y(i)\right| < \delta \tag{28b}$$

© 2001 by CRC Press LLC

where ε_x and ε_y are the average errors of the x and y components respectively, o_x and o_y are the desired outputs, $\hat{o}_x$ and $\hat{o}_y$ are the network estimated outputs, and K is the length of the training vector presented to the recurrent net. Recall that teacher forcing learning is the process of feeding back, through the output recurrent connections, the desired outputs instead of the actual network outputs. In this manner, the network was trained for one complete cycle of the trajectory. After the training was completed, the network was tested for its ability to produce a stable circular trajectory, given any initial states, when the actual network outputs were fed back.

Interestingly enough, even without removing the teacher forcing during training, the network was able to produce a very stable and roughly circular trajectory. In fact, the recurrent net was run continuously for about 100 cycles and it was found that after some brief period of transient response, the network converged to a single trajectory with only very slight deviations. With this satisfying result, the network was retrained, this time with the teacher forcing slowly removed according to the following equation

$$R_{oj}(i) = \chi A_{oj}(i) + (1-\chi) D_{oj}(i) \tag{29}$$

where $j = x, y$ and $i = 1, 2, \ldots, K$ with A_{oj} and D_{oj} denoting the actual output and desired output, respectively. The parameter χ was incremented from 0 to 1 with a step size of 0.1 each time the average absolute error for each of the components for a specific value of χ reduced to less than 0.06, thus progressively reducing the teacher forcing term.

Figs. 12a-c show the results of one retraining experiment. One may observe that the network converges to a single trajectory that is approximately circular. The generated trajectory can be made more accurate by enforcing a more stringent requirement on the absolute errors before terminating the training (for instance, by requiring the average absolute errors to be less than 0.01).

Of particular interest in this experiment is the sensitivity of the trained neural network to the initial state of neurons. To test this feature, several different simulations were conducted. First, the initial states of the output neurons were set to a point on the trajectory, while the remaining neurons were started at an initial state that is normally distributed around zero according to $N(0, 0.00001)$ (*i.e.,* similar to the initial state conditions of the recurrent net used during training). In all of the simulations conducted with this setup, the network converged to a single circular trajectory after a brief transient period. This is to be expected since this is the manner in which the network was trained. A more challenging scenario would be to set the initial states of all the neurons, including the output neurons, to a value normally distributed around zero with variance 0.00001. The results of several simulations conducted with this set-up also demonstrate that the network converges to a single circular trajectory in all the cases. To further challenge the stability of the recurrent net, several

© 2001 by CRC Press LLC

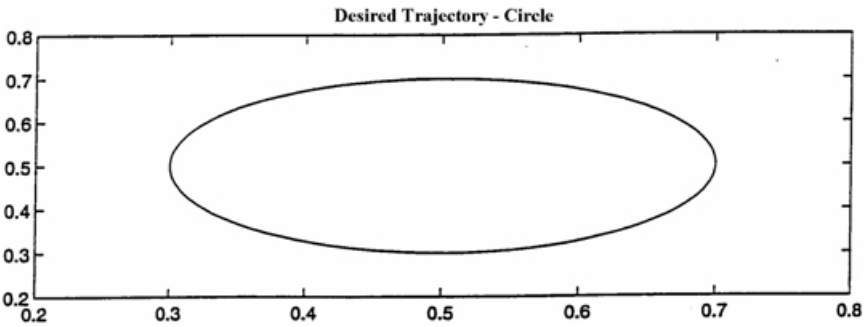

Figure 12a. Desired trajectory of Experiment

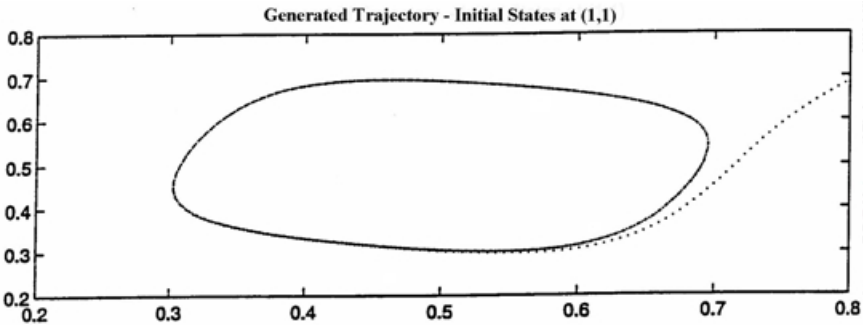

Figure 12b. Generated trajectory of a simplex trained recurrent network (Experiment 1)

