Chapter 10

COMPARISON OF RECURRENT NEURAL NETWORKS FOR TRAJECTORY GENERATION

David G. Hagner

Product Development Division
Ford Motor Company

Mohamad H. Hassoun

Department of Electrical & Computer Engineering
Wayne State University

Paul B. Watta

Department of. Electrical & Computer Engineering
University of Michigan-Dearborn

I. INTRODUCTION

Recurrent neural networks are universal approximators of dynamic systems and hence can be used to model the behavior of a wide range of practical systems which can be described by ordinary differential equations [Funashi and Nakamura, 1993]. The ability to model such systems is an important task for nonlinear control systems design, system identification, and testing.

An interesting feature of recurrent neural networks is their ability to “learn” a trajectory from training data. Under certain conditions, these networks can also *generalize* [Hassoun, 1995; Hagner, 1999] from the training data to produce smooth and consistent dynamic behavior for entirely new inputs or new regions of state space (i.e., inputs or regions of state space not encountered during training).

In this chapter, we discuss the use of single-and multilayer recurrent neural networks for the approximation of two famous 2-dimensional limit cycles: the circle and the figure-eight. We will give a qualitative and quantitative analysis of the neural net approximations of these autonomous systems for various network architectures (internally and externally recurrent), learning rules (incremental and conjugate gradient descent, and three variations of the extended Kalman filter), and initial conditions (previous states on the trajectory and previous states set near the origin).

A variety of approaches and architectures has been proposed in the literature for approximating such trajectories, including discrete-time, feedforward networks with tapped delay line external recurrence [Tsung and Cottrell, 1995]; discrete-time, feedforward networks with adaptable time delays [Lin, Dayhoff, and Ligomenides, 1995]; discrete-time, single-layer, recurrent networks with adaptable time constants [Sundareshan and Condarcure, 1998]; continuous time, single-layer recurrent networks with adaptable time constants [Toomarian and Barhen, 1992; Pearlmutter, 1995]; and continuous time, single-layer recurrent networks with adaptable time constants and adaptable time delays [Cohen, Saad, and Marom, 1997].

Previous studies of two-dimensional limit cycle trajectories have involved simulations of one architecture and one learning algorithm, and used only one set of initial conditions. Several studies also investigated only the circle trajectory, which proved to be relatively easy for any architecture/algorithm combination to learn, and thus does not provide a test able to delineate the differences in performance. Recurrent network architectures have been experimentally compared [Horne and Giles, 1995], though not on autonomous network applications like those considered here. Additionally, recurrent network learning algorithms have been compared [Logar, Corwin, and Oldham, 1993; Williams and Zipser, 1995], though with a focus on algorithm speed, and not on architecture and performance (defined here as the ability of the network to accurately match the desired training data).

The remainder of this chapter is organized as follows. Section 2 reviews the structure of recurrent neural network architectures and gives definitions of internal and external recurrence. Section 3 discusses how the training sets for the circle and the figure-eight are generated. Section 4 presents the quantitative error measures and performance metrics which are used to assess the quality of the network dynamics during the playback phase. Section 5 briefly reviews the five training algorithms which are simulated. Section 6 describes the simulations performed in this work and the results of these simulations, and provides comparative analyses of network architecture and training algorithm performances and properties. Section 7 presents the conclusions reached concerning the capabilities and limitations of the network architectures and training algorithms when applied to learning the limit cycle trajectories, and a discussion of possible future extensions of this work.

II. ARCHITECTURE

Feedforward neural networks can model static mappings, but do not have the capability to generate dynamic behavior. By adding recurrent connections, though, a feedforward network can be transformed into a recurrent network which can be used to model dynamic systems. For the recurrent networks described here, we will start with a multilayer feedforward neural net, where neurons are grouped into layers and layers are cascaded one after the next. We will assume full interconnectivity between layers, but each layer will be connected to the layer which immediately follows. For example, in a 3-layer network, layer 1 will be fully connected to layer 2, and layer 2 fully connected to layer 3, but no direct feedforward connections will be present between layer 1 and layer 3.

Once the feedforward structure of the network is fixed, recurrent connections can be added by using two main types of recurrence: internal and external. *Internal recurrence* is defined here as the connection of outputs of units of a given layer to the inputs of units in that same layer. *External recurrence* is defined as the connection of outputs of the final (output) layer of a network to the inputs of units in the first (input) layer. This type of network that has both feedforward and recurrent layers has been termed a recurrent multilayer perceptron (RMLP) network [Puskorius and Feldkamp, 1994], and combines the instantaneous mapping capabilities of multilayer feedforward networks (often referred to as multilayer perceptron, or MLP, networks) with the system state memory, or dynamics, of recurrent networks.

Each unit in the recurrent network has inputs from other units, as well as a single output to other units (and possibly the external environment). The output y of a unit at time step n+1 is given by its describing function

$$y(n+1) = f\left[\sum_{j=1}^{J} x_j(n) w_j(n)\right]$$

Here, $x_1(n)$, $x_2(n)$, . . ., $x_J(n)$ are the inputs to the neuron at (discrete) time step n. Note that in general, the total input vector $\mathbf{x}$ is composed of outputs of other units, bias inputs, and external inputs, though for the autonomous networks considered here, there will be no external inputs. Associated with each input is a weight $w_1(n)$, $w_2(n)$, . . ., $w_J(n)$, which, during the training phase, also evolves in time. In this chapter, the discussion will focus on activation functions which are either linear: $f(x) = x$, or sigmoidal: $f(x) = \tanh(x)$.

An example of a 3-layer recurrent network is shown in Figure 1. This network has three feedforward units in layer 1, two recurrent units in layer 2, and two feedforward units in layer 3. In addition, this network has external recurrence with two unit time delays. We will use the notation $3 \times 2_R \times 2(2)$ to represent this structure. The subscript R indicates that the layer has recurrent connections (output of the layer is fed back into the input of that layer). The 2 in parenthesis at the end indicates that the network has external recurrence with two delays.

Various architectures were tested initially to determine the advantages and disadvantages of the different architecture types, and to determine which subset of the many possible architectures would be used for the final comparison analysis with the different training algorithms. The variations studied were 1) linear vs. sigmoidal unit activation functions, 2) single recurrent layer vs. hidden recurrent layers with a two-unit feedforward output layer, 3) single vs. multiple hidden feedforward layers with a two-unit feedforward output layer, 4) recurrent layer networks with and without external recurrence, and 5) up to five unit delays used for external recurrence.

During initial network simulation analysis, it was found that if a network contained a single layer, the units required sigmoidal activation functions to learn the trajectories, and if the network employed an output layer with feedforward linear units, the hidden layer (recurrent or feedforward) similarly required sigmoidal units. This is as expected for this application of learning nonlinear trajectories where a linear combination of unit values is not sufficient.

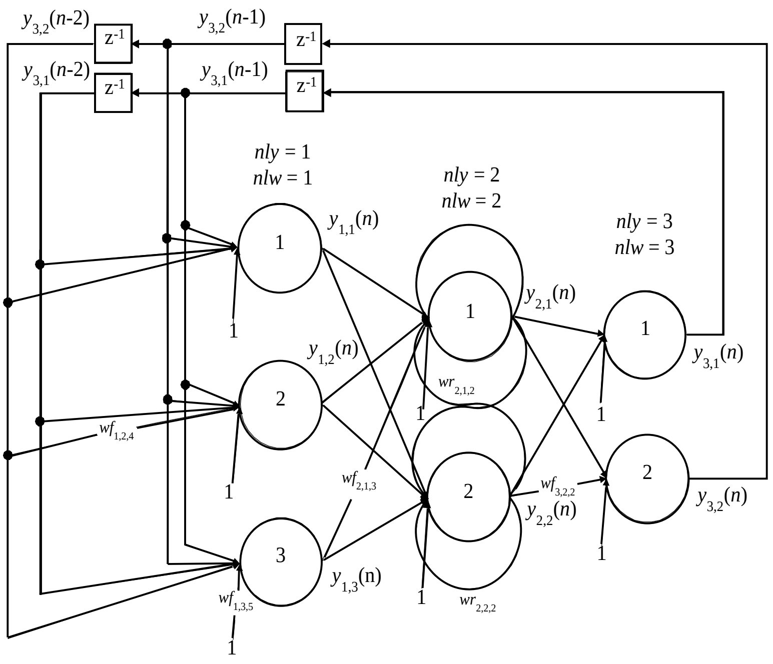

Figure 1. A 3-layer $3 \times 2_R \times 2(2)$ network with three units in layer 1, two recurrent units in layer 2, and 2 units in layer 3. This network has external feedback with 2 unit delays.

For a feedforward output layer with or without external recurrence, only two units are required, as any additional units' outputs would not be connected. For a feedforward output layer, linear units provided faster convergence, but the solutions exhibited inferior off-trajectory performance compared to feedforward output layers with sigmoidal units. These trajectories were similar to the *center*, or *vortex* trajectories generated by two-dimensional linear systems with a purely imaginary conjugate pair of eigenvalues (for a description of phase-plane analysis of linear and non-linear systems, see Van De Vegte [1986] and Dickinson [1991], indicating that the network was not exploiting the nonlinearities of the hidden units). Additionally, feedforward output layers with sigmoidal units were more robust during training, whereas linear unit learning often diverged during training. Thus sigmoidal activation functions were used for all units in both hidden and output layers.

It was found that single recurrent layer networks performed well, and the addition of hidden recurrent layers did not provide any noticeable benefits. Coupling these findings with the fact that the addition of external recurrence to a single layer recurrent network would be redundant, the only recurrent unit architecture to be tested in the final analysis was a single layer of recurrent units with sigmoidal activation functions.

Externally recurrent networks with one hidden layer generally performed as well as networks with multiple hidden layers. Additionally, externally recurrent networks with a recurrent hidden layer provided no noticeable benefit over networks with a single recurrent layer, or compared to externally recurrent networks with a single feedforward hidden layer.

As indicated above, the only two architectures that both provided good performance and were also different enough to warrant further comparison were the single layer recurrent n_R and single hidden layer feedforward with external recurrence $n \times L(D)$.

