Spurious Valleys in the Error Surface of Recurrent Networks—Analysis and Avoidance

Jason Horn, *Member, IEEE*, Orlando De Jesús, *Member, IEEE*, and Martin T. Hagan, *Member, IEEE*

***Abstract*—This paper gives a detailed analysis of the error surfaces of certain recurrent networks and explains some difficulties encountered in training recurrent networks. We show that these error surfaces contain many spurious valleys, and we analyze the mechanisms that cause the valleys to appear. We demonstrate that the principle mechanism can be understood through the analysis of the roots of random polynomials. This paper also provides suggestions for improvements in batch training procedures that can help avoid the difficulties caused by spurious valleys, thereby improving training speed and reliability.**

***Index Terms*—Backpropagation, error surface, recurrent neural networks, spurious minima, spurious valleys, training.**

I. Introduction

RECURRENT neural networks have been applied successfully in the identification and control of dynamic systems [18], prediction in financial markets [32], channel equalization in communication systems [12], phase detection in power systems [27], sorting [24], fault detection [6], speech recognition [13], [16], [31], handwriting recognition [17], learning of grammars in natural languages [29], and even the prediction of protein structure in genetics [14]. However, even though these networks have been widely used, the difficulty of recurrent network training has limited their widespread application [3], [9], [20]–[22].

One of the difficulties in training recurrent networks is the existence of spurious local minima in the error surface. It has been known for many years that even the error surfaces of multilayer feedforward networks can have local minima. Sontag and Sussman [33] showed that even networks without hidden layers can have such spurious minima. They considered pattern recognition problems, in which sigmoid transfer functions were used. Bianchini *et al.* [5] discussed the problem of local minima in recurrent neural networks. They restricted their analysis to the case of recurrent networks used for recognition of "frames." The networks that they considered also used sigmoid transfer functions. Their analysis showed how the network architecture and the learning environment both contributed to the complexity of the error surface. They showed that if the network architecture and the learning environment satisfy certain "recurrent network assumptions," then the error surface contains no local minima. However, these conditions for optimal learning are only sufficient, and satisfying the criteria may require networks with large input size based on the unfolding in time of the neural network. More recently, Gori and Sperduti [15] developed sufficient conditions which guarantee the absence of local minima of the error function in the case of learning directed acyclic graphs with recursive (related to recurrent) neural networks. They developed a method for designing a neural architecture with a local-minima-free error function for a given data set. As in previous work, their networks used sigmoid transfer functions and performed pattern recognition tasks.

There have been other approaches to recurrent network training that involve selecting the initial weights so that the chance of falling into a local minimum is minimized. For example, Wang and Chen [38] describe an automated procedure that combines minimal model determination, weight initialization, and performance optimization. This technique is designed for a specific network architecture that is used for dynamic system identification. Huang *et al.* [23] discuss the problem of local minima in recurrent networks and propose an efficient structure and parameter learning algorithm for the Jordan network. A key step in their procedure is a good initial guess for the network weights. Xiao *et al.* [39] propose a two-stage training process. In the first stage, particle swarm optimization is used to locate an initial guess that will speed network convergence. In the second stage, a backpropagation algorithm is used to train the network to convergence. All of these papers use weight initialization to attempt to avoid local minima in recurrent network error surfaces, but they do not explain why the minima occur.

This paper will focus on recurrent networks that are used for system identification, control, filtering, prediction, and related tasks, which involve sequence processing and produce continuous outputs. The concepts discussed here apply to arbitrary recurrent network architectures, although we will fully investigate only simple networks. We will demonstrate that the error surfaces of recurrent networks have spurious valleys, which can disrupt in a significant way the training of recurrent networks. We suggest a newly discovered mechanism that can explain, at least in part, the cause of spurious valleys in the error surfaces of recurrent networks. We show that this mechanism can even produce spurious valleys in a simple recurrent network with a

Manuscript received March 21, 2008; revised July 16, 2008 and October 06, 2008; accepted November 10, 2008. First published March 06, 2009; current version published April 03, 2009.

J. Horn is with the Agilent Technologies High Frequency Technology Center, Santa Clara, CA 95051 USA (e-mail: jason@jasonhorn.com).

O. De Jesús is with the Research Department, Halliburton Energy Services, Dallas, TX 75006 USA (e-mail: Orlando.DeJesus@Halliburton.com).

M. T. Hagan is with the School of Electrical and Computer Engineering, Oklahoma State University, Stillwater, OK 74078 USA (e-mail: mhagan@okstate.edu).

Color versions of one or more of the figures in this paper are available online at http://ieeexplore.ieee.org.

Digital Object Identifier 10.1109/TNN.2008.2012257

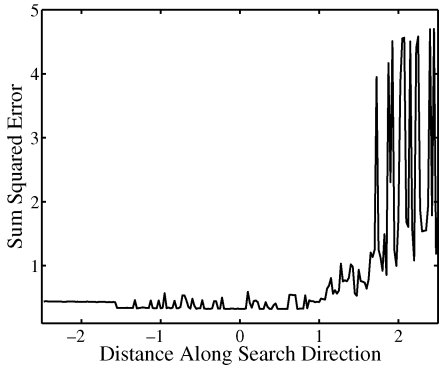


Fig. 1. Error profile.

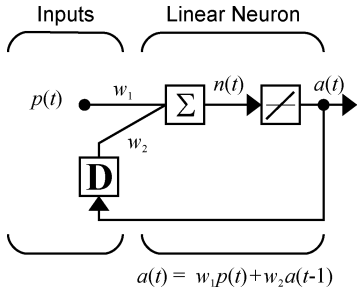


Fig. 2. First-order linear recurrent network.

linear transfer function and a single neuron. To our knowledge, this has not been previously reported in the literature. Based on our analysis of this mechanism, we will also propose modified training procedures that can provide improved convergence. We will demonstrate the operation of these modified training procedures on two simple recurrent networks.

II. Prelude

We begin with a description of how we encountered spurious valleys in the error surfaces of recurrent networks. While training a neural-network-based model reference controller [10], we found that the error sometimes increased during training, although a line search minimization was being executed at each iteration. In order to understand the failure of the line search, we plotted the error surface along the search direction. A typical profile is shown in Fig. 1. For the system shown, we have 65 weights being trained. The surface we present is along the direction of search [obtained by the Broyden–Fletcher–Goldfarb–Shanno (BFGS) quasi-Newton algorithm] through a 65-dimensional space. It is clear from this profile that any standard line search, using a combination of interpolation and sectioning, will have great difficulty in locating the minimum along the search direction. There are many local minima contained in very narrow valleys. (Some of the valleys were found to have widths on the order of 10^{-10}.) In addition, the bottom of the valleys are often cusps. (The neural network function is continuous and infinitely differentiable, so theoretically no cusps can exist. In practice, however, the valleys are so narrow that they appear as cusps on the domain of double-precision numbers, and therefore, they are effectively cusps for most training and analysis purposes.) We normally assume that the minimum will occur at the point where the derivative is zero. However, for some of these valleys, the derivative continues to increase as we approach the minimum. Even if our line search were to locate the minimum, it is not clear that the minimum represents an optimal weight location. In fact, in the remainder of this paper, we will demonstrate that spurious minima are introduced into the error surface due to characteristics of the input sequence.

In order to understand how spurious valleys can appear in the error surface, we analyzed the surfaces for some very simple recurrent networks. The idea was to find the simplest network that would produce the valleys. In the next section, we discuss a first-order linear recurrent network that produces spurious valleys. This is followed, in Section IV, with a theoretical analysis of the mechanism that causes the valleys. In Section V, we will show how adding nonlinear transfer functions can affect the shape of existing spurious valleys and generate new valleys. This is followed in Section VI by some modifications we propose to improve the training process, based on our analysis of the creation of the spurious valleys. Section VII of the paper tests the proposed modifications on first- and second-order recurrent networks. In the last section, we give a summary of the results.

III. First-Order Linear Recurrent Network

Fig. 2 illustrates the simplest possible recurrent network. As we will see, even this network produces spurious valleys similar to those shown in Fig. 1.

In order to generate an error surface, we first develop training data using the network of Fig. 2, where the weights are set to $w_1 = 0.5$ and $w_2 = 0.5$. We use a Gaussian white noise input sequence with mean zero and variance one for $p(t)$, and then use the network to generate a sequence of outputs $a(t)$. (When using recurrent networks to model dynamic systems, it is a common practice to use random, or pseudorandom, input sequences to generate the training data.) Our training objective is then to train another network with the same architecture to fit the training data. The global minimum of the error surface (sum square error over the training data) should occur at the values $w_1 = 0.5$ and $w_2 = 0.5$.

The left-hand side of Fig. 3 is a typical error surface obtained using the above procedure for one particular input sequence and the initial output $a(0) = 0$. The right-hand side of Fig. 3 indicates where the valleys occur. Although this network architecture is simple, the error surfaces generated by these networks have spurious valleys similar to those encountered in more complicated networks.

There are several interesting features of the surface. First, the error surface generally increases dramatically as the weight w_2 becomes larger than 1 in magnitude. This is to be expected, since the network is unstable for these weight values. What is unexpected are the two valleys that run through the surface. Even though the network is unstable for $|w_2| > 1$, for this particular input sequence, there are some values for w_2 in the unstable range that produce small network outputs (and, therefore, relatively small errors). We expect the output to grow without bound under these conditions, but this does not always happen.

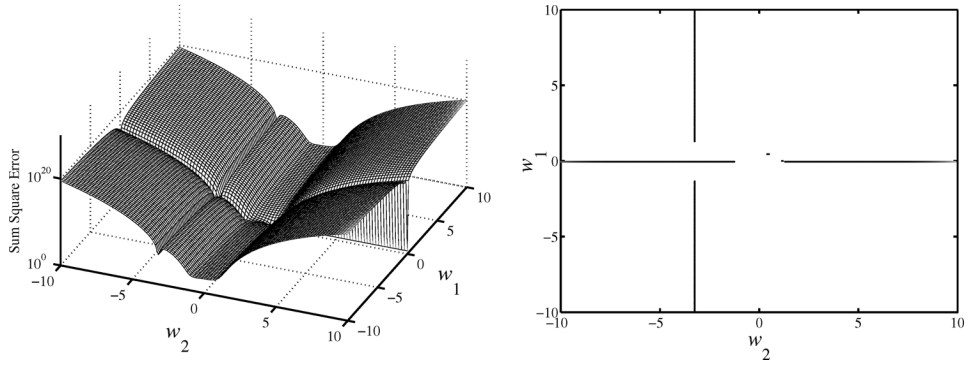


Fig. 3. Error surface (log scale) and valleys for first-order linear network.