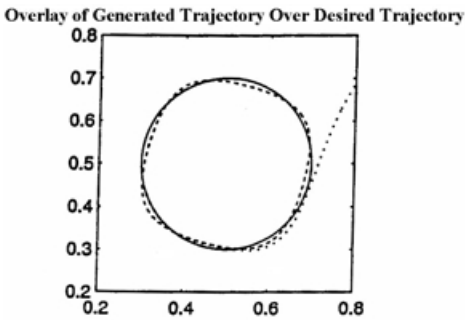

Figure 12c. Overlaid trajectories – desired and generated trajectory (Experiment 1)

simulations with the initial state of all the neurons set randomly according to $N(100,5)$ were conducted. Again, all the results obtained demonstrate convergence of the network to a single circular trajectory. One example of the various simulations is shown in Figs. 12a-c and 13a-b. Figs. 13a and b show the outputs of the two output neurons which demonstrate that the recurrent network has indeed captured the oscillating behavior required to generate the desired circular trajectory. Figs. 12a-c show the desired trajectory, the trajectory produced by the network, and both trajectories overlaid, which confirm that the

© 2001 by CRC Press LLC

network has indeed been trained by the Simplex optimization algorithm. The convergence of the recurrent network to produce the same attractor trajectory in all the simulations regardless of the initial states of the neurons illustrates the robustness of the trained network.

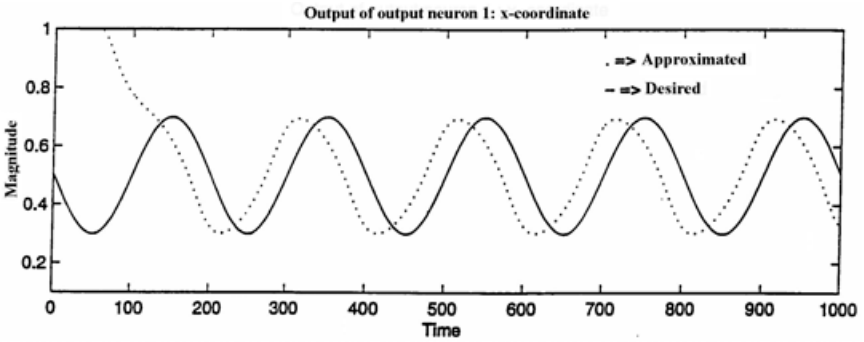

Figure 13a. Desired and generated x-coordinate output of a trained recurrent network (Experiment 1)

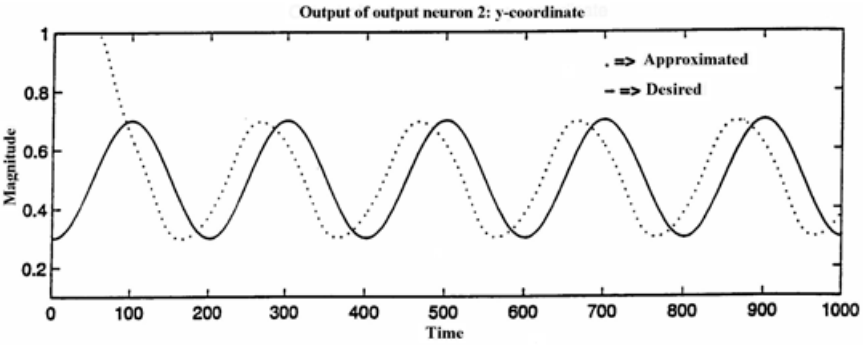

Figure 13b. Desired and generated y-coordinate output of a trained recurrent network (Experiment 1)

2. EXPERIMENT 2

Although convergence of the recurrent net to the desired trajectory was obtained in all of simulations conducted in Experiment 1, the transient response of the network does not appear to be as smooth and as controllable as desired. A more desirable response of the network is shown in Fig. 14a. As noted, the trajectory of the desired response starts at the center of the trajectory. The trajectory then slowly and smoothly diverges from the center and converges onto the circular attractor pattern. This smoother, more controllable and predictable response is particularly important in control applications where a smooth and predictable transient response is critical to the operation of the control system.