Initial experimentation with the number of units and the number of delays determined that the minimum network sizes for the circle trajectory were 2_R and $2 \times 2(1)$ for internal and external recurrence, respectively. One larger network (for each architecture) was then chosen to provide a significant increase in the number of parameters (weights), without increasing the network size such that it became computationally prohibitive. These network sizes were 4_R and $4 \times 2(1)$. Thus the total number of architecture/algorithm combinations to be compared for the circle trajectory was 4 networks x 5 algorithms =20.

Similar experimentation for the figure-eight trajectory led to the determination that the minimum network sizes were 4_R and $4 \times 2(4)$ for internal and external recurrence, respectively. Two additional, larger networks (for each architecture) were chosen for the figure-eight trajectory to be 6_R, 8_R, $6 \times 2(4)$, and $8 \times 2(4)$. Thus the total number of architecture/algorithm combinations to be compared for the figure-eight trajectory was 6 networks x 5 algorithms = 30.

III. TRAINING SET

The training set for the circle and the figure-eight consists of 2-dimensional samples of the trajectory, as shown in Figures 2a and b.

For both trajectories, M = 100 samples are used because this value offered a balance between a smaller M that provided more distinction between data points (beneficial because incremental training algorithms tend to optimize for the current region if that region presents little new information) and a larger M that provided a smoother, more accurate representation of the continuous-time trajectory.

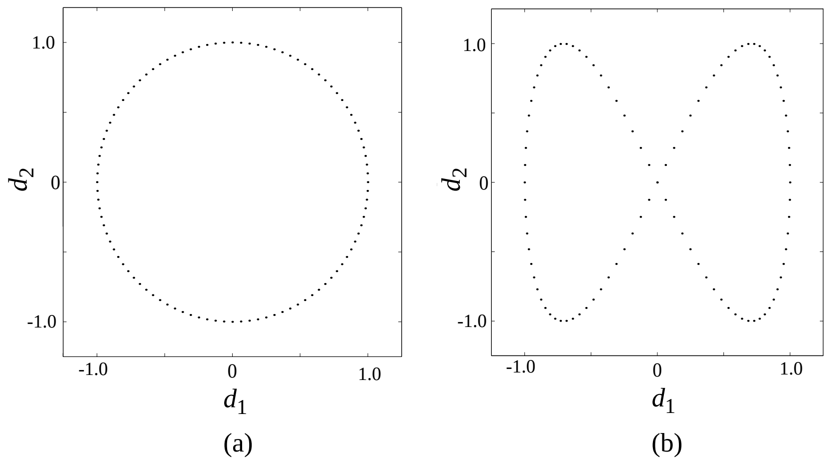

Figure 2. (a) The circle and (b) figure-eight training sets; both contain $M = 100$ samples.

In the case of the circle, the target vector ***d*** at time step *n* is given by

$$\mathbf{d}(n) = \begin{bmatrix} d_1(n) & d_2(n) \end{bmatrix} = \begin{bmatrix} \sin\left(\frac{2\pi n}{100} + \frac{\pi}{2}\right) & \sin\left(\frac{2\pi n}{100}\right) \end{bmatrix}$$

and for the figure-eight trajectory, the target vector is given by

$$\mathbf{d}(n) = \begin{bmatrix} d_1(n) & d_2(n) \end{bmatrix} = \begin{bmatrix} \sin\left(\frac{2\pi n}{100} + \frac{\pi}{2}\right) & \sin\left(\frac{4\pi n}{100}\right) \end{bmatrix}$$

To train the network to learn the limit cycle trajectories, the target values, ***d***(*n*), were taken as the coordinates of the subsequent point on the trajectory, and the target value for the final, *M*th point was the first point, to train the network to oscillate around the trajectory.

IV. ERROR FUNCTION AND PERFORMANCE METRIC

One way to assess network performance is by formulating an error function which measures the difference between the neural net approximation and the desired trajectory. A common measure is the standard sum of squared errors defined by

$$E(\mathbf{w}) = \frac{1}{2}\sum_{n=1}^{M}\sum_{l=1}^{L}[d_l(n) - y_l(n)]^2 \qquad (1)$$

where $d_l(n)$ is the target value and $y_l(n)$ is the actual output of output unit l at time step n. The error function given here is called the *batch error,* because it contains the sum of the errors over the entire training set; that is, the error over all M time steps. Some training algorithms make use of this error function, and others make use of the *instantaneous error,*

$$E(\mathbf{w}, n) = \frac{1}{2}\sum_{l=1}^{L}[d_l(n) - y_l(n)]^2 \tag{2}$$

where the errors are summed only over the L output units, and not over time, and thus may be written as $E(\boldsymbol{w}, n)$. Both definitions of the error function will be used in the discussion of training algorithms in the next section, as this work considers both incremental and batch training algorithms.

In training mode, as the network output is computed for each step and the error vector calculated, a technique called *teacher forcing* may be employed, which substitutes the previous target values for the past network output values after the computation of the error and prior to computing the next step. This has been shown to be an effective technique for maintaining training algorithm stability [Puskorius and Feldkamp, 1994; Williams, 1992; Hagner, 1999]. In full teacher forcing, the previous target value is substituted; for partial teacher forcing, a weighted sum of the previous target value and the network output value is used [Hagner, 1999].

Besides the use of error functions, network performance may also be qualitatively assessed by visual comparison of the network trajectory to the target trajectory. The network trajectory is generated in the *recall,* or *playback* mode, after the *training* mode is finished. In playback mode, a set of initial conditions is provided, and the network output is computed for the first time step. These results are then used to generate the network output for the second time step, etc., until the desired number of steps is taken. Because it is desired that the trajectory be a limit cycle trajectory (i.e., once the network output approaches the target trajectory, it remains on that trajectory) and there may be transient effects due to initial conditions, the network will be run through 10M steps, with all steps plotted, providing both the transient and steady-state portions of the trajectory.

A network's performance may be measured quantitatively by the error function and qualitatively by visual inspection of the network trajectory. These two measures often do not correlate well, as the error function calculated using teacher forcing may be quite different from the error function calculated during playback (which represents a network’s true performance), when teacher forcing is not used. Additionally, the trajectory generated during playback may become unstable after a certain number of steps, indicating that the network has not generated a true limit cycle, and thus a measurement of trajectory stability is also desirable.

A network's performance generally improves as the error decreases during training, but the relationship is often not smooth, as shown in Figure 3, and may vary by large amounts in only a few training cycles. For example, during a training run on the figure-eight trajectory, the value of the training error (calculated with teacher forcing) might decrease rapidly while the network trajectory is confined near a single point, and thus the trajectory error (calculated without teacher forcing) is large. As training continues and the training error decreases more slowly as a minimum is approached, a sudden improvement in performance may be seen as the network begins oscillation, or the trajectory changes abruptly from an elliptical oscillation to a figure-eight. Sometimes the error decreases quickly during these performance improvements, as the algorithm leaves a local minimum or a long, flat valley, and sometimes it changes little. At other points in training, a network's performance may be fairly good for a period of time (the algorithm may be in a plateau), and then may deteriorate rapidly as the algorithm enters a new region of weight space, even though the training error decreases continuously.

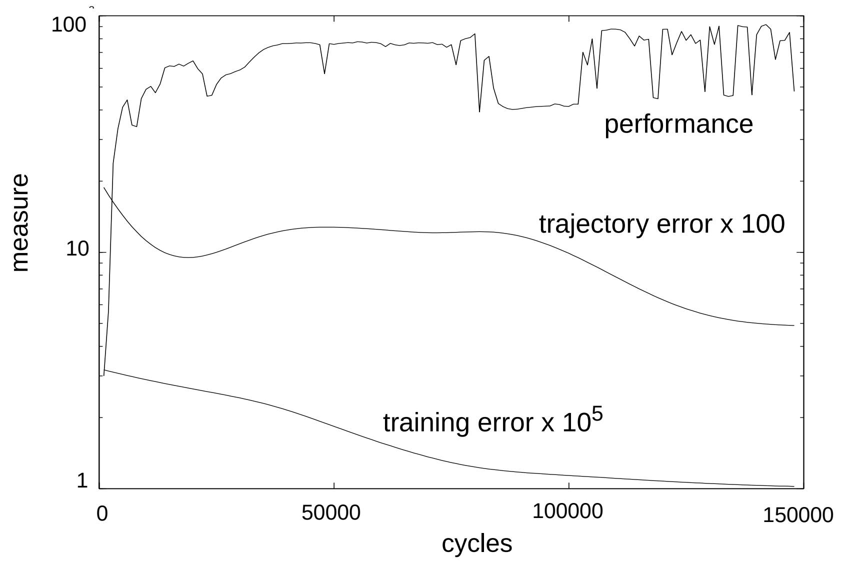

Figure 3. Performance and error vs. training cycles for an $8 \times 2(4)$ network trained with incremental gradient descent to learn the figure-eight trajectory.

Figure 3 shows the dependence of training error, trajectory error (with initial conditions on the trajectory), and performance (quantitatively given below in Equation 3) for one simulation of the incremental gradient descent algorithm on the figure-eight trajectory. Note the large variation in the performance metric for smoothly decreasing training and trajectory errors after 100,000 cycles, and also the variation in the trajectory error for smoothly decreasing training error around 25,000 cycles.

The trajectory without teacher forcing, made during playback, may be quantitatively measured. The straightforward method of calculating the value of the error function during playback provides a measure of the performance, but it is calculated over the first "loop" of *M* steps, and thus may contain transient characteristics of the playback, and may not represent the quality of the steady-state trajectory. Thus a measure of the error function is needed for the steady-state trajectory (as stated previously, 10 "loops" were sufficient for the networks to reach steady-state conditions, and thus the performance metric is defined over the last loop, or last *M* steps of the playback trajectory). The errors during the transients typically cause the trajectory to "fall behind" the target trajectory, because the network trajectory typically starts at a point off the target trajectory and eventually is attracted to it, though when this happens the points are not synchronized. A network trajectory that is on the target trajectory but unsynchronized, or out of phase with respect to the target trajectory, is defined here to be perfectly good; the network has learned the limit cycle trajectory, and because there is no external input to serve as a clock signal, the fact that the trajectories overlap is sufficient.

