Analysis of Recurrent Network Training and Suggestions for Improvements

Orlando De Jesús dorland@okstate.edu
Jason M. Horn hornjm@okstate.edu
Martin T. Hagan mhagan@okstate.edu
Oklahoma State University, School of Electrical and Computer Engineering,
202 Engineering South, Stillwater, OK, 74078-5032, USA

Abstract

This paper describes some of the difficulties in training recurrent neural networks, provides explanations for why these difficulties occur and explains how they can be mitigated.

1 Introduction

Recurrent neural networks have been applied successfully in the identification and control of dynamic systems [5]. Generally training of this class of neural networks has been difficult due to the time dependencies present in their architecture. Some researchers have suggested improvements that can be made to well know training algorithms, which can improve the training of recurrent networks. Bengio et al. [2] reported difficulties using gradient descent based algorithms. They proposed alternatives, such as simulated annealing, multi-grid random search, time-weighted Pseudo-Newton optimization and discrete error propagation. Bianchini et al. [3] discussed the problem of local minima in recurrent neural networks. The input structure analysis they provided was related to the size of the "frame input" and the sequence of frames. They provided "recurrent network assumptions (RNA)" that allow the design of recurrent network architectures. However, the conditions for optimal learning are only sufficient. The design criteria may create networks with large input size based on the unfolding in time of the neural network weights. Atiya and Parlos [1] proposed a new algorithm based on approximating the gradient calculation. They comment that although we have a no exact search direction, we can obtain faster convergence.

In this paper we will suggest a mechanism that can explain, at least in part, the difficulties that occur in training recurrent networks. 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 training procedures on two simple recurrent networks.

2 Prelude

We begin with an explanation of how we came across a certain characteristic of the error surfaces of recurrent networks. While training a neural-network-based Model Reference Controller [4], we found that the error sometimes increased during training, although a line search 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. Typical profiles are shown in Figure 1. For the system shown, we have 65 weights being trained. The surface we present is along the direction of search (obtained by the BFGS quasi-Newton algorithm) through a 65-dimensional space. It is clear from these profiles 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. 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 in the input sequence.

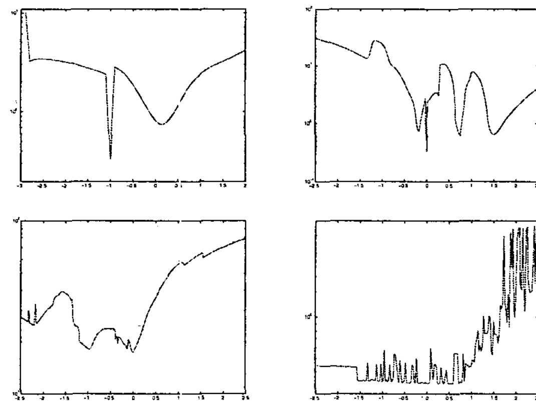

Figure 1: Error profile

In order to understand how the 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 will discuss a first-order linear recurrent network that produces the spurious valleys. We will also show how nonlinear transfer functions can affect the shape of the valleys.

0-7803-7044-9/01/$10.00 ©2001 IEEE

This is followed by some modifications we propose to improve the training process, based on our analysis of the creation of the spurious valleys. The fourth section 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.

3 First Order Model

3.1 Linear model

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

Figure 2: First order linear recurrent network.

In order to generate an error surface, we first developed training data using the network of Figure 2, where the weights were set to $\mathbf{W}^1 = 0.5$ and $\mathbf{W}^2 = 0.5$. We used a random input sequence for $u(t)$, and then used the network to generate a sequence of outputs $y(t)$. Our training objective was then to train another network with the same architecture to fit the training data. The global minimum of the error surface should occur at the values $\mathbf{W}^1 = 0.5$ and $\mathbf{W}^2 = 0.5$.

Figure 3 is a typical error surface obtained using the above procedure for one particular input sequence and the initial output $y(0) = 0$. (The plot is cut off at 10,000 to better visualize the surface.) There are several interesting features of the surface. First, the error surface generally increases dramatically as the weight $\mathbf{W}^2$ 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 $|\mathbf{W}^2| > 1$, for this particular input sequence there are some values for $\mathbf{W}^2$ in this range that produce small network outputs (and therefore relatively small errors). We expect the output to grow without bound under these conditions, but it doesn't always happen.