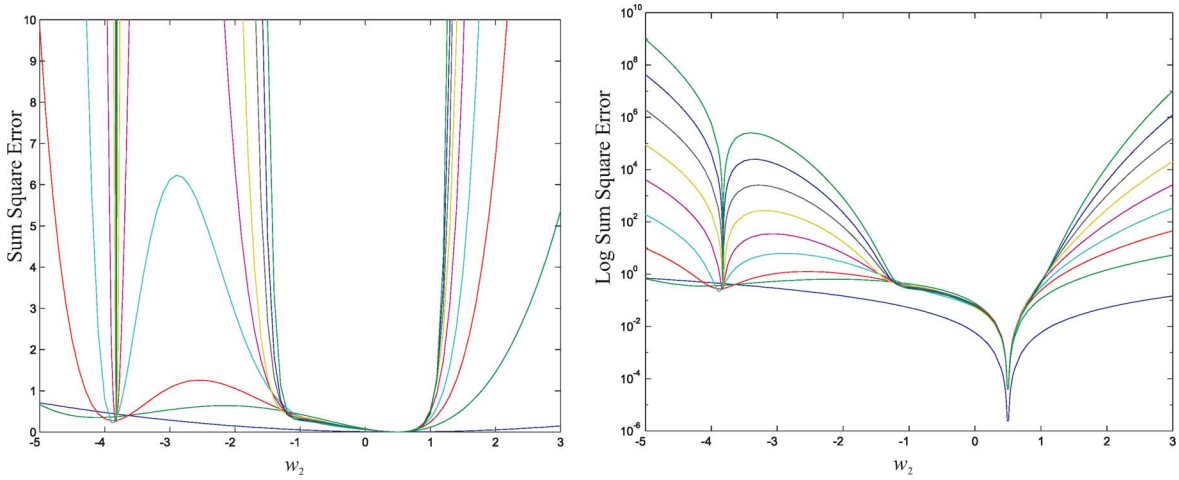


Fig. 4. Sum square error cross sections for $w_1 = 0.5$ for different values of sequence length k.

The two valleys in the error surface occur for two different reasons. One valley occurs along the line $w_1 = 0$. If this weight is zero, and the initial condition is zero, the output of the network will remain zero, no matter what value is used for w_2. Therefore, our mean squared error will be constant and equal to the mean square value of the target outputs.

The second, and more interesting, valley in the error surface is due to the input sequence that is presented to the network. For a given input $p(k)$, the system output will be

$$a(k) = w_1 p(k) + w_2 a(k-1). \tag{1}$$

If we accumulate the responses starting from some initial condition $a(0)$ up to time $k > 0$, we obtain

$$\begin{aligned} a(1) &= w_1 p(1) + w_2 a(0) = w_1 p(1) \\ a(2) &= w_1 p(2) + w_2 a(1) \\ a(2) &= w_1 p(2) + w_2 w_1 p(1) + {w_2}^2 a(0) \\ &\vdots \\ a(k) &= w_1 \{p(k) + w_2 p(k-1) + \cdots + {w_2}^{k-1} p(1)\} + {w_2}^k a(0). \end{aligned} \tag{2}$$

Here we can see that the response at time k is a polynomial in the parameter w_2. (It will be a polynomial of degree $k-1$, if the initial condition is zero.) The coefficients of the polynomial involve the input sequence and the initial condition. We obtain the second valley because this polynomial contains a root outside the unit circle. There is some value of w_2 that is larger than 1 in magnitude for which the output $a(k)$ is almost zero.

Of course, having a single output close to zero would not produce a valley in the error surface. However, we discovered that once the polynomial shown in (2) has a root outside the unit circle at time k, that same root also appears in the next polynomial at time $k+1$, and therefore, the output will remain small for all future times for the same weight value. The theoretical mechanism for the frozen root will be analyzed in Section IV.

Fig. 4 shows a cross section of the error surface presented in Fig. 3 for $w_1 = 0.5$ using different sequence lengths. The error falls abruptly near -3.8239. That is the root of the polynomial described in (2). The root maintains its location as the sequence increases in length (k increases). This causes the valley in the error surface.

To summarize, there are two mechanisms that create the spurious valleys. The first mechanism has to do with the initial conditions. If some initial conditions are zero, then there are certain combinations of weights that will produce zero outputs for all time. (This effect is more complex in larger networks, as we will see in Section VII.) The second mechanism has to do with the input sequence. There are values for the weights that produce an unstable network, but for which the output remains small for a particular input sequence. If the input sequence is modified, it will produce a valley in a different location.

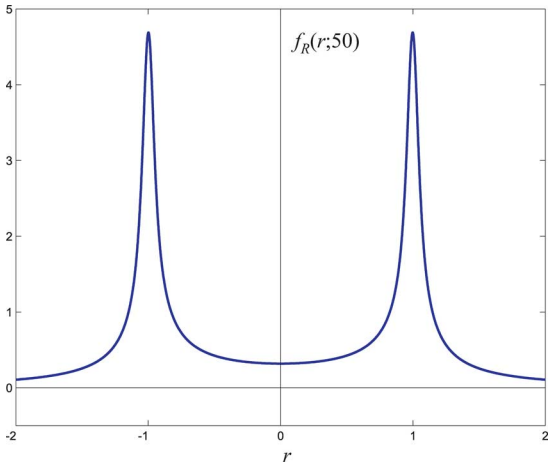


Fig. 5. Density of real roots of a random polynomial of degree 50.

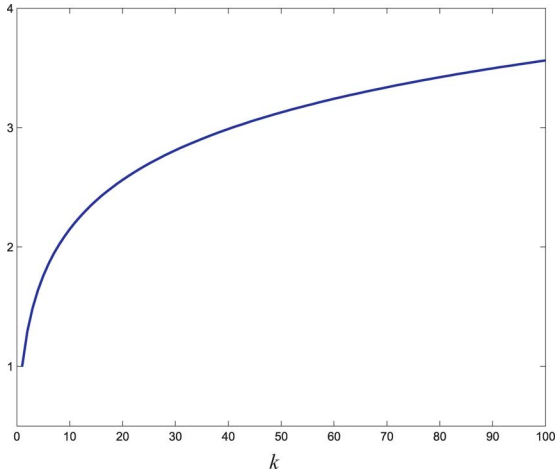


Fig. 6. Expected number of real roots of a RGP as a function of k.

Comprehensive prediction of valleys for this first-order linear recurrent network is relatively simple since only two types of valleys are encountered. We predict a valley where w_1 is equal to zero, and we predict a valley where w_2 is equal to a real root of the polynomial that has the input sequence as its coefficients. If the value of the root is not greater than one (producing an unstable system), then the output will not be significantly higher for values on either side of the root than for values at the root, so the valley will not occur.

In the following section, we will investigate the theoretical mechanisms for the spurious valley caused by the root of the polynomial in (2) outside the unit circle.

IV. Roots of Random Polynomials

One of the keys to the locations of the spurious valleys in the error surface of recurrent networks are the roots of polynomials, as in (2). In this section, we will investigate these polynomial roots in more detail. First, assume that we have a polynomial in the following form [4], [25]:

$$g(w;k) = c_0 + c_1 w + \cdots + c_k w^k = \sum_{i=0}^{k} c_i w^i. \qquad (3)$$

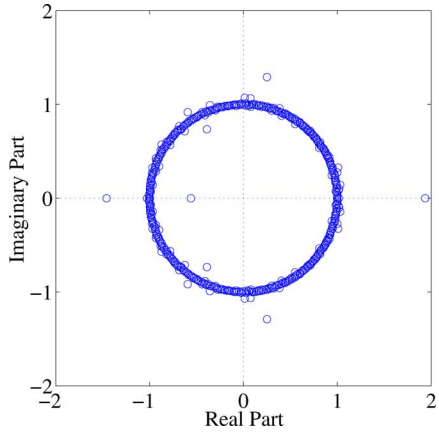


Fig. 7. Distribution of roots of a 500th-order RGP.

We will analyze such polynomials where the coefficients are independent random variables that have a Gaussian distribution with mean zero and variance one. We will call these polynomials random Gaussian polynomials (RGP). The general patterns that we will demonstrate are not extremely sensitive to the exact distribution of the coefficients, but the Gaussian assumption allows a clear development.

We are especially interested in the roots of RGPs that are real and that fall outside the unit circle. In that case, for the polynomial in (2), the network output will be zero for some feedback gain that is greater than one in magnitude, which would normally produce a large output. There are several things that we can say about the real roots of (3). First, the probability of getting a real root in the interval $[1, \infty]$ is the same as the probability of getting a real root in the interval $[0, 1]$. This is because the polynomial with a root at r can be converted to a polynomial with a root at $w = 1/r$ by reversing the order of the coefficients. Under our previous assumptions, these two polynomials would have equal probability of occurrence. In addition, the probability of getting a positive real root will be equal to the probability of obtaining a negative real root. This is because the polynomial with a root at $w = r$ can be converted to a polynomial with a root at $w = -r$ by changing the signs of the coefficients at odd powers of w. To summarize, the real roots of a RGP are equally likely to fall in any of the following four intervals: $[-\infty, -1]$, $[-1, 0]$, $[0, 1]$, and $[1, \infty]$. This means that half of the real roots of a RGP are likely to fall outside the unit circle.

Kac [26] derived the density for the real zeros of the RGP in (3)

$$f_R(r;k) = \begin{cases} \dfrac{1}{\pi}\sqrt{\dfrac{1}{(r^2-1)^2} - \dfrac{(k+1)^2 r^{2k}}{(r^{2k+2}-1)^2}}, & r \neq \pm 1 \\ \dfrac{1}{\pi}\sqrt{\dfrac{k(k+2)}{12}}, & r = \pm 1. \end{cases} \qquad (4)$$

This is plotted in Fig. 5 for the case where $k = 50$. This function should be divided by the area under the curve to obtain the conditional probability function for a root, given that it is real.