A measure of the amount of trajectory "overlap" is thus required; the standard error function will not provide a relevant result, because it relies on the synchronization of the trajectory points and would provide a poor result for an exact trajectory match that is out of phase. The measure of trajectory overlap is accomplished by performing a convolution of the last *M* steps of the network trajectory and the target trajectory. The value of the error function is calculated for the target trajectory and the network trajectory *M* times, where for each instance the starting point of the network trajectory is shifted 1 step. The minimal value of the convolution occurs for the shifted trajectory that provides the minimum error; this value is used as the measure of overlap. The error minimum obtained from the convolution is then used to calculate a performance measure, on the scale from 0 to 100, where 100 indicates an almost perfect overlap of trajectories.

The steady-state performance measure is given by

$$P_{ss}(\mathbf{w}) = \frac{1}{2}[100e^{-\alpha_p E_{ss1}(\mathbf{w})} + 100e^{-\alpha_p E_{ss2}(\mathbf{w})}] \tag{3}$$

where α_p is a constant chosen for the specific figure or trajectory such that visually "fair" steady-state trajectories achieved performance measures between 80 and 90, "very good" trajectories achieved performance measures between 90 and 95, and "excellent" trajectories achieved performance measures between 95 and 100, $E_{ss1}(\mathbf{w})$ is the value of the error function that provided the minimum during the convolution procedure for the first set of initial conditions (on-trajectory), and $E_{ss2}(\mathbf{w})$ is the value of the error function that provided the minimum during the convolution procedure for the second set of initial conditions (off-trajectory). The network's overall performance measure was thus the average of the individual performance measures for the two steady-state trajectories resulting from the two sets of initial conditions. For a network to achieve an "excellent" overall performance measure of 95 to 100, both steady-state trajectories must achieve "excellent" performance measures (e.g., if the on-trajectory performance measure was 95 and the off-trajectory performance measure was 11, the resulting overall performance measure would be 53, indicating "poor" performance).

V. TRAINING ALGORITHMS

There are several training algorithms that have been developed and applied to training recurrent neural networks, the principal ones being the real-time recurrent learning (RTRL) algorithm [Williams and Zipser, 1989a], backpropagation-through-time (BPTT) [Rumelhart, Hinton, and Williams, 1986; Werbos, 1990], and the extended Kalman filter [Williams, 1992; Puskorius and Feldkamp, 1994]. These algorithms all make use of the gradient of the error function with respect to the weights to perform the weight updates (the specific method in which gradient is incorporated into the weight updates distinguishes the different methods). RTRL computes the gradient information by integrating forward in time as the network runs, while BPTT integrates backwards in time after the network takes a single step forward (other variations of the BPPT algorithm use various quantities for the number of forward steps and the number of backward integration steps [Williams and Zipser, 1995]). BPTT and RTRL are considered gradient-descent algorithms. The extended Kalman filter algorithm uses the gradient in the linearization of the system, such that the method of Kalman filtering may be applied, and is not a gradient-descent algorithm. All three algorithms have been implemented as incremental algorithms, though BPTT has also been modified for use in batch mode [Williams and Zipser, 1995].

The incremental versions of BPTT and RTRL are relatively slow, due primarily to the fact that small learning rates are typically used in order to keep the algorithms stable during training. The batch version of BPTT is faster, as it performs the backwards error integration every M steps, and it can provide the same gradient information as RTRL in a more efficient manner [Williams and Zipser, 1995]. Batch versions also have the attractive quality that they may be used with second-order gradient techniques which generally converge to a minimum in fewer cycles than first-order, incremental algorithms. The extended Kalman filter algorithm has also been shown to converge to a solution in relatively fewer steps [Singhal and Wu, 1989; Shah and Palmieri, 1990; Puskorius and Feldkamp, 1994], and may be implemented with gradient information obtained similarly to RTRL. Therefore, to commonize the development of the algorithms for this analysis, the RTRL method of obtaining gradient information was used and applied to all the algorithms tested here: incremental gradient descent, conjugate gradient descent, and the extended Kalman filter. Note that a faster conjugate gradient algorithm would have used the batch version of BPTT, but RTRL provided the same gradient information and was also applicable in its incremental form for all the other algorithms.

The following sections briefly review gradient-descent-based and Kalman filter-based training algorithms. A more complete discussion can be found in Hagner [1999].

A. GRADIENT DESCENT AND CONJUGATE GRADIENT DESCENT

In gradient descent, the parameters (in this case the weights) of the system are adjusted at each step in the direction of steepest descent, or in the direction of the negative of the gradient vector of the error function. For batch mode, the weights are thus updated according to

$$\mathbf{w}(k+1) = \mathbf{w}(k) - \eta \nabla_w E(\mathbf{w})$$

where η is a positive *learning rate* parameter, and $\nabla_w E(\mathbf{w})$ is the gradient of the error function with respect to the weight vector.

If the instantaneous, or incremental, error function is used instead of the batch error function, the resulting algorithm will not follow the true gradient, but rather an approximation to it. The weights generated by this algorithm will thus have a component of randomness, and therefore this incremental algorithm is termed *stochastic gradient descent*. It is also referred to as the *least mean squares* (LMS) algorithm [Haykin, 1994; Hassoun, 1995] or *incremental gradient descent*. The weight updates are made every step, based on the incremental error of Equation 2.

Gradient descent methods that use second-order information about the error surface to determine (and thus vary) η during training offer improved performance, especially if the error function is a quadratic function of the weights (or close to quadratic). Newton's method [Haykin, 1994; Hassoun, 1995] uses the Hessian matrix $\nabla_w^2 E(\mathbf{w})$ along with the current gradient $\nabla_w E(\mathbf{w})$ to generate the weight updates according to

$$\mathbf{w}(k+1) = \mathbf{w}(k) - [\nabla_w^2 E(\mathbf{w})]^{-1} \nabla_w E(\mathbf{w})$$

This method has the serious drawback that the inverse of the Hessian matrix of the error function (with respect to the weights) is prohibitively time-consuming to calculate for most networks with more than a few weights (the size of the Hessian is the square of the number of weights), and is thus impractical. Additionally, and possibly more importantly, the inverse of the Hessian is required, and there is no guarantee that this matrix is nonsingular at each step.

A more useful method that also employs the Hessian matrix is the conjugate gradient algorithm [Press et al., 1992; Haykin, 1994; Hassoun, 1995], which uses the Hessian matrix implicitly in its calculation of weight updates. It uses the previous gradient and the last step direction to compute a new direction that is conjugate to both, and it does so iteratively without requiring the calculation of either the Hessian or its inverse. The direction vector is calculated in terms of the previous direction vector as

$$\mathbf{v}(k+1) = -\nabla_w E(\mathbf{w}) + \alpha(k)\mathbf{v}(k)$$

where the scalar $\alpha(k)$ is here taken from the *Polak-Ribiere conjugate gradient* formulation [Press et al., 1992; Haykin, 1994] given by

$$\alpha(k) = \frac{[\nabla_w E(\mathbf{w}, k) - \nabla_w E(\mathbf{w}, k-1)] \cdot \nabla_w E(\mathbf{w}, k)}{\nabla_w E(\mathbf{w}, k-1) \cdot \nabla_w E(\mathbf{w}, k-1)}$$

Here all the weights of the network have been collected in a single vector **w**, and the gradient components have also been arranged in a vector. A line-minimization routine is employed to search in the direction **v** to find where the error function takes on its smallest value (along the vector **v**). The step size which results in this minimal value is η_{opt} for this update. The update to the weight vector is then

$$\mathbf{w}(k+1) = \mathbf{w}(k) + \eta_{opt}\mathbf{v}(k+1)$$

The conjugate gradient method provides determination of η_{opt}, as well as a greatly increased convergence rate compared to incremental or batch gradient descent. A reduction in the number of training cycles required for convergence of one to two orders of magnitude was typical for simulations conducted here. The conjugate gradient algorithm has been applied to the training of feedforward neural networks [Makram-Ebeid, Sirat, and Viala, 1989; van der Smagt, 1994]. The conjugate gradient algorithm applied here to the RTRL dynamic derivatives has exhibited very large learning rates at times during the training process, but this does not seem to hamper its performance (it has been reported in Williams and Zipser [1989a] that small learning rates are required for stable algorithm performance).

The conjugate gradient algorithm is by definition a batch algorithm, and as such is not suited for on-line training, where the size of the training data set is not known a priori. However, for the application of trajectory learning here, this was not an issue.

B. RECURSIVE LEAST SQUARES AND THE KALMAN FILTER

The formulation of the least squares algorithm that computes parameter updates based on past parameter estimates is termed the *recursive least squares* (RLS) *filter* and is a special case of the more general *Kalman filter.* For a complete derivation of both, see Haykin [1996]. Both algorithms generate an estimate of an optimal parameter vector that minimizes an error measure (typically the sum of squared error) for a linear system and therefore are applicable to the training of neural networks.

In the case of a single unit, and taking the activation function to be linear such that the unit response is given by $y = \mathbf{w}^{\mathrm{T}}\mathbf{x}$, the method of linear least squares filtering may be employed to find a set of weights that minimizes the weighted sum of squared error given by

$$E(\mathbf{w}) = \sum_{i=1}^{n} \lambda^{n-i} e^{2}(i)$$

where $e(i) = d(i) - y(i)$, $d(i)$ is the target value at time i, and λ^{n-i} is an exponential *forgetting factor*, $0 < \lambda < 1$, used to decrease the effect of past data and permit the algorithm to track variations in data.

The RLS algorithm may be adapted for the case of a network that contains hidden units as well as visible output units, and for the case of units that have nonlinear activation functions (the least squares method and the Kalman filter are methods directly applicable to *linear* systems). This algorithm is called the extended RLS, the *extended Kalman filter* (EKF), or equivalently the *global extended Kalman filter* (GEKF); *global* because the algorithm is applied to the network as a whole, *extended* because the linear RLS has been extended to the nonlinear case, and *Kalman filter* because RLS is a special case of the Kalman filter.