For this experiment, the same five-neuron recurrent net was utilized. The recurrent net was again trained for one complete cycle of the trajectory. The

© 2001 by CRC Press LLC

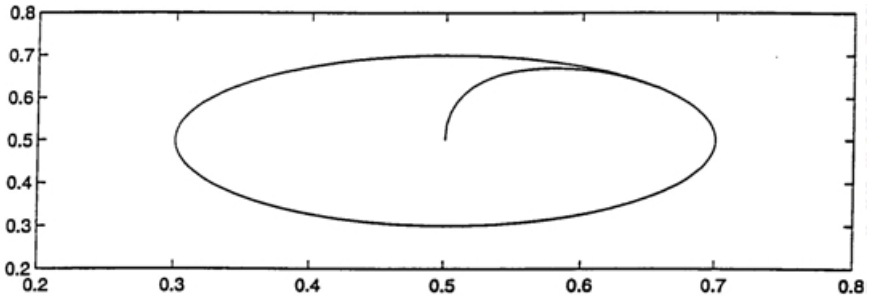

Figure 14a. Desired trajectory of Experiment 2

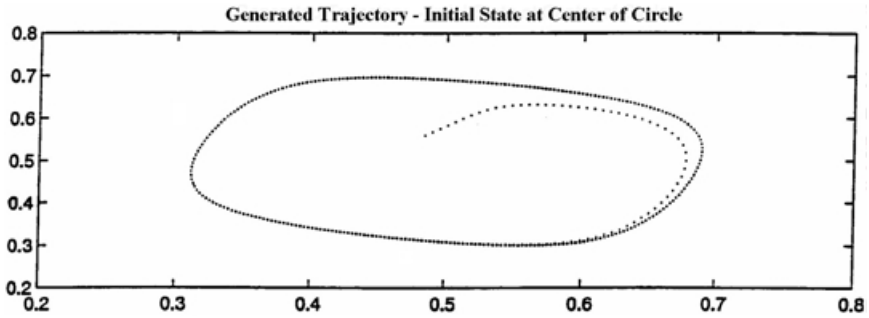

Figure 14b. Generated trajectory of a simplex trained recurrent network (Experiment 2)

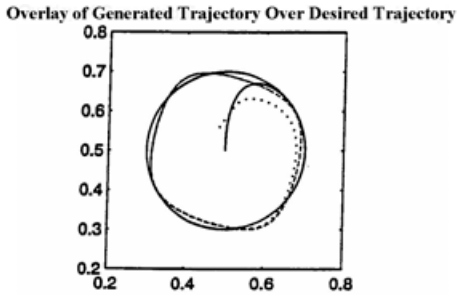

Figure 14c. Overlaid trajectories – desired and generated trajectory (Experiment 2)

equations governing the trajectory were modified slightly to accommodate the smoother and more predictable transient response given by

$$x(t) = (1 - e^{-\partial t})A\cos(\omega t) + b \tag{30a}$$

$$y(t) = (1 - e^{-\partial t})A\sin(\omega t) + b \tag{30b}$$

(as in Experiment 1, b is set to 0.5). Note that Eqs. (30a) and (30b) differ from (27a) and (27b) with the introduction of the exponential term, $(1 - e^{-\partial t})$. The exponential term is introduced to control the growth of the trajectory from its initial point. The parameter ∂ is preset to achieve the desired trajectory growth.