In addition to knowing the probable locations of the real roots of RGPs, we would also like to know how many of the roots will

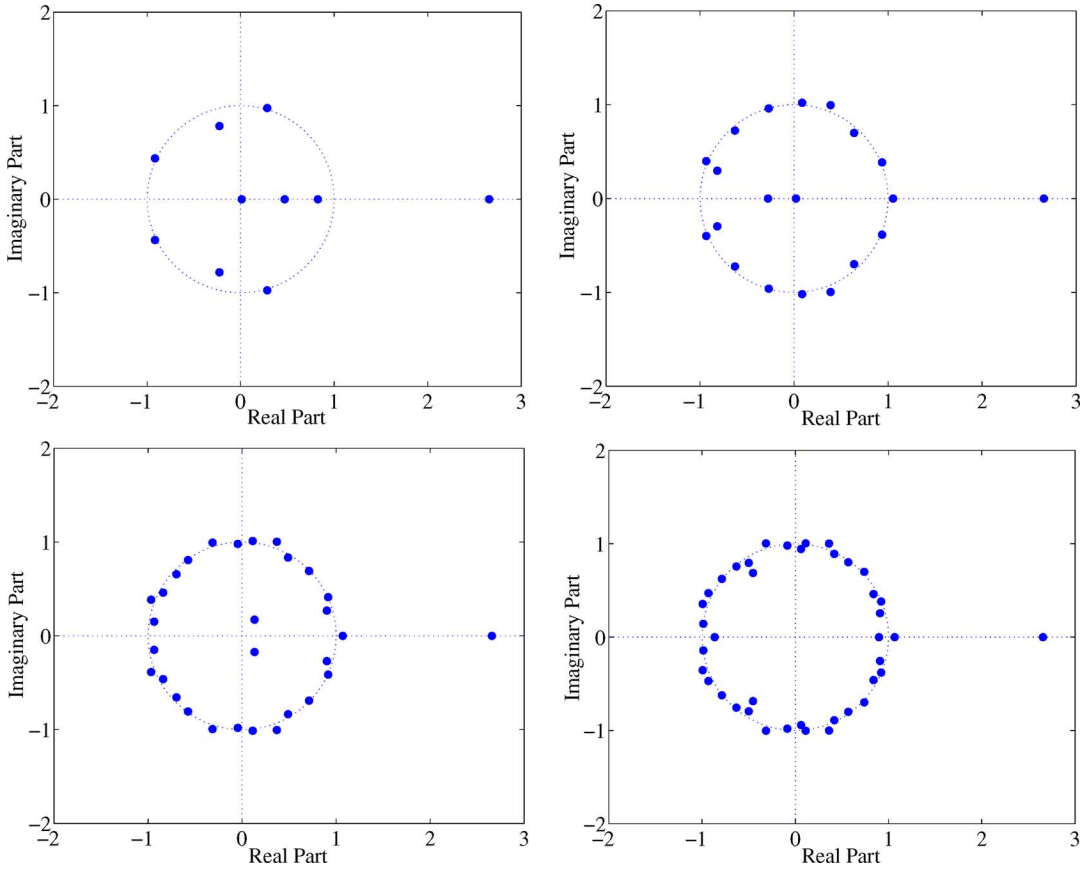


Fig. 8. Movement of roots as order is increased ($k = 10, 20, 30, 40$).

be real. The expected number of real roots can be obtained from the integral of (4) over the real line. This is shown in Fig. 6 as k varies from 1 to 100.

As can be seen from Fig. 6, the number of real roots is small. Kac [25] showed that the number of real roots goes up as the log of the order of the polynomial, as is seen in the following strict upper bound:

$$\frac{2}{\pi}\log(k) + \frac{14}{\pi}. \tag{5}$$

The number of real roots goes up fairly rapidly as the order is initially increased, but then the rate of increase diminishes quickly. This means that you are likely to have a few real roots, even if the order of the polynomial is small, but the number of real roots does not increase significantly as the order increases.

For the RGP roots that are not real, Bharucha-Reid [4] has shown that they are distributed near the unit circle. This can be seen in Fig. 7, which shows the roots of a 500th-order RGP. We can see that most of the roots are complex, and are heavily concentrated near the unit circle. There are several real roots, and one is located well outside the unit circle, near 2. Note that the distribution of (4) and Fig. 5 has heavy tails (decays slower than the exponential distribution), which means that there is a significant probability of having roots well outside the unit circle.

Note that in our recurrent network response of (2) the order of the polynomial increases with time. We have found through numerous experiments that if a real root of the polynomial falls well outside the unit circle (e.g., with a magnitude of 2), then the root maintains its location, even as time (and, therefore, the order of the polynomial) is increased. (As mentioned in the previous section, this causes a spurious valley in the error surface, as shown in Figs. 3 and 4.) The effect is demonstrated in Fig. 8, which shows the movement of the roots of a polynomial as the order of the polynomial is increased. Note that when the order is 10, there is one root at approximately 2.66. This root maintains its location as the order of the polynomial is increased, while the other roots move toward the unit circle.

To investigate why this root is frozen, let us assume that the polynomial of (3) has a root at w_0. Consider the displacement of the root δw, due to a perturbation in the coefficient c_j of δc_j[11]

$$g^*(w_0 + \delta w; k) = \sum_{i=0}^{k} c_i (w_0 + \delta w)^i + \delta c_j (w_0 + \delta w)^j \tag{6}$$

where $g^*()$ is the polynomial of (3) with the perturbed coefficient. This expression is equal to zero, because $w_0 + \delta w$ is a root of the modified polynomial. Note that the first term (the summation) on the right-hand side of (6) corresponds to $g(w_0 + \delta w; k)$, therefore we can write

$$g(w_0 + \delta w; k) = -\delta c_j (w_0 + \delta w)^j. \tag{7}$$

Now perform a Taylor series expansion of each side of (7)

$$g(w_0 + \delta w; k) = \sum_{i=1}^{k} \frac{1}{i!} g(w_0; k)^{(i)} (\delta w)^i \tag{8}$$

where the $i = 0$ term is missing from the summation because w_0 is a root of $g(w;k)$, and

$$-\delta c_j(w_0+\delta w)^j = -\delta c_j\left\{{w_0}^j + j{w_0}^{j-1}\delta w + \cdots + \frac{j(j-1)\cdots(j-k-1)}{k!}{w_0}^{j-k}(\delta w)^k\right\}. \tag{9}$$

If we now set (8) equal to (9) and take the limit as δc_j, and therefore, also δw, go to zero, we find

$$g(w_0;k)^{(1)}\delta w = -\delta c_j {w_0}^j \tag{10}$$

or

$$\frac{\delta w}{\delta c_j} = -\frac{{w_0}^j}{g(w_0;k)^{(1)}}. \tag{11}$$

This tells us the sensitivity of the root location, as a function of one of the coefficients in the polynomial. It is related to the condition of the polynomial, which is defined in [11].

To relate this result to the recurrent network response of (2), we will first assume that $a(0) = 0$ and $w_1 = 1$. (This will simplify the development without changing the overall conclusions.) The resulting network response will be

$$\begin{aligned} a(k+1) &= p(k+1) + w_2 p(k) + \cdots + {w_2}^k p(1) \\ &= \sum_{i=0}^{k} p(k+1-i)\,{w_2}^i. \end{aligned} \tag{12}$$

If we equate this expression with (3), we see that

$$c_i = p(k+1-i). \tag{13}$$

Now consider the coefficient $c_0 = p(k+1)$. This is the last input to come into the network, and it increases the order of the polynomial by 1. When $c_0 = p(k+1) = 0$, all previous roots are unchanged. The sensitivity of a previous root to changes in this coefficient is given by (11)

$$\frac{\delta w}{\delta c_j} = -\frac{{w_0}^0}{g(w_0;k)^{(1)}} = -\frac{1}{g(w_0;k)^{(1)}}. \tag{14}$$

The denominator in this term is the first derivative of the polynomial $g(w;k)$, evaluated at the root w_0

$$g(w_0;k)^{(1)} = \sum_{i=1}^{k} c_i \cdot i \cdot {w_0}^{i-1}. \tag{15}$$

If the coefficients are random with mean zero and variance 1, then this term has variance given by

$$\operatorname{var}\left(g(w_0;k)^{(1)}\right) = \sum_{i=1}^{k} i^2 \cdot {w_0}^{2(i-1)}. \tag{16}$$

If the root w_0 is greater than 1 in magnitude, then this variance will be very large even for moderate values of k. This means that it is highly likely that $g(w_0;k)^{(1)}$ will be very large, and, based on (14), that any root w_0 that is greater than 1 in magnitude will not change significantly when the order of the polynomial is increased. Therefore, any root of (2) with magnitude greater than 1 will be frozen in place as time is increased. This is exemplified by Figs. 4 and 8, as well as by many other experiments that we have performed.

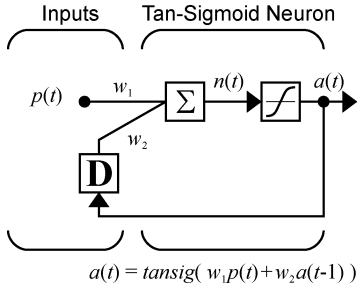


Fig. 9. First-order nonlinear recurrent network.

Here are the key results of this section that are most relevant to the error surfaces of recurrent networks: 1) the roots of a RGP are very likely to have some real roots that are greater than 1 in magnitude, and 2) if a RGP does have a root that is larger than 1 in magnitude, that root will maintain its location as the order of the polynomial is increased. These results explain one cause of the spurious valleys that appear in the error surfaces of recurrent networks. In the next section, we will demonstrate how using a nonlinear transfer function will increase the number and complexity of the resulting spurious valleys.

V. First-Order Nonlinear Recurrent Network

In addition to the first-order linear recurrent network, we analyzed the error surface of a first-order nonlinear network, illustrated in Fig. 9. It is a simple extension of the linear network of Fig. 2, in which a sigmoid nonlinearity replaces the linear transfer function.

Fig. 10 presents the error surface for the nonlinear network, using the same input sequence used in Section III. Due to the sigmoid nonlinearity, the output is bounded for large weight values. Therefore, the error does not grow without bound, as in the linear network. We notice that the valley is still present, however it is bent. This curving valley is still able to trap the training algorithm and even to move the weights away from the true minimum. In addition, several new valleys appear. As you can see, the addition of the nonlinearity to the network significantly complicates the error surface.