The learning equations [Haykin, 1996] which result from the Kalman filter approach are given below in Equations (4) - (7). Further discussion and analysis of these equations can be found in Hagner [1999].

$$\mathbf{K}(n) = \lambda^{-1}\mathbf{P}(n-1)\mathbf{H}(n)[\mathbf{I} + \lambda^{-1}\mathbf{H}^{\mathrm{T}}(n)\mathbf{P}(n-1)\mathbf{H}(n)]^{-1} \tag{4}$$

$$\mathbf{e}(n) = \mathbf{d}(n) - \mathbf{y}(n) \tag{5}$$

$$\mathbf{w}(n) = \mathbf{w}(n-1) + \mathbf{K}(n)\mathbf{e}(n) \tag{6}$$

$$\mathbf{P}(n) = \lambda^{-1}\mathbf{P}(n-1) - \lambda^{-1}\mathbf{K}(n)\mathbf{H}^{\mathrm{T}}(n)\mathbf{P}(n-1) + \mathbf{Q}(n) \tag{7}$$

Here, $\mathbf{H}(n)$ is a matrix of derivatives of the network unit outputs with respect to the network weights, $\mathbf{K}(n)$ is the Kalman gain matrix, $\mathbf{P}(n)$ is the conditional error covariance matrix, and **Q**(n) is a diagonal covariance matrix which introduces artificial process noise.

It should be noted that the GEKF algorithm formulation given in Equations (4) - (7) has been derived from the RLS algorithm with exponential forgetting. The forgetting factor λ is not employed (or, equivalently, set to unity) in the EKF formulation of Singhal and Wu [1989] and the GEKF formulations of Puskorious and Feldkamp [1994], and thus the GEKF algorithm presented here is slightly different. A variable scalar learning rate, $\eta(n)$, is used in Puskorious and Feldkamp [1994] which results in a formula for the Kalman gain, ***K***(n), different from (4), given by

$$\mathbf{K}(n) = \mathbf{P}(n-1)\mathbf{H}(n)[\eta^{-1}(n)\mathbf{I} + \mathbf{H}^{\mathrm{T}}(n)\mathbf{P}(n-1)\mathbf{H}(n)]^{-1}$$

where $\eta(n)$ is typically set to a value less than unity at the start of training and increases to unity as training progresses.

The GEKF algorithm is computationally intensive due primarily to the update calculations for the approximate (due to the linear system approximation) conditional error covariance matrix ***P***(n), which scales as the square of the number of weights in the network. A modification to the GEKF algorithm that assumes certain interactions between weights are negligible is the *decoupled extended Kalman filter* (DEKF) algorithm [Puskorius and Feldkamp, 1994]. The negligible weight interactions are accounted for as zeros in the ***P***(n) matrix, and if the weights are grouped such that there are assumed to be no interactions between weights in different groups, the ***P***(n) matrix can be arranged in block-diagonal form. If the groups are chosen such that weights feeding a unit make up a group, then the decoupling is termed *node-decoupled*, and the algorithm is called *node-decoupled* EKF, or NDEKF [Puskorius and Feldkamp, 1994].

The derivation of the DEKF (or NDEKF) algorithm proceeds similarly to that for the GEKF algorithm, except that the block-diagonal form of ***P***(n) is exploited to reduce the computational complexity. For the case of g groups of weights, there will now be g weight vectors ***w***(n), as well as g ***H***(n), ***P***(n), and ***K***(n) matrices, which are subsets of their full, GEKF counterparts.

A variation of the single-unit RLS algorithm that employs linearization of the nonlinear unit activation function (similarly to the EKF algorithms described above), and an approximation to the estimation error, $e(n)$, has been developed and termed the *multiple extended Kalman algorithm* (MEKA) [Shah and Palmieri, 1990], and is applicable to multi-layered networks. This algorithm in effect applies an RLS optimization separately to the individual units of the network, whereas the NDEKF algorithm, which includes only weight interactions in a unit's weight group, carries out a global filtering (estimation) operation.

Various EKF algorithms have been successfully applied to the training of both feedforward (MLP) [Shah and Palmieri, 1990] and recurrent [Singhal and

Wu, 1989; Williams, 1992; Puskorius and Feldkamp, 1994] networks. GEKF is computationally intensive, and the modifications (DEKF and MEKA) to the standard algorithm have provided substantial reduction in the computations required, resulting in faster algorithms that retain much of the power of GEKF. These applications of EKF algorithms have been shown to provide convergence in relatively few training iterations, offset partially by an increase in computations over gradient-based algorithms.

VI. SIMULATIONS

A. ALGORITHM SPEED

Comparisons have been made among the speeds of training algorithms for recurrent networks [Logar, Corwin, and Oldham, 1993; Williams and Zipser, 1995]. The focus of this work is on the effects of recurrence on network performance and the efficacy of various training algorithms for architectures with recurrence, and thus the analysis here has been limited to only five training algorithms in a primary effort to find the common effects of recurrence, and secondarily to compare the algorithms' performances on the applications considered here (the ability of the algorithm to converge to a good solution was analyzed in more detail than its pure computational complexity).

All of the algorithms obtained the gradient information from the identical RTRL calculation, which has a computational complexity of $O(n^2)$, where n is the number of weights. The optimization algorithms had complexities of $O(n)$ for the incremental and conjugate gradient descent algorithms; $O\left[\sum_{i=1}^{g} n_i^2\right]$ where n_i is the number of weights for a given unit, and g is the total number of units in the network for the MEKA and NDEKF algorithms; and $O(n^2)$ for the GEKF algorithm. Therefore, the overall training algorithm speed was dominated by the common RTRL calculation.

The RTRL type of calculation for obtaining the gradient, $\nabla_w E(\mathbf{w})$, that includes dependence on past values of the gradient, is required for internally recurrent networks and for externally recurrent networks that use less than full teacher forcing. If an externally recurrent network uses full teacher forcing, then the current gradient does not depend on past values, because the output units had their values set to the target values before each step, and these target values are constants and have no dependence on the weights. This permits a straightforward backpropogation of the error to obtain the gradient, with only partial derivatives and no total derivatives used, which has a computational complexity of $O(n)$. This would have been applicable to all the externally recurrent networks using full teacher forcing, resulting in increased training speed for all the algorithms, but especially for the incremental and conjugate gradient-descent algorithms, which would have had a *total* computational complexity of $O(n)$. RLS algorithms’ speed would have then been dominated by the RLS complexity of $O\left[\sum_{i=1}^{g} n_i^2\right]$ for the for the MEKA and NDEKF algorithms, and $O(n^2)$ for the GEKF algorithm.

The resulting algorithm computational complexities were measured for a fixed number of iterations, and calculated in units of seconds/cycle, where each cycle involved one pass through the M data points. These values were then plotted versus the square of the number of weights, as shown in Figure 4, to check the overall $O(n^2)$ dependence expected.

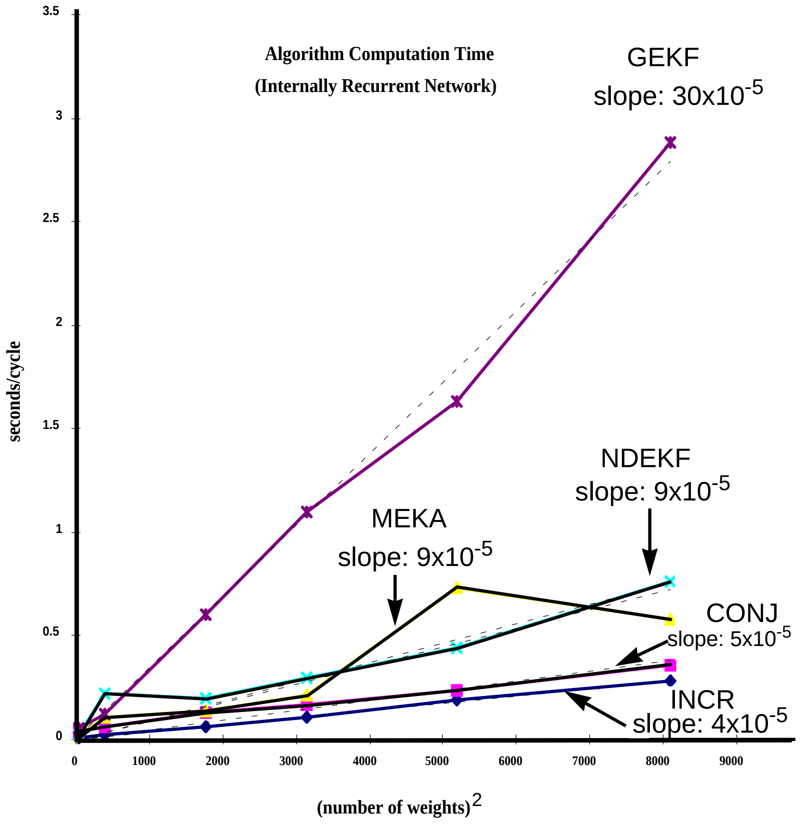

Figure 4. Training algorithm computation time (for internally recurrent networks) in seconds per training cycle vs. the square of the number of network weights (using a 200 MHz Pentium CPU). Approximate slopes of linear trend lines are shown.

The approximately linear dependence on n^2 is evident for all the algorithms, and the slopes of the linear trend lines may be compared to give the relative speeds of the training algorithms. Normalizing the speed of the slowest algorithm

(GEKF) to 1, the relative speeds of the other algorithms are approximately 3.3 for the MEKA and NDEKF algorithms, 6.0 for the conjugate gradient, and 7.5 for the incremental gradient-descent algorithm. This gives approximate confirmation to the speed-up expected for the MEKA and NDEKF algorithms over the GEKF algorithm (e.g., for a network with 6 units, $n = 42$, $n^2 = 1764$, and $\sum_{i=1}^{6} 7^2 = 294$, the expected order of speed-up is 1764/294 = 6, which is relatively close to the 3.3 obtained experimentally). Additionally, both the incremental and conjugate gradient algorithms are faster than the RLS algorithms, as expected, with the conjugate gradient algorithm being somewhat slower than the incremental gradient algorithm, due to the computational burden of performing the conjugate direction calculation, minimum bracketing, and line minimization routines.

Algorithm computation times for externally recurrent networks are not shown in Figure 4, due to the lack of validity of comparison with internally recurrent networks. In fact, because of the use of the RTRL algorithm to obtain gradient information instead of using standard backpropogation for the externally recurrent networks with full teacher forcing, all the algorithms ran more slowly on these networks than on the internally recurrent networks, due to the additional computations for multiple layers and delays (standard backpropogation would have enabled externally recurrent networks to train more quickly than internally recurrent networks). So, while the externally recurrent networks could have been faster, they were approximately twice as slow in experimental computation time measurements.