© 2001 by CRC Press LLC

The training was commenced with teacher forcing learning. The initial states of the output neurons were normally distributed according to $N(0.5, 0.00001)$, while the initial states of the rest of the neurons were selected according to $N(0, 0.00001)$. The training was stopped when the same error criterion as used in Experiment 1 was met (*i.e.*, the average absolute errors in (28a) and (28b) were satisfied with δ chosen to be 0.06). After the training was completed in this manner, several validating simulations were conducted to investigate the response of the network for arbitrary initial states. In the various experiments conducted with the initial states of the network selected according to the specific distributions used in the training process, the network converged to a single circular trajectory in all instances with a much smoother and more predictable transient response. A representative trajectory generated by the network is shown in Fig. 14b. Clearly the transient response of the generated trajectory is similar to the desired transient response illustrated in Fig. 14a (Fig. 14c shows an overlay of the desired trajectory and the trajectory generated by the network). Figs. 15a and b indicate that the network has indeed captured the oscillating behavior required for trajectory generation. An important outcome from this set of experiments is the demonstration that the network can be trained to produce a desired trajectory with specific transient response. As noted before, this characteristic can be exploited in designing control systems with specified trajectory paths.

Also of interest in this experiment was the investigation of how the network would respond when the network was started at initial states different from the one used during training. To test this feature, several simulations were conducted with the initial states of the network selected randomly outside the area enclosed by the trajectory. It is interesting to note that the network failed to converge to the desired trajectory in all instances. Hence it seems that in training the network to produce a smoother and more predictable transient response, the robustness of the network demonstrated in Experiment 1 is lost. In other words, the network trained in this manner is sensitive to the initial states of its neurons. It may be noted, in conclusion, that a more desirable result, *i.e.,* a smoother and more circular trajectory, can be achieved by retraining the network with the teacher forcing term slowly removed according to (29), and enforcing a more stringent stopping requirement (*i.e.,* by requiring the average absolute errors to be less than 0.01 for example).

© 2001 by CRC Press LLC

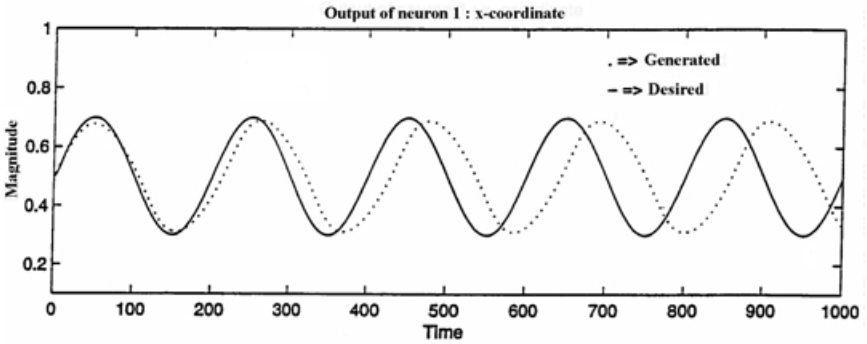

Figure 15a. Desired and generated x-coordinate output of a trained recurrent network (Experiment 2)

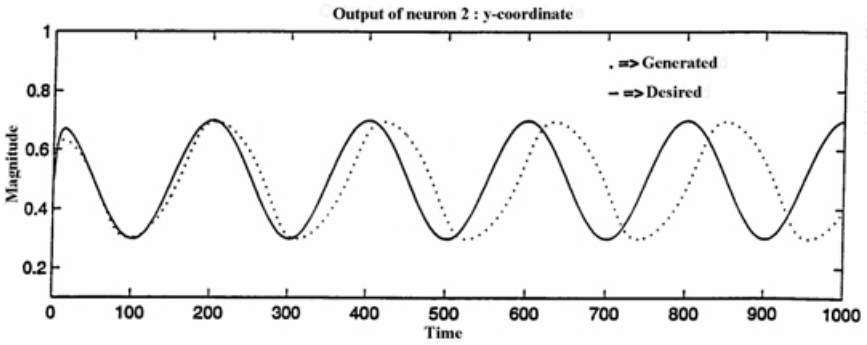

Figure 15b. Desired and generated y-coordinate output of a trained recurrent network (Experiment 2)