Four types of valleys were identified in the error surfaces of the nonlinear recurrent network. Like the linear network, a valley appears along the line $w_1 = 0$. The cause of this valley is the same in the nonlinear case as it is in the linear case: if the initial condition $a(0)$ is zero, then the output of the network will be zero for all values of w_2 when $w_1 = 0$. The sum square error is, therefore, limited to the sum square target values. The other three types of valleys differ from those encountered in the linear network, although they are related to the roots of a polynomial.

Equation (17) gives the output equation for the nonlinear recurrent network

$$a(k) = \operatorname{tansig}(w_1 p(k) + w_2 a(k-1)). \tag{17}$$

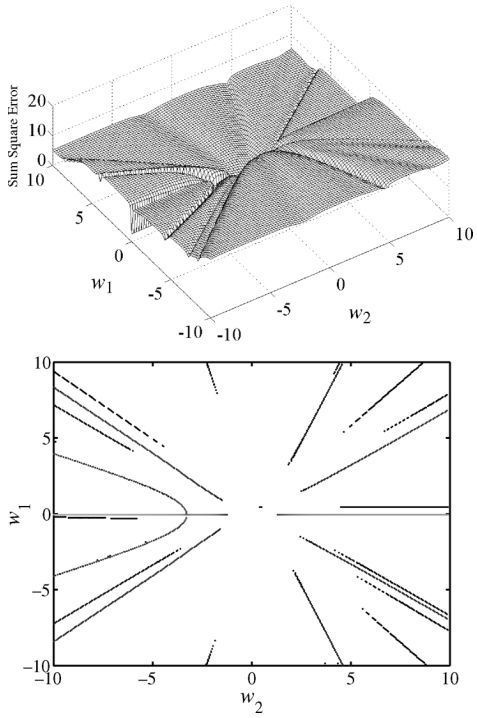


Fig. 10. Error surface and valleys for first-order nonlinear network.

For small values of the weights, the argument of the *tansig* function is small, and this equation reduces to that of the linear recurrent network. It is, therefore, not surprising that the valleys that were seen in the linear case, related to the roots of the polynomial of (2), appear also in the nonlinear case for small values of w_1. As the value of w_1 increases, however, the nonlinearity causes the behavior of the valley to differ from the linear case. In Fig. 3, we can see a valley that occurs at $w_2 = 3.8239$, which corresponds to the root of the polynomial. In Fig. 10, we see a valley that starts at $w_2 = 3.8239$ for small values of w_1, but then curves to the left as w_1 increases in magnitude.

As soon as w_1 reaches a certain threshold, which can be determined by the magnitude of the terms in the input sequence and the magnitude of w_2 at which the valley occurs, the output begins to saturate at a certain point in time as the network leaves the linear operating region of the *tansig* function. The output at this point in time saturates and begins to approach a value of $+1$ or -1. The output for all following points in time, which are still small enough to be operating in the approximately linear region of the *tansig* function, can be approximated by

$$a(t+k) \approx w_1\{p(t+k) + w_2 p(t+k-1) + \cdots + {w_2}^{k-1} p(t+1)\} + {w_2}^k a(t). \quad (18)$$

We again have a polynomial in the parameter w_2, but in this case, there is an additional term which does not include w_1. This means that the root of the polynomial is dependent on w_1, so as w_1 increases, the root will increase as well. The valley found in the nonlinear network follows the curve of this increasing root.

As w_1 increases further, more points in the input sequence will saturate and cause additional valleys. This brings us to the third type of valley, which occurs as w_1 and w_2 increase enough to cause saturation in the output of the network at most time

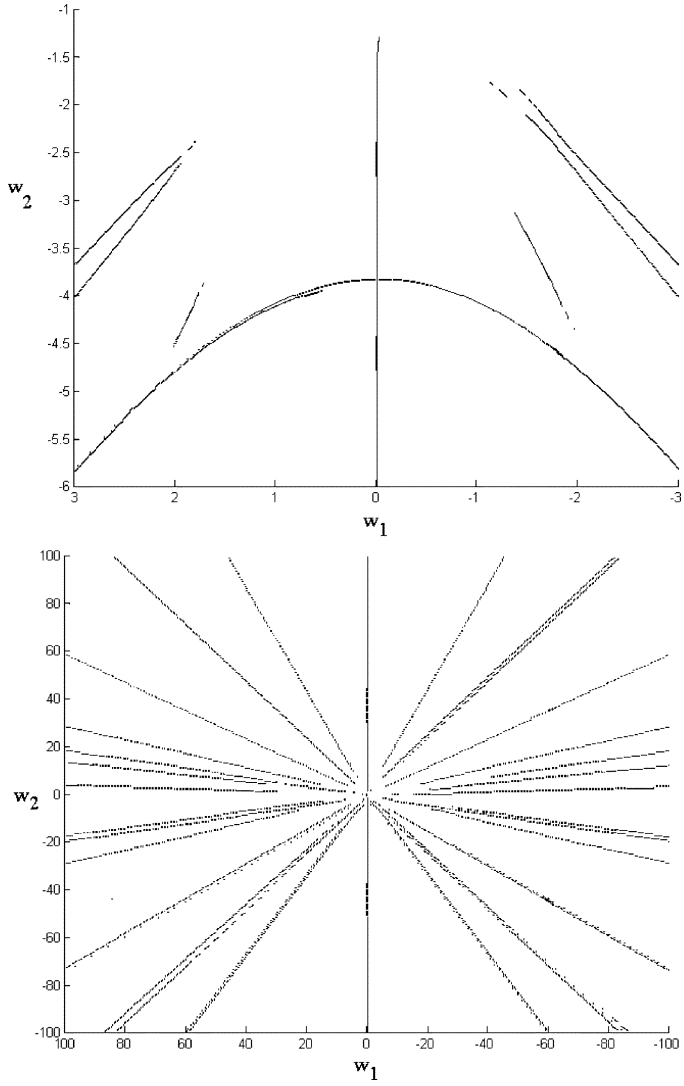


Fig. 11. Valleys in the error surface of the second and third type.

points. Fig. 11 illustrates both the second and third types of valleys.

When the output of the network at time $k-1$ is saturated near $+1$ or -1, then the output of the network at time k becomes a function of w_1 and w_2. For some combinations of weights, the output will be near $+1$, and for others it will be near -1. However, there is a transition point, while it is switching between positive and negative saturation, at which it will cross the desired network output. Because the outputs at all other time points are saturated and unchanging, changing this one output so that it is equal to the target output will cause a valley in the error surface. The possible locations of this valley can be found by substituting $+1$ or -1 for $a(k-1)$ and the desired output (or zero if this is not available) for $a(k)$ in the output equation of the network and solving for w_1 in terms of w_2. This gives us

$$w_1 = \frac{a_{\text{desired}}(k) \pm w_2}{p(k)}. \quad (19)$$

To make matters worse, a shift of the output at time $k-1$ between $+1$ and -1 also has the potential to cause the output at time k to shift between $+1$ and -1. This can cause another

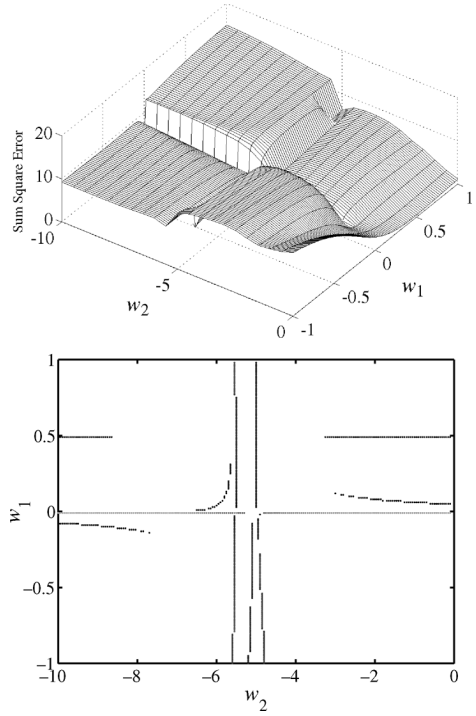


Fig. 12. Error surface and valleys for small w_1 (fourth type of valley) for the first-order nonlinear network.

valley, and also can cause the output at time $k+1$ to shift between $+1$ and -1. This cycle has the potential to continue until the final time point, so a single shift at time k in an input sequence of length n has the potential to cause $n-k+1$ valleys near the line given by (19).

Fortunately, not all of these potential valleys actually appear on the error surface. To determine which valleys will occur, a simple algorithm can be written which keeps track of the sign of the output at each time and determines which outputs will shift between $+1$ and -1 as w_1/w_2 varies from $-\infty$ to 0 and from 0 to ∞. Only the outputs that actually shift will cause valleys to occur. Using this method, we were able to predict accurately the locations of all valleys of this type that actually would occur for a given input/output sequence of training data.

The fourth and final type of valley that we identified and analyzed in the error surface of the nonlinear recurrent network is also related to the effects of saturation. When w_1 is small and w_2 is large, the output of the network for early points in time will be near zero. As time progresses, however, the power to which w_2 is raised increases and the output will eventually saturate to $+1$ or -1 depending on the input sequence as well as the signs of w_1 and w_2. Like the case where an output was switching between positive and negative saturation, the transition between zero and the saturated value may cause the output to equal the desired output for some combination of w_1 and w_2. All other points in the output sequence remain near zero or saturated near this combination, so a valley is formed. These valleys are illustrated in Fig. 12.

In order to predict the location of these valleys, a simplification of the output equation at time k is necessary. Because the output at all time points less than k is near zero, it can be assumed to operate in the linear region of the sigmoid function. The output equation at time k then reduces to that of the linear network, given in (1). For large values of w_2 and small values of w_1, the output will be approximately equal to the desired output when the following is satisfied:

$$w_1 = \frac{a(k)}{{w_2}^{k-1} p(1)}. \tag{20}$$

This equation was reached by solving for w_1 and eliminating all other terms, which are insignificant compared to $(w_2)^{k-1}$ because w_2 is large. This equation can be used to predict accurately and reliably all valleys of this type that will occur in the error surface for given set of training data.

Comprehensive prediction of all valleys that will occur in an error surface for the first-order nonlinear network can now be achieved by combining the prediction methods for each of the four types of valleys. The constraints mentioned for each type of valley must be followed, so valleys will not be predicted for real roots of the input sequence that have a magnitude less than one. Valleys predicted by (19) and (20) will only occur for large values of w_2.