B. CIRCLE RESULTS

To learn the circle trajectory, all algorithms used full teacher forcing, as it generally provided the fastest and most robust learning. Partial teacher forcing sometimes resulted in very fast convergence, but was not a robust technique; learning often diverged as the algorithms became unstable.

The network weights for all the architectures and algorithms were obtained from a uniform random distribution from -0.1 to +0.1.

The incremental gradient descent algorithm for the circle trajectory employed learning rates for the feedforward weights and recurrent weights of 0.002. Larger learning rates decreased the number of cycles required for convergence, but resulted in solutions with lower performance due to the algorithm taking relatively large steps around the vicinity of the minimum. The value used here provided a good trade-off between performance and convergence speed. The figure-eight trajectory simulations used learning rates for the feedforward weights of between 0.1 and 0.2 for the externally recurrent architectures, and a feedforward weight learning rate of .01 and recurrent weight learning rate of 0.2 for the internally recurrent architectures. The learning rates for the figure-eight trajectory were larger than those for the circle trajectory because of the large number of iterations needed to approach convergence; the largest rates possible that permitted stable algorithm performance were used. It was found that for the internally recurrent architectures, a feedforward weight learning rate that was smaller than the recurrent weight learning rate by at least a factor of ten ensured algorithm stability.

The RLS algorithms (GEKF, NDEKF, MEKA) were "tuned" for the different algorithm/architecture combinations, though the parameters that were varied were within the following typical ranges. The process noise matrix, ***Q***, was diagonal with elements typically set to 10^{-4}. The forgetting factor, λ, was typically set to the schedule of 0.999 - 0.9999, increasing by 4.5×10^{-6} each step.

Two sets of initial conditions were tested for the results presented here, the first providing the output units with initial condition values **on** the trajectory, and the second with values **off** the trajectory. Note that for externally recurrent networks it is possible to obtain initial conditions that place the network exactly on the desired trajectory, because the initial unit output values have no effect (the network output is a function of only the input and the weights). However, for the internally recurrent networks this exact placement is not possible, because the initial unit output values do contribute to the network output, and these unit values are not known. In these simulations, a "best" estimate for the internally recurrent network initial unit output values was used, which was the actual unit output values at the last or *M*th step during training. Thus as the internally recurrent network training error was reduced, the initial conditions of the hidden unit outputs more closely matched those required to be on the target trajectory.

A typical simulation result is shown in Figure 5, which shows a network's output for 10 "loops" (one loop is defined as taking M steps, where M is the number of training points, 100 in these simulations) starting from the two different initial conditions on and off the desired trajectory. The result shown is for an internally recurrent network with 2_R architecture, trained with the conjugate gradient algorithm for 7 cycles.

The first step that the network takes from these initial conditions is indicated on the plots by the small circle indicating the first trajectory point. The trajectory with initial conditions off-trajectory provides some measure of a solution's *basin of attraction*, and the degree to which the limit cycle trajectory is an attractor. The trajectory shown in Figure 5 is a stable attractor, with the off-trajectory initial condition resulting in a trajectory that spirals outward from the origin in a few loops to converge to the limit cycle oscillation of the desired trajectory. Note that all 10 loops are shown in both plots, indicating both the degree to which the trajectory is stable, and the closeness to which it follows the desired trajectory (the target trajectory is indicated by the dotted line).

A limit cycle will exhibit convergence from both sides of the trajectory, and this characteristic is able to be seen in Figure 6. The results are for the same network as in Figure 5, though with two different initial conditions, one inside and one outside of the trajectory, obtained by setting the initial unit values to 0.07 and 0.27, respectively.

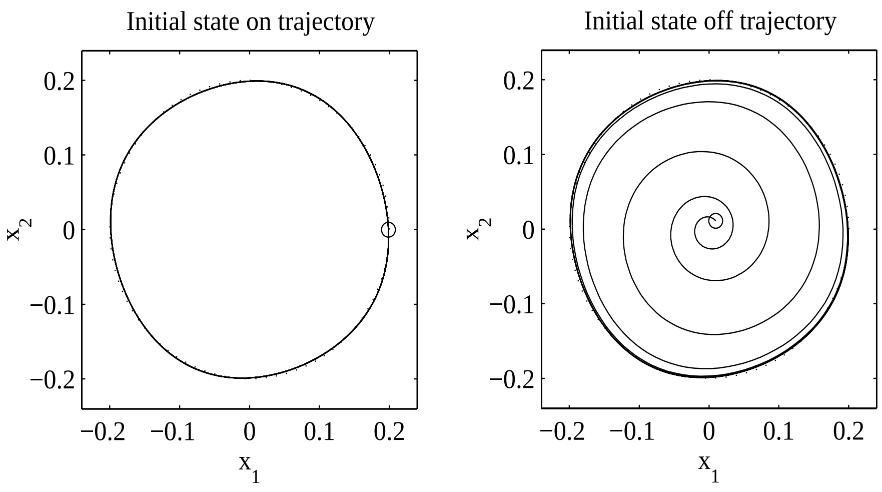

Figure 5. Circle trajectory generated by 2_R network trained with the conjugate gradient algorithm for 7 cycles (performance measurement: 99.7).

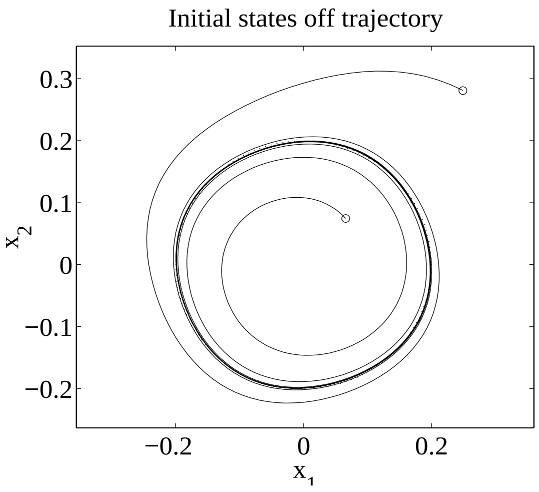

Figure 6. Two convergence regions for the same 2_R network as in Figure 5.

Figure 7 shows the resulting trajectories for the larger, $4 \times 2(1)$ network, trained with the GEKF algorithm for 300 cycles. In general, the faster trajectory convergence shown here, compared to that for the smaller, 2_R network (shown in Figure 5), was typical for the larger networks, possibly because the smaller networks required the units to operate further in their nonlinear regions to achieve the nonlinear trajectories, and possibly due simply to the convergence dynamics resulting from a larger number of units.

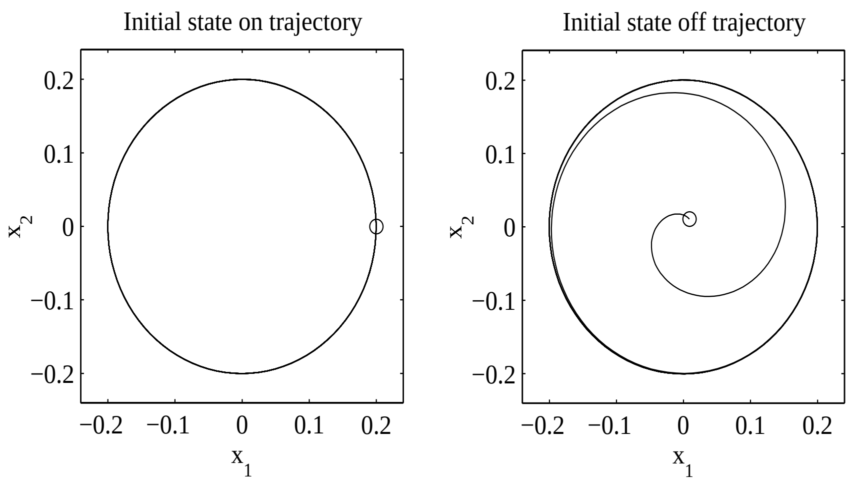

Figure 7. Circle trajectory generated by $4 \times 2(1)$ network trained with the GEKF algorithm for 300 cycles (performance measurement: 100).

All 20 architecture/algorithm combinations learned the circle trajectory and provided excellent performance for the 100 simulations, as shown in Table 1. Each architecture/algorithm was trained 5 times with different initial weight values to provide some measure of the learning performance repeatability, and the initial weight values were identical for all algorithms and a given architecture.

A performance was counted as successful if it provided stable, limit cycle oscillation for both on- and off-trajectory initial conditions (this stability was determined by visual inspection). The table row labeled “Success” gives the number of stable solutions out of 5 runs total. The row labeled “Ave Performance” gives the average performance value of those solutions that were stable. The row labeled “Ave Cycles” gives the average number of cycles required during learning to achieve the given performance for the stable solutions. The row labeled “Training Time” is an estimate of the total training time, in minutes, for the algorithm to iterate through the “Ave Cycles” given, calculated from the algorithms’ computation times given by the linear trend lines in Figure 4.

As expected, the conjugate gradient, GEKF, NDEKF, and MEKA algorithms in general converged to solutions in far fewer cycles than the incremental gradient algorithm (on the order of 100 times fewer cycles for the internally recurrent architecture, and approximately 5 times fewer cycles, on average, for the externally recurrent architecture). The one notable exception was the performance of the NDEKF algorithm on the $2 \times 2(1)$ architecture: this algorithm required a relatively large number (14400 on average) of cycles and time to converge, possibly due to the effect of neglecting the coupling of weights in the conditional error covariance matrix, ***P***(*n*).