3. EXPERIMENT 3

To further examine the optimization prowess of the Simplex algorithm, an attempt is made to train a recurrent network to generate an even more complex trajectory - the figure-eight pattern. As noted earlier, the figure eight trajectory can be produced by requiring the neural network outputs to converge to the periodic signals

$$x(t) = A\sin(\omega t) + b \tag{31a}$$

$$y(t) = A\sin(2\omega t) + b\,. \tag{31b}$$

The parameters A, b, and ω were selected to be the same as in the earlier experiments. All the training conditions, including the selection of the initial states, were maintained similar to Experiment 1. Initially a five-neuron network was utilized for this purpose. However, with the selected structure, the network training, with teacher forcing learning implemented, failed to converge to the desired trajectory. Hence, the network size was increased from five neurons to ten neurons. The training was repeated and this time, after a longer period of training than that required for the simpler trajectory - circular pattern, the

© 2001 by CRC Press LLC

network converged to the desired trajectory. The results of this experiment are shown in Figs. 16a-b and 17a-d. These results indicate that the network has indeed been trained to generate autonomously the desired trajectory. It should however be mentioned that unlike in the earlier experiments, the network does not demonstrate convergence to a single trajectory. Instead it converges to a series of trajectories.

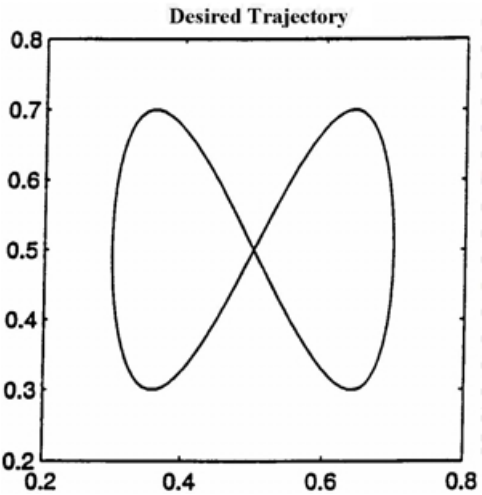

Figure 16a. Desired trajectory of Experiment 3

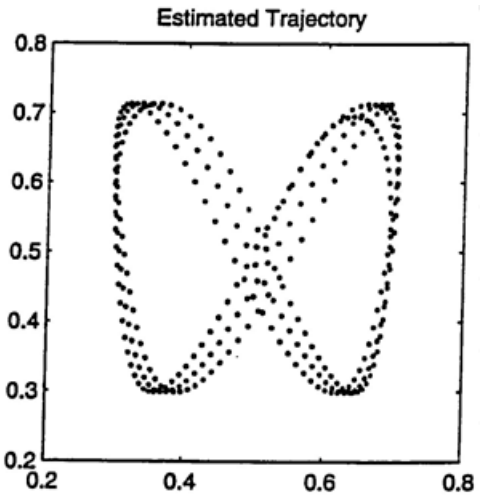

Figure 16b. Generated trajectory of a simplex trained recurrent network (Experiment 3)

© 2001 by CRC Press LLC

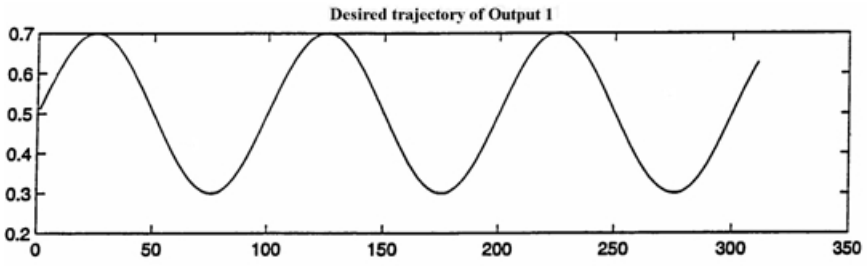

Figure 17a. Desired x-coordinate output of Experiment 3

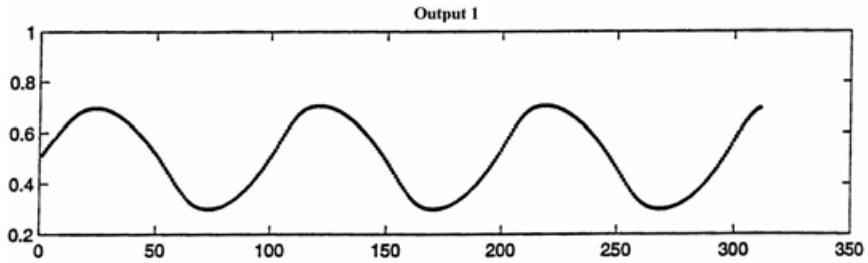

Figure 17b. Generated x-coordinate output of a trained recurrent network (Experiment 3)