Let us summarize the results of this section. We have analyzed the error surface of a single-neuron recurrent network with sigmoid transfer function. By simply replacing the linear transfer function with the sigmoid transfer function, we have greatly increased the number and complexity of the spurious valleys in the error surface. In fact, we have found four different types of spurious valleys that can occur in this simple nonlinear network. All of these valleys are related to the valleys that occur in linear recurrent networks, but they are much more numerous and complex because of the saturation of the sigmoid transfer function. We were able to develop techniques for completely predicting the locations of all the spurious valleys of this network, given knowledge of the training data set.

Up to this point, we have not carried out a detailed analysis beyond the single-neuron, nonlinear recurrent network. However, we have performed simulation experiments on a number of more complex networks. What is clear is that as the size of the network increases, the number and complexity of the spurious valleys increase as well. In addition, we can say that the locations of the valleys are dependent on the training data and the initial conditions of the network. If the training data and initial conditions are modified, the spurious valleys will be moved as well. In the next section, we will show how this knowledge can be used to modify training procedures to improve convergence.

VI. Modifications to the Training Procedure

From the previous sections, we see that difficulties in training recurrent neural networks could be due to the presence of spurious valleys. The shape of the valleys are complex for large nonlinear neural networks. If a gradient search algorithm falls inside a valley, we may converge to a region where the network is unstable or where the weights are unreasonably large. The location of those valleys depends on the input sequence and on the initial conditions. In this section, we will propose three modifications to standard training procedures that can mitigate the effects of the valleys.

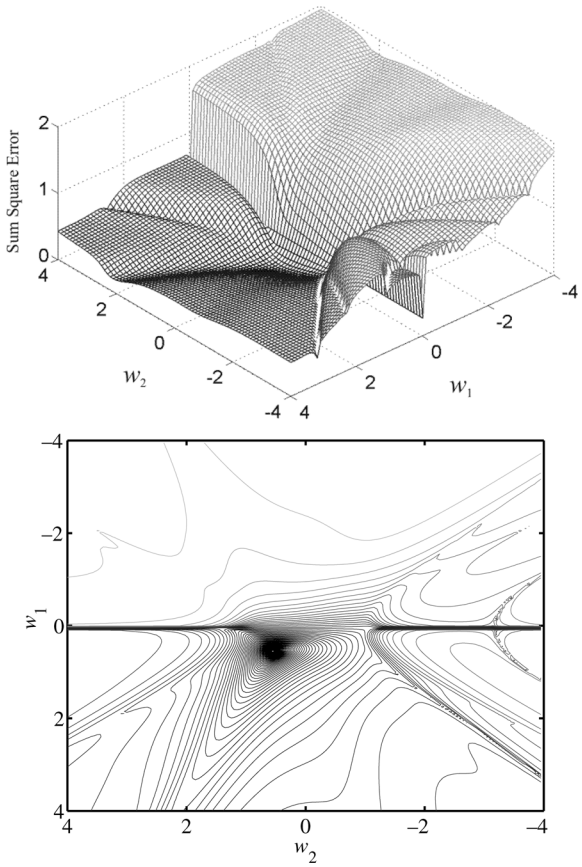


Fig. 13. Error surface for first-order nonlinear network for different input sequence.

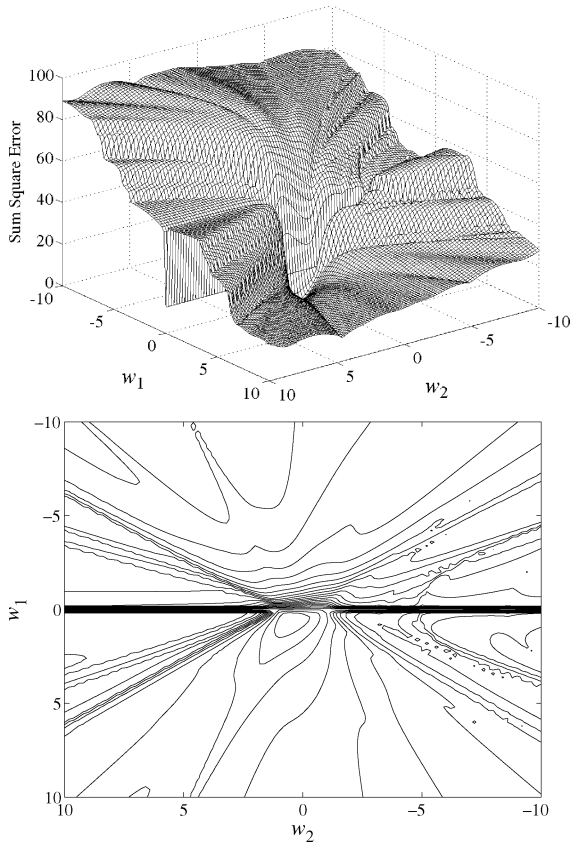


Fig. 14. Error surface using sequence averaging.

A. *Proposed Solutions*

In this section, we will propose three variations to the standard batch training algorithms for recurrent networks. These variations include regularization, switching training sequences, and randomly setting initial conditions.

If we compare the linear and nonlinear cases from Sections III and V, we notice that the linear case has a natural way of allowing convergence to the optimal weights, because larger weights generate large outputs. The farther we move from the stable region, the larger the gradient will become. A gradient-descent algorithm would generally move the weights toward the stable region. This effect does not occur in the nonlinear networks. However, we can obtain a similar effect if we combine regularization [30] with our mean square error performance function. In other words, we can use the performance function

$$\mathrm{J}(\mathbf{w}) = \mathrm{SSE} + \alpha \mathrm{SSW} \tag{21}$$

where SSE is the sum squared errors and SSW is the sum squared weights. This performance function would help to force the weights back into the stable region, because it would overwhelm the spurious valleys for large values of the weights. We can decrease the regularization factor α during training to ensure that we do not bias the final trained weights.

Another technique for improved training involves using more than one training sequence. Fig. 13 presents the error surface for the nonlinear network of Fig. 9, using a different training sequence. The valley that appeared in Fig. 10 has moved to a different region of w_2. For any two random input sequences, the valleys will appear in different locations.

This suggests another technique for improved training. We could use multiple training sequences. Because valleys are sequence dependent, we can use one sequence for a given number of epochs and then alternate to a new sequence. If we become trapped in a spurious valley, that valley will disappear when the new sequence is presented.

Another implementation of multiple sequences could be sequence averaging. We could compute the gradients for multiple sequences and then move in the direction of the average. Fig. 14 presents an average error surface for five sequences. This figure demonstrates how the spurious valleys are reduced in amplitude.

Another method to move the valleys is to use random initial conditions. Fig. 15 shows how the error surface is changed when we set the initial condition to $a(0) = 0.1$. The valley at $w_1 = 0$, which we discussed earlier, is missing. In later experiments with larger networks, we found that the valleys do not always disappear when nonzero initial conditions are used. They are often only moved to new locations. A better approach would be to use different small random initial conditions at different stages of training. We could switch the initial conditions in combination with the switching of sequences.

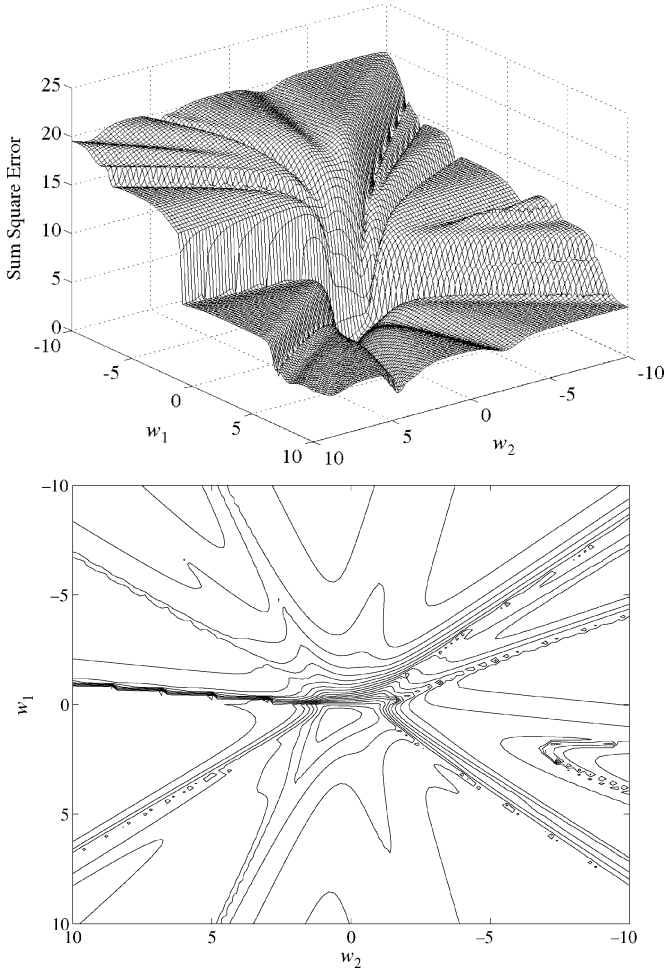


Fig. 15. Error surface using $a(0) = 0.1$.

In all, we have four proposed training modifications. For ease of reference, we will label them as follows: switching sequences (SS), averaging sequences (AS), regularization (REG), nonzero initial conditions (IC).

VII. Test Results

In this section, we will test the training modifications that were proposed in the previous section. For these tests, we will train the nonlinear network shown in Section V (and a more complex, second-order network) using the standard gradient-descent algorithm with a golden section line search. We will not worry about using the most sophisticated training algorithm. Rather, the objective will be to verify the ability of the new procedures to improve training performance. The results obtained with the basic gradient-descent algorithm will be our baseline. Other tests will be performed for each one of the proposed modifications. For the REG test, we divided α by 1.2 at each epoch. For the IC method, we set all layer initial conditions to 0.2. One test was performed using all three methods. We called this training procedure the "multiple" method. For all tests, the gradient is computed using the real-time recurrent learning method described in [7], [8], [19], and [34]–[36]. A batch gradient is used, which encompasses the full length of the training sequence.