Training Algorithm	**Metric**	**Architecture**			
		Single Layer		**Tap Delay Net**	
		$\mathbf{2}_{\boldsymbol{R}}$	$\mathbf{4}_{\boldsymbol{R}}$	**2x2(1)**	**4x2(1)**
Incremental Gradient Descent	Success	5	5	5	5
	Ave Performance	99.7	99.6	99.1	99.1
	Ave Cycles	3000	3000	6400	4000
	Training Time (min)	0.1	0.8	12.3	2.6
Conjugate Gradient Descent	Success	5	5	5	5
	Ave Performance	99.7	99.5	99.8	99.9
	Ave Cycles	7	11	1100	1440
	Training Time (min)	0.0002	0.004	0.3	1.2
GEKF	Success	5	5	5	5
	Ave Performance	99.4	99.6	100	100
	Ave Cycles	19	31	680	720
	Training Time (min)	0.003	0.1	1.0	3.5
NDEFK	Success	5	5	5	5
	Ave Performance	99.4	99.4	97.4	98.6
	Ave Cycles	17	34	144000	1200
	Training Time (min)	0.001	0.02	6.2	1.7
MEKA	Success	5	5	5	5
	Ave Performance	99.4	99.3	99.5	99.5
	Ave Cycles	20	22	360	340
	Training Time (min)	0.001	0.01	0.2	0.5

Table 1. Circle trajectory simulation results.

There was no performance improvement for the larger networks compared to their smaller counterpart [4_R vs. 2_R and $4 \times 2(1)$ vs. $2 \times 2(1)$], indicating that the smaller networks were adequate to learn the circle trajectory and were not affecting network or algorithm performances (except for convergence dynamics, as noted earlier).

The primary conclusion drawn from the above experimental analysis is that the speed of convergence of the conjugate gradient, GEKF, NDEKF, and MEKA algorithms for the internally recurrent architectures was much greater than for the externally recurrent architectures. The internally recurrent networks converged in at least 10 times fewer (and often 100 times fewer) cycles, and in at least 50 times less (and often 200 times less) time.

It is also notable that for this trajectory, the very simple algorithm of incremental gradient descent provided solutions with performances comparable to those for the more complex algorithms, indicating that for certain trajectories, incremental gradient descent is adequate. And though incremental gradient descent required many more training cycles, it had the smallest cycle computation time, resulting in total training times comparable to the other algorithms.

C. FIGURE-EIGHT RESULTS

The initial weights, teacher forcing, and algorithm parameters were set to values similar to those used for the circle trajectory. It was found that full teacher forcing again provided the best learning performance.

The resulting figure-eight trajectories for the different successful (stable attractor) solutions were dissimilar for the different simulations, unlike the circle results, which were almost identical. The trajectories shown in Figures 8 and 9 show the results of solutions for the 4_R architecture trained with GEKF for 2000 cycles, and the 8_R architecture trained with GEKF for 500 cycles, respectively. Both trajectories are stable attractors and the basins of attraction exhibit quite different dynamics prior to convergence to the final trajectory. As for the circle trajectory, it was found that, in general, the larger the network, the smoother the convergence from the off-trajectory starting point to the final trajectory, consistent with the results shown in Figures 8 and 9.

The results for the 150 simulations of the single layer architectures are given in Table 2, and the results for the tap delay networks are given in Table 3. Again, the numbers in the tables represent the results of 5 different runs.

The figure-eight was, in general, far more difficult to learn for all the networks and training algorithms than the circle. This is most likely because the trajectory crosses itself in output-unit (phase-plane) space, so that the network must store not only the past state of the trajectory, but also information about *multiple* previous states (e.g., storing the direction, or derivative of the trajectory).

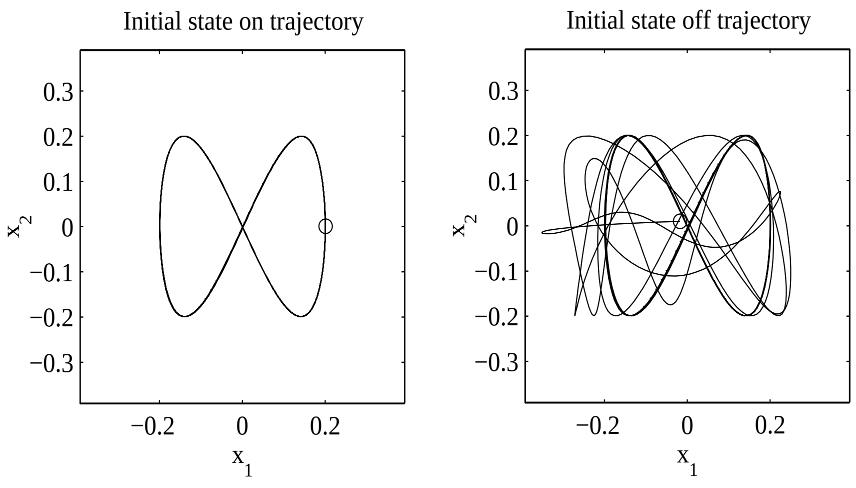

Figure 8. Figure-eight trajectory generated by 4_R network trained with the GEKF algorithm for 2000 cycles (performance measurement: 100).

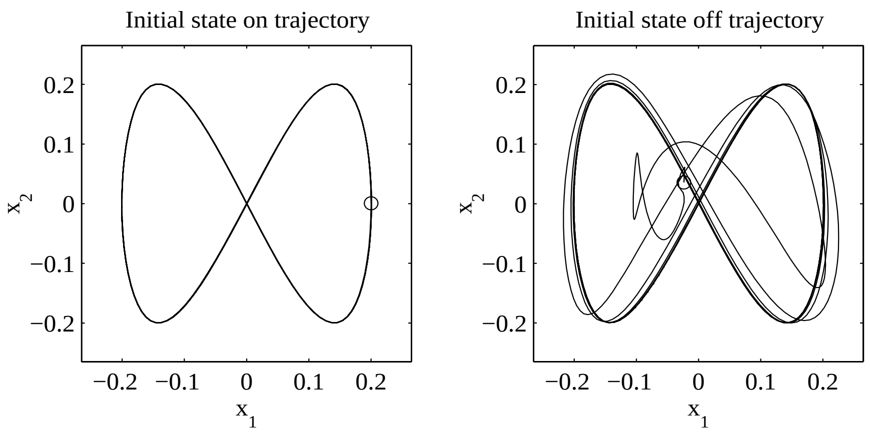

Figure 9. Figure-eight trajectory generated by 8_R network trained with the GEKF algorithm for 500 cycles (performance measurement: 99.9).

As with the circle trajectory, the conjugate gradient, GEKF, NDEKF, and MEKA algorithms converged to solutions in far fewer cycles than the incremental gradient algorithm (on the order of 15 times fewer cycles for the internally recurrent architecture and approximately 150 times fewer cycles, on average, for the externally recurrent architecture).

The incremental gradient descent and MEKA algorithms exhibited some performance improvement for the larger internally recurrent networks compared to their smaller counterparts, as did the conjugate gradient algorithm for the externally recurrent architecture. This does not, however, indicate that the smaller networks were inadequate to learn the figure-eight trajectory, because the excellent performances of the networks trained by the GEKF algorithm show that all the networks contained ample representational capability.

Training Algorithm	**Metric**	**Single Layer Architecture**		
		4_R	**6_R**	**8_R**
Incremental Gradient Descent	Success	1	1	2
	Ave Performance	91.7	95.5	97.1
	Ave Cycles	48000	14000	29000
	Training Time (min)	12.8	16.5	100.2
Conjugate Gradient Descent	Success	0	0	0
	Ave Performance	-	-	-
	Ave Cycles	-	-	-
	Training Time (min)	-	-	-
GEKF	Success	3	3	3
	Ave Performance	99.3	99.6	99.8
	Ave Cycles	1470	900	770
	Training Time (min)	2.94	7.9	20.0
NDEFK	Success	2	2	4
	Ave Performance	98.1	98.6	98.6
	Ave Cycles	2900	2050	475
	Training Time (min)	1.74	5.4	3.7
MEKA	Success	1	2	4
	Ave Performance	85.1	98.4	97.9
	Ave Cycles	1200	2100	2050
	Training Time (min)	0.72	5.6	15.9

Table 2. Figure-eight trajectory simulation results.

Training Algorithm	Metric	Tap Delay Net Architecture		
		4x2(4)	6x2(4)	8x2(4)
Incremental Gradient Descent	Success	0	0	0
	Ave Performance	-	-	-
	Ave Cycles	-	-	-
	Training Time (min)	-	-	-
Conjugate Gradient Descent	Success	2	1	2
	Ave Performance	95.3	97.6	99.4
	Ave Cycles	4500	9000	10000
	Training Time (min)	15.9	69.4	135.0
GEKF	Success	1	4	3
	Ave Performance	98.6	99.6	99.2
	Ave Cycles	800	730	600
	Training Time (min)	16.9	33.8	48.6
NDEFK	Success	0	0	0
	Ave Performance	-	-	-
	Ave Cycles	-	-	-
	Training Time (min)	-	-	-
MEKA	Success	0	0	1
	Ave Performance	-	-	89.5
	Ave Cycles	-	-	450
	Training Time (min)	-	-	10.9

Table 3. Figure-eight trajectory simulation results.

The primary conclusion drawn from the figure-eight trajectory simulations is that 4 out of the 5 training algorithms were able to converge to good solutions for the internally recurrent network, and only 2 out of the 5 were able to do so for the externally recurrent architectures, indicating that for this limit cycle trajectory, the internally recurrent architecture is the better choice.

The internally recurrent architecture did, however, pose difficulty for the conjugate gradient algorithm, which became quickly trapped in poor local minima for all of the 15 simulations. This may indicate that internal recurrence results in more local minima than external recurrence, and that the 4 incremental algorithms are robust enough to escape these minima but the conjugate gradient algorithm is not.

The other notable conclusion is that the GEKF learning algorithm was far superior to the other 4 algorithms for this trajectory. The GEKF algorithm reached good solutions 57% of the time (17 out of 30 simulations), and converged to good solutions for all 6 of the architectures. None of the other algorithms was able to reach good solutions for all the architectures or both types of recurrence, as indicated by the blank entries in Tables 2 and 3. Of course, it is possible that the other algorithms might have reached good solutions for these architectures if additional simulations had been run. Additionally, the performances of the solutions obtained with the GEKF algorithm were in all simulations superior to those obtained by the other algorithms. This excellent performance indicates that the capability of the algorithm more than made up for its relatively high computational complexity, with the result that it is the preferred algorithm for learning this figure-eight trajectory.