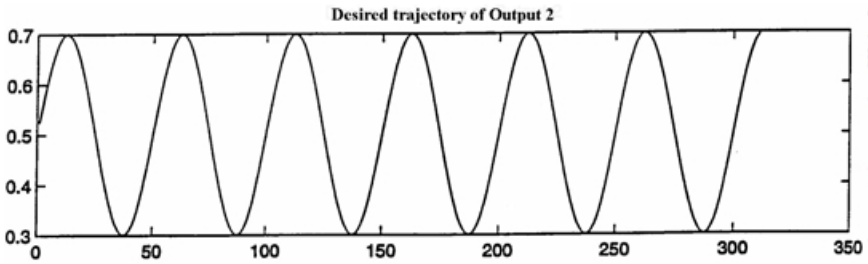

Figure 17c. Desired y-coordinate output of Experiment 3

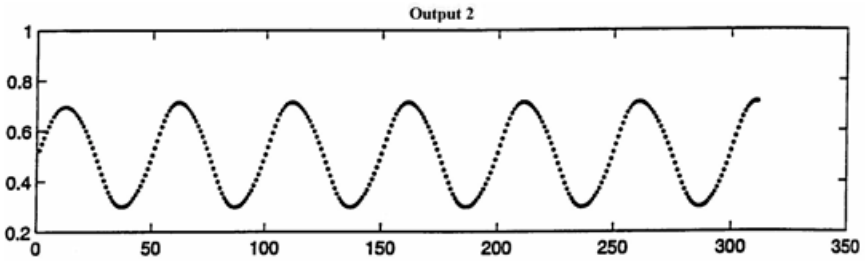

Figure 17d. Generated y-coordinate output of a trained recurrent network (Experiment 3)

© 2001 by CRC Press LLC

V. CONCLUSIONS

Two distinct methods for training recurrent neural networks that eliminate the need for the computation of error gradients were presented in this article. Since gradient computation constitutes the major part of the overall training complexity in the use of gradient-based methods such as backpropagation learning, the methods discussed in this article provide attractive alternatives to the training of neural networks in general and recurrent networks in particular. One of these methods based on the theory of learning automata utilizes the concepts of reinforcement learning and employs use of penalty-reward methods for tailoring specific training policies. The other method utilizes the nonlinear simplex optimization approach and provides a systematic procedure for the adjustment of neural network parameters. The training performance resulting from the two approaches was demonstrated by application to a complex spatiotemporal learning problem of designing a dynamical neural network that outputs a prescribed attractor trajectory pattern. Simulation experiments conducted here with specific benchmark trajectory patterns confirm the efficacy of the learning automata approach and the simplex optimization approach for a simple and efficient training of recurrent nets.

REFERENCES

Almeida, L. B. A learning rule for asynchronous perceptrons with feedback in a combinatorial environment, *Proc. of the IEEE 1st Annual Intl. Conf. on Neural Networks*, 609, San Diego, 1987.

Bartle, R. and Sherbet, D., *Introduction to Real Analysis*. Wiley: New York, 1992.

Behrens, H., Gawronska, D., Hollatz, J. and Schurmann, B., Recurrent and feedforward backpropagation for time-independent pattern recognition, in *Proc. 1991 Intl. Joint Conf. on Neural Networks (IJCNN)*, Seattle, July 1991.

Bertsekas, D. P., *Dynamic Programming,* Prentice-Hall: Englewood Cliffs, NJ, 1987.

Condarcure, T., *A learning automaton approach to trajectory learning and control system design using dynamic recurrent neural networks*, M.S. Thesis, ECE Department, The University of Arizona, 1991.

Duan, Q., Gupta, H. V., and Sorooshian, S., Effective and efficient global optimization for conceptual rainfall-runoff models, *Water Resources Research*, 28, 1015, 1992.

© 2001 by CRC Press LLC

Karakasoglu, A., Sudharsanan, S. I., and Sundareshan, M. K., Identification and decentralized adaptive control using dynamical neural networks with application to robotic manipulators, *IEEE Trans. on Neural Networks*, 4, 919, 1993.