TABLE I
Convergence Percentages for Single-Neuron Recurrent Network Training

Method	STD of the initial weights		
	1	5	20
Baseline	92.1	61.2	37.9
REG	99.6	99.7	99.9
SS	96.5	64.7	45.7
AS	94.3	58.1	42.7
IC	95.6	71.1	45.0
Multiple	100.0	100.0	100.0

TABLE II
Convergence Percentages for Two-Neuron Recurrent Network Training

Method	STD of the initial weights		
	1	5	20
Baseline	82.2	12.8	0.3
REG	93.0	95.0	97.0
SS	95.8	38.6	2.5
IC	54.6	7.2	0
Multiple	100.0	99.0	100.0

A. First-Order Nonlinear System

For the first-order nonlinear system, we generated training data using $w_1 = 0.5$ and $w_2 = 0.5$. The training was done using 25 000 different sequences of 15 samples each and random initial conditions. The random initial weights were generated in three different levels: 1, 5, and 20 standard deviations from the true solution.

Table I summarizes the results of the first tests on the first-order network. It shows the percentage of tests in which the weights converged close to the optimal weights. (For our tests, "close" is defined as a distance of 0.5 from the optimal weights. The results are not sensitive to small changes in this criterion, although training times are longer when smaller distances are used.) Each method provides some improvement on the baseline method. However, the multiple method is the only one that guarantees accurate convergence.

Fig. 16 shows the final (converged) weight positions in the w_2 versus w_1 plane for baseline, SS, AS, and IC. (Fig. 16 does not show results for the multiple method, because all converged weight positions were very close to the optimal weight position—$w_1 = 0.5$ and $w_2 = 0.5$.) For the first three methods, many tests finished along the valley at $w_1 = 0$. That condition was removed when we changed the initial conditions. When we switch the sequences, we avoid many cases where training may be trapped in the other spurious valleys. (In Fig. 16, only the valley caused by the zero initial condition is clearly displayed. This is because the other valleys are dependent on the input sequence, which was changed for each Monte Carlo trial.) The averaging of sequences did not improve our training results, resulting in worse results than the baseline method for 5 std.

B. Two-Layer Neural Network

Fig. 17 shows a neural network with two layers, where each layer is fed back to the previous layers. This system will allow us to test the previous training procedure modifications on a

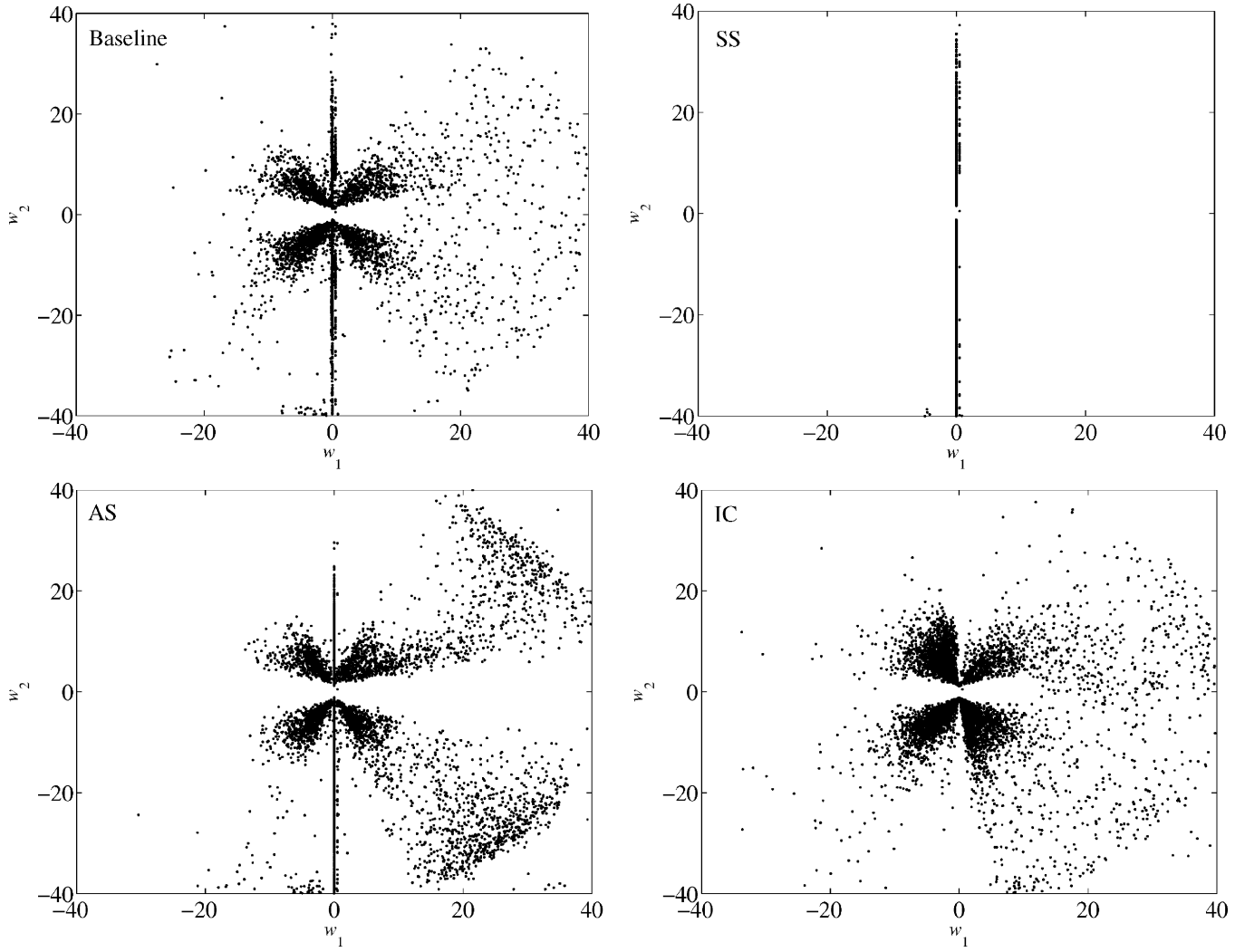


Fig. 16. Final weight positions in the w_2 versus w_1 plane for 5 std.

more complex system. For these tests, we generated training data using the following weights:

$$w_1 = 0.5 \quad w_2 = -0.5 \quad w_3 = 0.25 \quad w_4 = -0.3.$$

Table II shows the percentage of weights close to the final weights (within a distance of 0.5) after the training process. For this neural network architecture, regularization resulted in a success rate of over 90%. However, it is again the multiple method that guarantees the best convergence.

Fig. 18 presents the final weight positions in the w_4 versus w_3 plane for the baseline, SS, and IC training methods. For the baseline training method, we notice the presence of three valleys (due to zero initial conditions) where the training converged. The SS method can eliminate the diagonal valleys due to the initial conditions, as well as the valleys due to the input sequence. However, the valley along $w_3 = 0$ remains. When we set the initial layer conditions to 0.2, we can see from the last figure that although the valley at $w_3 = 0$ is removed, two new valleys appear. This demonstrates that setting the initial conditions to nonzero values does not necessarily remove spurious

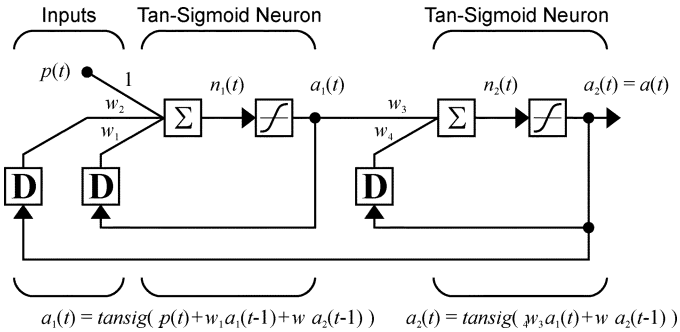


Fig. 17. Two-layer nonlinear model.

valleys. It may just move them to new locations. This suggests that we should vary the initial conditions whenever we switch the training sequence.

Fig. 19 shows how the final distance to the optimal weights is affected by the switching sequence interval. While training for 10 000 epochs, we switched the training sequence every 1, 10, 100, 500, and 1000 epochs. Frequent changes consistently resulted in more accurate final weights. If training continues

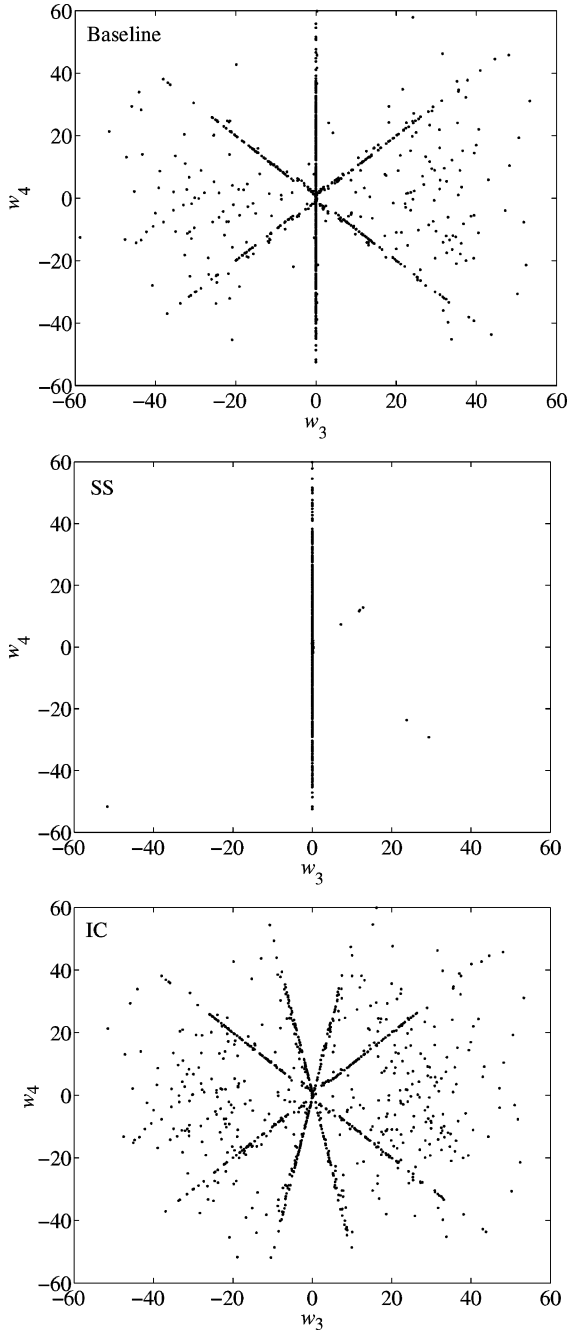


Fig. 18. Final weight positions in the w_4 versus w_3 plane for 20 std.