D. ALGORITHM ANALYSIS

Incremental gradient descent. This algorithm was relatively slow, as expected, compared to the other, second-order algorithms. This was not a large problem for networks learning the circle trajectory, as this algorithm found minima that provided excellent performance results very similar to the other algorithms. This was most likely due to the shape of the error cost function in weight space, which appeared to contain very few, if any, local minima. This shape of the error surface permitted all the algorithms to find minima with good solutions (in fact, very often the different algorithms running with different initial weight values converged to the *same* minimum, identified by the nearly identical final weight vector).

When this algorithm was applied to the figure-eight trajectory, however, it performed poorly. The algorithm converged very slowly, requiring tens of thousands of training cycles to approach a minimum, which was often one that provided poor performance. Gradient descent was so slow that it was impractical for use with the figure-eight trajectory, compared to the superior convergence properties of the second-order algorithms. (Gradient descent required almost 2 hours to reach a good (though poorer than the other 4 algorithms) solution for the 8_R network, and almost 24 hours for the $8\times2(4)$ network; note that the solutions for the $8\times2(4)$ network were not counted as stable, as the performance metric never became consistent). An advantage of this algorithm is that it was very robust (given small enough learning rates), requiring no heuristics to keep the algorithm from diverging or to optimize performance.

Conjugate gradient descent. This algorithm converged in relatively few iterations, compared to the incremental gradient-descent algorithm, as expected. It performed very well on the circle trajectory, but less well on the figure-eight. This algorithm, due to its inherent line minimization routine, was susceptible to becoming trapped in local minima, which was evident for the internally recurrent architecture learning the figure-eight trajectory. It performed better with the externally recurrent architecture on the figure-eight, though its convergence rate was inferior to those of the RLS algorithms. As was the case for the incremental gradient-descent algorithm, this method was very robust, requiring no heuristic adjustments to algorithm parameters to ensure stable convergence characteristics.

Recursive least squares. These algorithms also converged in far fewer iterations than did the incremental gradient-descent algorithm. In addition, as a group they performed better than the incremental and conjugate gradient-descent methods. They did, however, require the appropriate setting of algorithm parameters to optimize performance, which required additional "set-up" time not necessary for the gradient-descent algorithms. It was necessary to set two primary parameters, the process noise matrix $\boldsymbol{Q}$ and the forgetting factor λ, to values appropriate for the application.

Values for $\boldsymbol{Q}$ in the range of 10^{-2} to 10^{-6} were best, and typically 10^{-4} was used in the simulations. While the inclusion of the process noise matrix is not included in the standard RLS algorithm derivation, it is a standard part of the Kalman filter algorithm. Because of the similarity of these algorithms, the process noise matrix was tested with the RLS algorithms, found to be very beneficial in increasing the convergence rate, and was thus used in all the simulations.

The use of forgetting factor λ, (a positive number less than unity called the *exponential forgetting factor* [Åström and Wittenmark, 1989]), provides an ability for the estimator to track variation in the input or, equivalently, to discount old data by weighting it less. Values less than unity also had the effect of increasing the rate of convergence quite substantially, most likely due to the fact that the forgetting of old information when the actual trajectory was far from the desired trajectory was beneficial. Initial λ values smaller than final values provided even faster convergence, and the typical schedule was 0.999 - 0.9999, increasing by 4.5×10^{-6} each step (this implied reaching the final value in 200 steps, or 2 training cycles, for the 100-point data sets used here). Forgetting factor values less than unity did, however, cause the RLS algorithms to become unstable during periods when the updates to the weights were small, as will be discussed in the following section.

E. ALGORITHM STABILITY

The conjugate gradient-descent algorithm was the most stable of the five tested here. It never diverged during training, and required no tuning of parameters to ensure this stability. However, this stability sometimes came at the cost of the algorithm becoming trapped in local minima. The incremental gradient descent algorithm was also very stable. The algorithm diverged only when too large a learning rate was chosen. This was easily remedied by decreasing the learning rate through trial-and-error to find the largest value that was stable.

The RLS algorithms were not as stable as the other two algorithms just discussed. There were two sources of instability: the process noise matrix $\boldsymbol{Q}$ and the forgetting factor λ.

If $\boldsymbol{Q}$ was too large, the algorithm would not converge, in effect attempting to estimate the noise rather than learning the trajectory. This problem was addressed by the trial-and-error method to choose the largest element values for $\boldsymbol{Q}$ that permitted smooth error reduction during training.

The forgetting factor λ was typically chosen, as previously indicated, to vary over the range of 0.999 to 0.9999, incrementing by 4.5 x 10^{-6} each step. If values much smaller than 0.999 were used for the *initial* value, the algorithm was unstable during the period in which λ was small. If values much smaller than 0.9995 were used for the *final* value, the algorithm reduced error rapidly but sometimes became unstable before reaching a minimum. A final value of 1.0 was stable, but resulted in very slow reduction of error.

As indicated above, the use of *exponential forgetting* provides an ability for the RLS estimator to track variation in the input and discount old data. When the algorithm enters a region where the updates to the weights are very small, then the inputs to the estimator are fairly constant, and there is little new information provided by each step. $\boldsymbol{P}(n)$ increases exponentially, leading to what is termed *estimator windup* [Åström and Wittenmark, 1989].

Exponential forgetting is thus sensitive to the degree to which the system is persistently excited, or the amount of new information that is provided at each step. Unfortunately in the problems considered here, where there is no external input (excitation), the input will not be (sufficiently) persistently excited during all phases of training, and methods to ensure sufficient excitation that are useful on certain system identification problems, such as injecting extra perturbation signals, are not applicable here, as perturbation would cause the system to learn a response different from the desired limit cycle oscillation.

Other methods to avoid estimator windup are to keep $\boldsymbol{P}(n)$ bounded, to stop weight updates when the estimator error is small, and to adjust the forgetting factor automatically [Haykin, 1996] or by a schedule such as setting λ to 1.0 after a predetermined number of cycles or at a certain level of estimator error. Methods for ensuring $\boldsymbol{P}(n)$ remains bounded, such as by keeping the trace of the $\boldsymbol{P}(n)$ matrix constant at each iteration or selectively forgetting information only in the direction generating new information, are given in [Haykin, 1996].

The problem of estimator windup was evident in all three RLS algorithms after they had reached points at which the weight updates were very small, but had a more deleterious effect on both the MEKA and NDEKF algorithms, as indicated in Tables 2 and 3 by their relatively poor performances on the figure-eight trajectory for internally recurrent architectures and the lack of stable solutions found for the externally recurrent architectures. These algorithms often diverged before the error had been reduced to values small enough to result in good performances. In this respect, these versions of the RLS algorithm were more susceptible to estimator windup than the GEKF algorithm, which did exhibit divergence, though after the algorithm had advanced sufficiently close to minima providing good performance. It is unknown if these algorithms could have efficiently reduced the error further, and if this would have resulted in solutions with better performance, though the ability of the conjugate gradient descent algorithm to do so for the externally recurrent networks suggests that this is the case. It is thus likely that the MEKA and NDEKF algorithms could benefit significantly from the use of the stabilizing heuristics mentioned above for avoiding estimator windup, or from the use of different variations of the RLS algorithm such as the square-root adaptive filter [Söderström and Stoica, 1989; Haykin, 1996].

It should be noted that the instability caused by estimator windup is due to the exponential forgetting employed in the RLS derivation. It is not a problem for the slightly different EKF algorithms that are derived using Kalman filter methods [Singhal and Wu, 1989; Puskorius and Feldkamp, 1994], and thus these formulations may provide more stable operation than those derived here using RLS methods. It is not known, however, how the benefits of the learning rate heuristic used in Puskorius and Feldkamp [1994] for the GEKF and NDEKF algorithms compare to those exhibited by the exponential forgetting in the RLS-derived GEKF and NDEKF algorithms given here.

F. CONVERGENCE CRITERIA

The learning algorithms occasionally generated network weights that provided good results *prior* to convergence, and poor results once convergence was attained. These good solutions were not due to weights that constituted a minimum in the error surface, and thus the algorithms passed through these regions of weight space on the way to a minimum.

Training was stopped for the simulations in this work if convergence was reached (for the conjugate gradient and RLS algorithms, this was fairly evident by the fact that the reduction of training error at successive iterations became negligible), or if the reduction in training error was small and the performance value for successive iterations remained within a band, typically ± 2 units of the performance metric used here, given in Equation 3.

The good solutions obtained in the middle of training did not usually meet these convergence criteria, and thus were not accepted as valid. This did not greatly affect the number of good solutions found by the architecture/algorithm combinations because algorithms tended to converge later to minima with solutions of equal or higher performance. However, for some simulations, especially for incremental gradient descent, convergence resulted in poor results, indicating that the algorithm was beginning to overfit the data and had passed the point where the network generalizes well. In these instances, it would have been possible to employ a form of *early stopping* [Hassoun, 1995] to stop training in a region of weight space that provided good performance, though prior to convergence. For example, the early stopping technique could have been used during the simulation depicted in Figure 3, where the training had relatively short periods where the performance was good, prior to and after convergence.

G. TRAJECTORY STABILITY AND CONVERGENCE DYNAMICS

The basin of attraction that a trained network exhibits is one measure of the stability of the network, or its robustness with respect to initial conditions. To study this property, the networks were tested with starting points far from the trajectory. As stated above, for the circle trajectory the initial condition values were 0.0001 and for the figure-eight 0.01.

All 100 of the circle simulations resulted in a similar basin of attraction for the initial condition off the trajectory, as shown in Figure 5. The trajectory spiraled out from the origin, taking several “loops” to converge to the desired circle trajectory. Ten loops of network trajectory are shown in the plot, indicating that the remaining loops were coincident, and therefore had converged to a stable trajectory. Thus the trajectory was a stable attractor.

As seen in Tables 2 and 3, only 38 out of 150 (or 25%) of the simulations for the figure-eight resulted in networks that produced stable attractors. Many of the simulations did produce sustained oscillators (the on-trajectory initial conditions resulted in a trajectory following the target trajectory), but not attractors (the off-trajectory initial conditions resulted in a trajectory that failed to converge to the desired trajectory within 10*M* steps). In some simulations this may have been due to the trajectories being similar to the *center*, or *vortex*, trajectories generated by two-dimensional *linear* systems with a purely imaginary conjugate pair of eigenvalues, indicating that the network was not exploiting the nonlinearities of the hidden units. The statistics were not kept for this subset of results, as the interest was is generating stable attractor trajectories.