Khalil, H. K., *Nonlinear Systems,* Macmillan: New York, 1992.

Lakshmivarahan, S., *Learning Algorithms: Theory and Applications*. New York: Springer-Verlag, 1981.

Lapedes, A. and Farber, R., Programming a massively parallel computation universal system: static behavior, in *Neural Networks for Computing,* Denker, J. S., Ed., AIP Conference Proceedings, 151, 283, 1986.

Lewis, T. and Payne, W. H., Generalized feedback shift register pseudorandom number algorithm, *Journal of the Association for Computing Machinery*, 20 (3), 456, 1973.

Lin, D. T., Dayhoff, J. E., and Ligomenides, P. A., Trajectory production with the adaptive time-delay neural network, *Neural Networks*, 8, 447, 1995.

Mendel, J. M. and Fu, K. S., Eds., *Adaptive, Learning and Pattern Recognition Systems,* Academic: New York, 1970.

Narendra, K. S. and Thathachan, M. A. L. *Learning Automata, an Introduction*, Addison Wesley, Reading, MA, 1989.

Nelder, A. J. and Mead, R., A simplex method for function minization, *Comput. Journal*, 7, 308, 1965.

Nilsson, N. J., *Learning Machines: Foundations of Trainable Pattern Classifying Systems,* McGraw-Hill: New York, 1965.

Pearlmutter, B. Learning state space trajectories in recurrent neural networks, *Neural Computation,* 1, 263, 1989.

Pearlmutter, B. Gradient calculations for dynamic recurrent neural networks: a survey, *IEEE Trans. on Neural Networks,* 6, 1212, 1995.

Pineda, F. J. Generalization of backpropagation in recurrent neural networks, *Physical Review Letters*, 59 (19), 2229, 1987.

Ruiz, A., Owens, D. H., and Townley, S., Existence, learning, and replication of periodic motions in recurrent neural networks, *IEEE Trans. on Neural Networks*, 9, 651, 1998.

© 2001 by CRC Press LLC

Rumelhart, D. E., Hinton, G. E., and Williams, R. J., Learning internal representations by error propagation, in *Parallel Distributed Processing: Explorations in the Microstructure of Cognition*, Rumelhart, D. E. and McClelland, J. L., Eds., MIT Press: Cambridge, 45, 1986.

Sato, M. A real time running algorithm for recurrent neural networks, *Biological Cybernetics,* 62, 237, 1990.

Sudharsanan, S. I. and Sundareshan, M. K., Training of a three layer recurrent neural network for nonlinear input-output mapping, *Proc. 1991 Intl. Joint Conf. on Neural Networks* (IJCNN-91), Seattle, 1991.

Sudharsanan, S. I. and Sundareshan, M. K., Supervised training of dynamical neural networks for associative memory design and identification of nonlinear maps, *Intl. J. of Neural Systems*, 5, 165, September 1994.

Sudharsanan, S. I. and Sundareshan, M. K., Equilibrium characterization of dynamical neural networks and a systematic synthesis procedure for associative memories, *IEEE Trans. on Neural Networks*, 2, 509, September 1991a.

Sudharsanan, S. I. and Sundareshan, M. K., Exponential stability and a systematic synthesis of a neural network for quadratic minimization, *Neural Networks*, 4, 599, 1991b.

Toomarian, N. and Barhen, J., Learning a trajectory using adjoint functions and teacher forcing, *Neural Networks*, 5, 473, 1992.

Varshavskii, V. I. and Vorontsova, I. P., On the behavior of stochastic automata with variable structure, *Automat. Remote Contr.*, 24, 327, 1963.

Werbos, P., Backpropagation through time: what it does and how to do it, *Proc. of the IEEE*, 78, 1550, 1990.

Williams, R. and Zipser, D., A learning algorithm for continually running fully recurrent neural networks, *Neural Computation*, 1, 270, 1989.

Wong, Y. C. and Sundareshan, M. K., A simplex trained neural network architecture for sensor fusion and tracking of target maneuvers, *Kybernetika,* No. 4-5, 1999.

© 2001 by CRC Press LLC