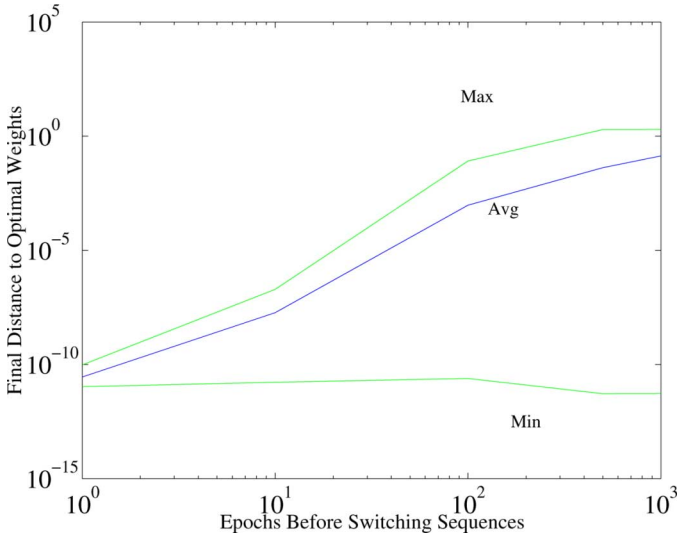


Fig. 19. Final distance to optimal weights for different switching sequence intervals.

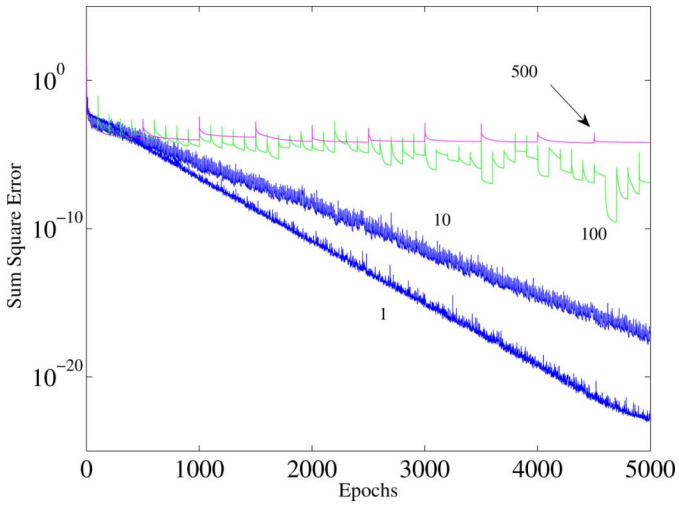


Fig. 20. Average performance for different switching sequence intervals (1, 10, 100, 500).

with the same sequence, we could be caught in a spurious valley, resulting in failed training.

Fig. 20 shows the average performance for four different switching intervals (the training sequence is switched every epoch, every 10th epoch, every 100th epoch, and every 500th epoch). We obtain substantial improvement when the sequence is switched more frequently. In fact, for this test problem, the best results were obtained when the sequence was switched after each epoch. We can conclude that we should not maintain the same sequence for long periods, when training recurrent neural networks.

Another battery of tests was performed to evaluate how to adjust α when regularization is being used. We adjusted α by dividing it by a constant at each epoch. In this way, the regularization penalty will not bias the final weights, but intermediate weights will be forced out of the regions with spurious valleys. The constants we used were 1.01, 1.2, and 2. Using a constant of 2 means that the regularization parameter α is reduced by half at each epoch. Fig. 21 shows the average performance when α is divided by 1.01 and 1.2 at each epoch. The best results were obtained for 1.2. (The results for 2 were almost identical to the results for 1.2.) From this test, we can conclude that α must be decreased in some way to obtain the best training results.

Fig. 22 shows the number of floating point operations (FLOPs) required to train the two-layer neural network to convergence using the multiple method with different sequence lengths. This figure does not demonstrate any advantage to using long sequences for this network. The algorithm converged for all sequences, but the longer sequences require more computation. One would expect that for more complex networks there might be some advantage to longer sequences, because it might

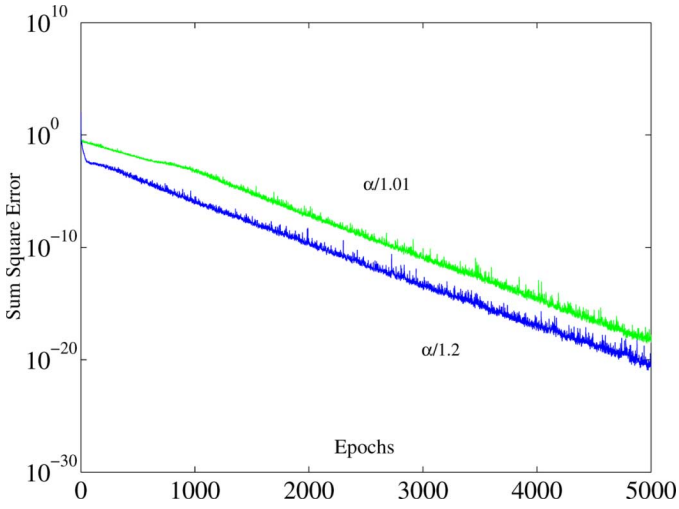


Fig. 21. Average performances for regularized training when α is divided by 1.01 and 1.2 at each epoch.

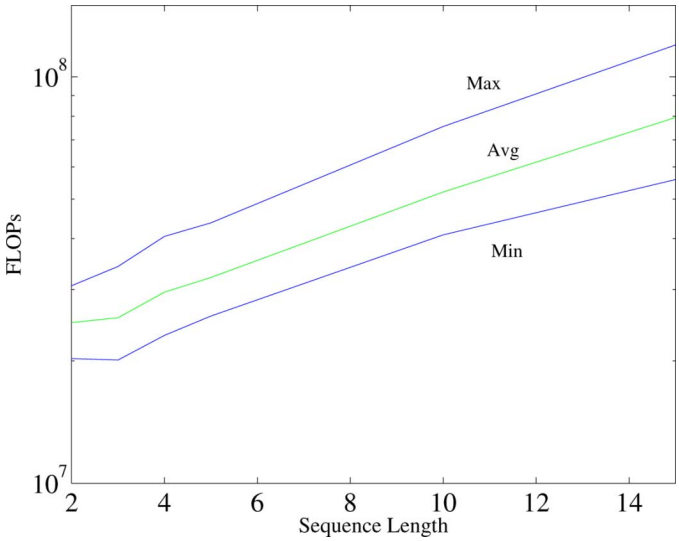


Fig. 22. FLOPs required to obtain convergence as a function of sequence length.

take a longer time sequence to enable accurate identification of the more complex dynamics of the corresponding process.

VIII. Conclusion

This paper has presented an analysis of some problems that are encountered when training recurrent neural networks. We found that the error surface for recurrent neural networks contains spurious valleys that make the training more difficult for batch gradient-descent algorithms. The formation of these valleys can be understood through an analysis of random polynomials. This type of analysis has not been previously used to examine recurrent network performance. We have identified four different types of spurious valleys, and have developed algorithms to predict the valley locations for simple networks.

Even though a detailed analysis of the valley formation for large networks has not been performed, we know that the locations of the valleys are dependent on the data used to train the networks and on the initial conditions of the layer outputs. If the training data or the initial conditions are changed, then the locations of the valleys are moved. Using this knowledge, we proposed several techniques for improving the convergence of recurrent network batch training algorithms. We found that regularization, frequent switching of training sequences, and application of random initial conditions to the layer outputs are useful training modifications for recurrent networks that mitigate the effects of the spurious valleys.

The algorithm modifications that we have proposed are not entirely new. The idea of switching sequences is related to stochastic algorithms, such as stochastic gradient and extended Kalman filter methods, in which the weights are updated as each input is presented to the network, and no line search is performed. (It has been known for some time that stochastic algorithms perform better than standard batch algorithms for recurrent networks. The analysis provided in this paper provides an explanation for this behavior.) Regularization has been used for many years, and can be developed from a Bayesian framework in which a Gaussian prior is assumed for the network weights [28], although the decay of the regularization parameter is a new modification for recurrent network training. To our knowledge, the random setting of initial conditions (for the delay states in the network—not for the network weights) has not been previously suggested in the literature. The novelty of our approach is that all of these techniques are used in combination to avoid spurious local minima that are caused by the specific training input sequence and initial delay states.

The analysis in Sections IV and V suggests yet another approach for avoiding the spurious valleys. Because the valleys are related to instabilities in the neural network, one might be able to use a constrained optimization process to avoid these instabilities during training. However, the implementation of the constraints would be extremely complex for the general training of recurrent networks. For the simple network of Fig. 2, stability could be maintained by forcing the magnitude of the weight w_2 to be less than 1 in magnitude. However, the constraint would be much more complex for a network with arbitrary recurrent connections. We cannot really talk about the stability of an arbitrary nonlinear recurrent network, but rather the stability of a particular trajectory—typically an equilibrium point (see [2] and [37]). For a given network, some solutions might be stable and others unstable. In addition, we may want to use a recurrent network to model a chaotic system. Such a system would have locally unstable solutions, although the responses would be globally bounded. For these cases, we would not want to constrain the weights to maintain stability. The constrained optimization approach might be possible if these issues could be addressed.

References

[1] A. F. Atiya and A. G. Parlos, "New results on recurrent network training: Unifying the algorithms and accelerating convergence," *IEEE Trans. Neural Netw.*, vol. 11, no. 3, pp. 697–709, May 2000.

[2] N. E. Barabanov and D. V. Prokhorov, "Stability analysis of discrete-time recurrent neural networks," *IEEE Trans. Neural Netw.*, vol. 13, no. 2, pp. 292–303, Mar. 2002.

[3] Y. Bengio, P. Simard, and P. Frasconi, "Learning long-term dependencies with gradient descent is difficult," *IEEE Trans. Neural Netw.*, vol. 5, no. 2, pp. 157–166, Mar. 1994.

[4] A. T. Bharucha-Reid and M. Sambandham, *Random Polynomials*. Orlando, FL: Academic, 1986.