In some of the figure-eight simulations, when the algorithms were near convergence to a minimum, the performance would sometimes switch between excellent (values in the high 90's) and poor (values in the high 40's) as the off-trajectory result changed from converging to the desired trajectory to not converging (not an attractor), as shown in Figure 3. Because the on-trajectory result was still good (the trajectory was stable) and the contribution of the off-trajectory result was zero, the total performance measure was reduced by a factor of two. This indicated that the off-trajectory performance was very sensitive to the initial conditions. If only small changes in the weights could cause the trajectory to switch between converging and not converging, it is most likely that changes to the values of the off-trajectory initial conditions would also have a dramatic effect on the trajectory convergence characteristics.

Note that in the simulations presented above, the neural nets were trained only with data on the trajectory itself, and not with noisy data, or data from a basin of attraction around the trajectory. Although not explicitly trained to learn an attractor limit cycle, the simulation results show that the networks do, in fact, produce such asymptotically stable attractors. This inherent stability was evident for both the internally and externally recurrent networks and has been previously reported [Williams and Zipser, 1989b; Pearlmutter, 1995; Tsung and Cottrell, 1995; Cohen, Saad, and Marom, 1997]. The basins of attraction for these types of figures were studied in Tsung and Cottrell [1995] and Sundareshan and Condarcure [1998], but in Tsung and Cottrell [1995] the training data were chosen specifically to produce desired basins of attraction, and in Sundareshan and Condarcure [1998] the desired trajectory data included the initial, transient trajectory from the origin out to the final circle trajectory and thus was explicitly trained.

Why recurrent neural nets, trained only with data on the trajectory, are able to produce stable attractor limit cycles is not clear. Further, the use of full teacher forcing in effect trains the network to step to a point close to the trajectory, starting from a point on the trajectory, as discussed in Tsung and Cottrell [1995]. This is the opposite of what is required for a limit cycle, for which the network needs to step to a point on the trajectory, starting from a point off the trajectory (as is done when no teacher forcing is used). Thus the limit cycle properties observed in these simulations are inherent characteristics of the resultant network dynamics.

VII. CONCLUSIONS

Internally recurrent hidden layers did not increase network performance over single-layer internally recurrent networks, and multiple feedforward hidden layers did not improve the performance of feedforward, externally recurrent networks, for the limit cycle trajectories considered in this work.

All the architecture/algorithm combinations were able to learn the circle trajectory, with the internally recurrent architectures providing convergence in far fewer cycles than the externally recurrent architectures, especially for the conjugate gradient and RLS algorithms.

The figure-eight trajectory proved to be much more difficult to learn than the circle, presumably due to the trajectory's crossing itself. In this case, two different points on the trajectory require the network to produce identical output values. The internally recurrent architectures permitted convergence to good solutions more often than did the externally recurrent architectures (28 vs. 14 good solutions out of 75 simulations each for the internally and externally recurrent networks, respectively). The GEKF algorithm proved to be the superior training algorithm for this trajectory, providing the most good solutions and the solutions with the best performances. The GEKF algorithm found limit cycle solutions for 17 of the 30 possible (compared to 8, 8, 5, and 4 for the NDEKF, MEKA, conjugate gradient, and incremental gradient algorithms, respectively).

GEKF was able to repeatedly find good solutions for both the internally and externally recurrent network architectures, an ability that was not achieved by any of the other algorithms. It appears that the excellent performance of the GEKF algorithm was due to its ability to converge to minima with very low values of error, and it did so in relatively few training cycles. While initial experimentation on nonlinear single input-single output system identification shows agreement with the above findings, further analysis is needed to determine if these results are applicable to problems of nonlinear dynamic system modeling.

The incremental and conjugate gradient-descent algorithms are quite stable, while the RLS algorithms suffer from instability due to estimator windup near convergence, though this was less of a problem for the GEKF algorithm.

The networks were, in general, able to learn to generate limit cycle trajectories, with basins of attraction in which trajectories converged to final, steady-state trajectories. This convergence property was inherent in the resulting network dynamics, and not explicitly part of the training method.

In retrospect, it would have been beneficial to separate the performances for the two initial conditions tested, and thus have distinct metrics for the network's performance as a sustained oscillator and as an oscillator that was an attractor. Also, testing trained networks with multiple off-trajectory initial conditions (rather than only one) would have provided more information about the basin of attraction for the trajectories. Cursory testing of the figure-eight trajectories with multiple initial conditions indicates that these trajectories had complex attractor characteristics, where some initial conditions resulted in convergence to limit cycles, some converged to fixed points, and some produced chaotic trajectories that did not appear to either converge or diverge.

A possibility for future work would be to continue the initial analysis on identification of nonlinear systems, and extend this by studying the performances of the recurrent network architectures and training algorithms for identification of real physical systems with experimentally collected data sets. This would indicate if the findings here were applicable to a broader class of systems, and facilitate analysis of the capabilities of the networks and learning algorithms to model systems when presented with noisy, real data.

REFERENCES

Åström, K. J. and Wittenmark, B., *Adaptive Control*. Reading, MA: Addison-Wesley Publishing Company, 1989.

Cohen, B. C., Saad, D., and Marom, E., "Efficient training of recurrent neural network with time delays," *Neural Networks*, 10(1), 51, 1997.

Dickinson, B. W., *Systems: Analysis, Design, and Computation*. Englewood Cliffs, NJ: Prentice-Hall, 1991.

Funashi K.-I. and Nakamura, Y., “Approximation of dynamical systems by continuous time recurrent neural networks,” *Neural Networks*, 6, 801, 1993.

Hagner, D. G., *Experimental Comparison of Recurrent Neural Network Architectures and Training Algorithms for Trajectory Generation*, Master’s Thesis, Department of Electrical and Computer Engineering, Wayne State University, Detroit, 1999.

Hassoun, M. H., *Fundamentals of Artificial Neural Networks*. Cambridge, MA: MIT Press, 1995.

Haykin, S., *Neural Networks: A Comprehensive Foundation*. Upper Saddle River, NJ: Prentice Hall, 1994.

Haykin, S., *Adaptive Filter Theory.* Upper Saddle River, NJ: Prentice Hall, 1996.

Horne, B. G. and Giles, C. L., “An experimental comparison of recurrent neural networks,” in *Advances in Neural Information Processing Systems* 7, 1994, Tesauro, G., Touretzky, D., and Leen, T., Eds. Cambridge, MA: MIT Press, 697, 1995.

Lin, D.-T., Dayhoff, J. E., and Ligomenides, P. A., “Trajectory production with the adaptive time-delay neural network,” *Neural Networks*, 8(3), 447, 1995.

Logar, A. M., Corwin, E. M., and Oldham, W. J. B., “A comparison of recurrent neural network learning algorithms,” in *International Joint Conference on Neural Networks*, San Francisco, 1129, 1993.

Makram-Ebeid, S., Sirat, J.-A., and Viala, J.-R., “A rationalized error back-propagation learning algorithm,” in *International Joint Conference on Neural Networks*, Washington, 2, 373, 1989.

Pearlmutter, B. A., “Gradient calculations for dynamic recurrent neural networks: a survey,” *IEEE Trans. Neural Networks*, 6(5), 1212, 1995.

Press, W. H., Teukolsky, S. A., Vetterling, W. T., and Flannery, B. P., *Numerical Recipes in C, The Second Edition*. Cambridge, UK: Cambridge University Press, 1992.

Puskorius, G. V. and Feldkamp, L. E., “Neurocontrol of nonlinear dynamical systems with Kalman filter trained recurrent networks,” *IEEE Trans. Neural Networks*, 5(2), 279, 1994.

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., Vol. 1. Foundations, Cambridge, MA: MIT Press, 319, 1986.

Shah, S. and Palmieri, F. , “MEKA—a fast, local algorithm for training feedforward neural networks," in *International Joint Conference on Neural Networks*, San Diego, Vol. 3, 41, June 17-21, 1990.

Singhal, S. and Wu, L., “Training multilayer perceptrons with the extended Kalman algorithm,” in *Advances in Neural Information Processing Systems 1*, Denver, 1988, Touretzky, D. S., Ed. San Mateo, CA: Morgan Kaufmann, 133, 1989.

Söderström, T. and Stoica, P., *System Identification*. New York: Prentice Hall, 1989.

Sundareshan, M. K. and Condarcure, T. A., “Recurrent neural-network training by a learning automaton approach for trajectory learning and control system design,” *IEEE Trans. Neural Networks*, 9(3), 354, 1998.

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

Tsung, F.-S. and Cottrell, G. W., “Phase—space learning,” in *Advances in Neural Information Processing Systems 7*, Tesauro, G., Touretzky, D., and Leen, T., Eds. Cambridge, MA: MIT Press, 481, 1995.

Werbos, P. J., “Backpropagation through time: what it does and how to do it”, *Proceedings of the IEEE*, 78(10), 1550, Oct., 1990.

Williams, R. J., “Training recurrent networks using the extended Kalman filter,” in *International Joint Conference on Neural Networks*, Baltimore, Vol. 4, 241, 1992.

(a) Williams, R. J. and Zipser, D., “A learning algorithm for continually running fully recurrent neural networks,” *Neural Computation*, 1, 270, 1989.

(b) Williams, R. J. and Zipser, D., “Experimental analysis of the real-time recurrent learning algorithm,” *Connection Science*, 1(1), 1989.

Williams, R. J. and Zipser, D., “Gradient-based learning algorithms for recurrent networks and their computational complexity,” in *Backpropogation: Theory, Architectures, and Applications*, Chauvin, Y. and Rumelhart, D. E., Eds., Hillsdale, NJ: Lawrence Erlbaum Associates, 433, 1995.

van der Smagt, P. P., “Minimization methods for training feedforward neural networks,” *Neural Networks*, 7(1), 1, 1994.

Van De Vegte, J., *Feedback Control Systems*. Englewood Cliffs, NJ: Prentice-Hall, 1986.