[5] M. Bianchini, M. Gori, and M. Maggini, "On the problem of local minima in recurrent neural networks," *IEEE Trans. Neural Netw.*, vol. 5, no. 2, pp. 167–177, Mar. 1994.
[6] G. Chengyu and K. Danai, "Fault diagnosis of the IFAC benchmark problem with a model-based recurrent neural network," in *Proc. IEEE Int. Conf. Control Appl.*, 1999, vol. 2, pp. 1755–1760.
[7] O. De Jesús, "Training general dynamic neural networks," Ph.D. dissertation, Schl. Electr. Comput. Eng., Oklahoma State Univ., Stillwater, OK, 2002.
[8] O. De Jesús and M. T. Hagan, "Backpropagation algorithms for a broad class of dynamic networks," *IEEE Trans. Neural Netw.*, vol. 18, no. 1, pp. 14–27, Jan. 2007.
[9] O. De Jesús, J. M. Horn, and M. T. Hagan, "Analysis of recurrent network training and suggestions for improvements," in *Proc. INNS-IEEE Int. Joint Conf. Neural Netw.*, Washington, DC, Jul. 2001, vol. 4, pp. 2632–2637.
[10] O. De Jesús, A. Pukrittayakamee, and M. T. Hagan, "A comparison of neural network control algorithms," in *Proc. INNS-IEEE Int. Joint Conf. Neural Netw.*, Washington, DC, Jul. 2001, vol. 1, pp. 521–526.
[11] R. T. Farouki and V. T. Rajan, "On the numerical condition of polynomials in Bernstein form," *Comput.-Aided Geometric Des.*, vol. 4, pp. 191–216, 1987.
[12] J. Feng, C. K. Tse, and F. C. M. Lau, "A neural-network-based channel-equalization strategy for chaos-based communication systems," *IEEE Trans. Circuits Syst. I: Fundam. Theory Appl.*, vol. 50, no. 7, pp. 954–957, Jul. 2003.
[13] S. Fernández, A. Graves, and J. Schmidhuber, "Sequence labelling in structured domains with hierarchical recurrent neural networks," in *Proc. 20th Int. Joint Conf. Artif. Intell.*, Hyderabad, India, 2007, pp. 774–779.
[14] P. Gianluca, D. Przybylski, B. Rost, and P. Baldi, "Improving the prediction of protein secondary structure in three and eight classes using recurrent neural networks and profiles," *Proteins: Structure, Function, Genetics*, vol. 47, no. 2, pp. 228–235, 2002.
[15] M. Gori and A. Sperduti, "The loading problem for recursive neural networks," *Neural Netw.*, vol. 18, pp. 1064–1079, 2005.
[16] A. Graves, S. Fernández, F. Gomez, and J. Schmidhuber, "Connectionist temporal classification: Labelling unsegmented sequence data with recurrent neural nets," in *Proc. 23rd Int. Conf. Mach. Learn.*, Pittsburgh, PA, 2006, pp. 369–376.
[17] A. Graves, S. Fernández, M. Liwicki, H. Bunke, and J. Schmidhuber, "Unconstrained on-line handwriting recognition with recurrent neural networks," *Advances in Neural Information Processing Systems*, vol. 20, pp. 577–584, 2008.
[18] M. T. Hagan and H. B. Demuth, "Neural networks for control," in *Proc. Amer. Control Conf.*, San Diego, CA, 1999, pp. 1642–1656.
[19] M. T. Hagan, O. De Jesús, and R. Schultz, "Training recurrent networks for filtering and control," in *Recurrent Neural Networks: Design and Applications*, L. Medsker and L. C. Jain, Eds. Boca Raton, FL: CRC Press, 1999, ch. 12, pp. 311–340.
[20] S. Hochreiter, "Untersuchungen Zudynamischen Neuronalen Netzen," Diploma, Technische Universitat Munchen, Institut fur Informatik, Munich, Germany, 1991.
[21] S. Hochreiter and J. Schmidhuber, "Long short-term memory," *Neural Comput.*, vol. 9, no. 8, pp. 1735–1780, 1997.
[22] J. Horn and M. Hagan, "Analysis of the error surface of simple recurrent neural networks," in *Intelligent Engineering Systems Through Artificial Neural Networks (ANNIE2004)*. New York: ASME Press, 2004, vol. 14.
[23] T.-Y. Huang, C. J. Li, and T.-W. Hsu, "Structure and parameter learning algorithm of Jordan type recurrent neural networks," in *Proc. Int. Joint Conf. Neural Netw.*, Orlando, FL, Aug. 12–17, 2007, pp. 1819–1824.
[24] Jayadeva and S. A. Rahman, "A neural network with O(N) neurons for ranking N numbers in O(1/N) time," *IEEE Trans. Circuits Syst. I: Reg. Papers*, vol. 51, no. 10, pp. 2044–2051, Oct. 2004.
[25] M. Kac, "On the average number of real roots of a random algebraic equation," *Bull. Amer. Math. Soc.*, vol. 49, pp. 314–320, 1943.
[26] M. Kac, *Probability and Related Topics in the Physical Sciences*, 1st ed. New York: Interscience, 1960.
[27] I. Kamwa, R. Grondin, V. K. Sood, C. Gagnon, V. T. Nguyen, and J. Mereb, "Recurrent neural networks for phasor detection and adaptive identification in power system control and protection," *IEEE Trans. Instrum. Meas.*, vol. 45, no. 2, pp. 657–664, Apr. 1996.
[28] D. J. C. MacKay, "Bayesian interpolation," *Neural Comput.*, vol. 4, pp. 415–447, 1992.
[29] L. R. Medsker and L. C. Jain, *Recurrent Neural Networks: Design and Applications*. Boca Raton, FL: CRC Press, 2000.
[30] T. Poggio and F. Girosi, "Networks for approximation and learning," *Proc. IEEE*, vol. 78, no. 9, pp. 1481–1497, Sep. 1990.
[31] A. J. Robinson, "An application of recurrent nets to phone probability estimation," *IEEE Trans. Neural Netw.*, vol. 5, no. 2, pp. 298–305, Mar. 1994.
[32] J. Roman and A. Jameel, "Backpropagation and recurrent neural networks in financial analysis of multiple stock market returns," in *Proc. 29th Hawaii Int. Conf. Syst. Sci.*, 1996, vol. 2, pp. 454–460.
[33] E. D. Sontag and H. J. Sussmann, "Backpropagation can give rise to spurious local minima even for networks without hidden layers," *Complex Syst.*, vol. 3, pp. 91–106, 1989.
[34] R. Williams and D. Zipser, "A learning algorithm for continually running fully recurrent neural networks," *Neural Comput.*, vol. 1, no. 2, pp. 270–280, 1989.
[35] W. Yang, "Neurocontrol using dynamic learning," Ph.D. dissertation, Schl. Electr. Comput. Eng., Oklahoma State Univ., Stillwater, OK, 1994.
[36] W. Yang and M. T. Hagan, "Training recurrent networks," in *Proc. 7th Oklahoma Symp. Artif. Intell.*, Stillwater, OK, 1993, pp. 226–233.
[37] L. Wang and Z. Xu, "Sufficient and necessary conditions for global exponential stability of discrete-time recurrent neural networks," *IEEE Trans. Circuits Syst. I, Reg. Papers*, vol. 53, no. 6, pp. 1373–1380, Jun. 2006.
[38] J.-S. Wang and Y.-P. Chen, "A fully automated recurrent neural network for unknown dynamic system identification and control," *IEEE Trans. Circuits Syst. I, Reg. Papers*, vol. 53, no. 6, pp. 1373–1380, Jun. 2006.
[39] P. Xiao, G. K. Venayagamoorthy, and K. A. Corzine, "Combined training of recurrent neural networks with particle swarm optimization and backpropagation algorithms for impedance identification," in *Proc. IEEE Swarm Intell. Symp.*, 2007, pp. 9–15.

Jason Horn (M'04) received dual B.S. degrees in electrical engineering and computer science from Oklahoma State University, Stillwater, in 2005 and the M.S. degree in electrical engineering from the University of Texas at Austin, in 2007.

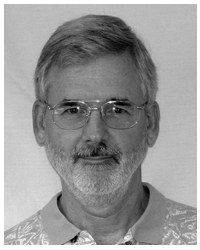
He began working at Agilent Technologies High Frequency Technology Center, Santa Clara, CA, as an intern in 2004, and accepted a full time position as an R&D Engineer in 2007. His primary areas of interest are high-frequency nonlinear measurement and behavioral modeling.

Mr. Horn is a member of the Microwave Theory and Techniques Society (MTT-S).

Orlando De Jesús (M'00) received the degrees of Engineer in electronics and Project Management Specialist from Universidad Simón Bolívar, Caracas, Venezuela, in 1985 and 1992, respectively, and the M.S. and Ph.D. degrees in electrical engineering from Oklahoma State University, Stillwater, in 1998 and 2002, respectively.

He held engineering and management positions at AETI C.A., Caracas, Venezuela, from 1985 to 1996, developing data acquisition and control systems for the oil industry in Venezuela. He later developed the control system blocks and the dynamic neural networks training algorithms for the Matlab Neural Network toolbox. He is currently a Technical Advisor in the Research Department, Halliburton Energy Services, Dallas, TX His research interests include computational fluid dynamics, control systems, signal processing, software modeling, neural networks, robotics, automation, and instrumentation applications. He has coauthored 13 technical publications and holds ten U.S. patents.

Dr. De Jesús is a member of the Society of Petroleum Engineers (SPE).



Martin T. Hagan (M'78) received the B.S. degree in electrical engineering from the University of Notre Dame, Notre Dame, IN, in 1972, the M.S. degree in information and computer science from Georgia Institute of Technology, Atlanta, in 1973, and the Ph.D. degree in electrical engineering from the University of Kansas, Lawrence, in 1977.

Currently, he is Professor of Electrical and Computer Engineering at Oklahoma State University, Stillwater, where he has taught and conducted research in the areas of statistical modeling and control systems since 1986. He was previously with the faculty of Electrical Engineering at the University of Tulsa from 1978 to 1986. He was also a Visiting Scholar with Department of Electrical and Electronic Engineering, University of Canterbury, Christchurch, New Zealand, during the 1994 academic year and with the Laboratoire d'Analyse et d'Architecture des Systèms, Centre National de la Recherche Scientifique, Toulouse, France, during the 2005–2006 academic year. He is the author, with H. Demuth and M. Beale, of the textbook *Neural Network Design* (Boston, MA: PWS, 1994). He is also a coauthor of the Neural Network Toolbox for MATLAB.