Figure 3: Error surface for first order linear model.

The two valleys in the error surface occur for two different reasons. One valley occurs at $\mathbf{W}^1 = 0$. If this weight is zero, and the initial condition is zero, the output of the network will remain zero. Therefore, our mean squared error will be constant and equal to the mean square value of the target outputs.

The second valley in the error surface is due to the input sequence that is presented to the network. For a given input $u(t)$, the system output will be:

$$y(t) = \mathbf{W}^1 u(t) + \mathbf{W}^2 y(t-1) \tag{1}$$

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

$$\begin{aligned} y(k) &= \mathbf{W}^1 u(k) + \mathbf{W}^2 y(k-1) \\ &= \mathbf{W}^1 u(k) + \mathbf{W}^2(\mathbf{W}^1 u(k-1) + \mathbf{W}^2 y(k-2)) \\ &\quad \dots \quad \dots \quad \dots \quad \dots \\ &= \mathbf{W}^1\{u(k) + \mathbf{W}^2 u(k-1) + (\mathbf{W}^2)^2 u(k-2) \\ &\quad + \dots + (\mathbf{W}^2)^{k-1} u(1)\} + (\mathbf{W}^2)^k y(0) \end{aligned} \tag{2}$$

Here we can see that the response at time k is a polynomial in the parameter $\mathbf{W}^2$. 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 $\mathbf{W}^2$ that is larger than 1 in magnitude for which the output $y(k)$ 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 Eq. (2) has a root outside the unit circle, that root appears also in the next polynomial at time k+1, and therefore the output will remain small for future times.

Figure 4 shows a cross-section of the error surface presented in Figure 3 for $\mathbf{W}^1 = 0.5$ using different sequence lengths. The error falls abruptly near -3.8239. That is the root of the polynomial described in Eq. (2). The root maintains its location as the sequence increases in length (k increases).

Figure 4: Error cross-section for $\mathbf{W}^1 = 0.5$.

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 a later example.) 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 may produce a valley in a different location.

In section 4 we will propose some modified training procedures that can mitigate the effects of the spurious valleys. Before introducing that topic, let's investigate the effect of nonlinear transfer functions on the error surface.

3.2 Nonlinear network

Figure 5 presents a modification of the linear network presented in the previous section. Here we include a sigmoid nonlinearity at the output.

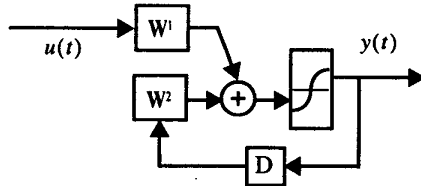

Figure 5: First order nonlinear model.

Figure 6 presents the error surface for the same input sequence used in the previous section. Due to the nonlinearity, the output is bounded for large weight values. So 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.

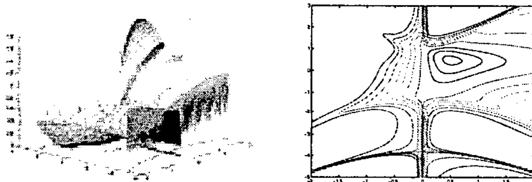

Figure 6: Error surface for first order nonlinear network.

4 Modifications to the Training Procedure

From the previous section we see that difficulties in training recurrent neural networks could be due to the presence of spurious valleys. The shape of the valleys could be 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.

4.1 Proposed solutions

In this section we will propose three variations to the standard 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 section 3, we notice that the linear case has a natural way of allowing convergence to the optimal weights, because larger weights generate a 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 [7] with our mean square error performance function. In other words we can use the performance function

$$J(\mathbf{W}) = SSE + \alpha SSW, \quad (3)$$

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 don't bias the final trained weights.

Another technique for improved training involves using more that one training sequence. Figure 7 presents the error surface for the nonlinear model of Figure 5, using a different sequence. The valley that appeared in Figure 6 has moved to the positive region of $\mathbf{W}^2$. For any two random input sequences, the valleys will appear in different locations.

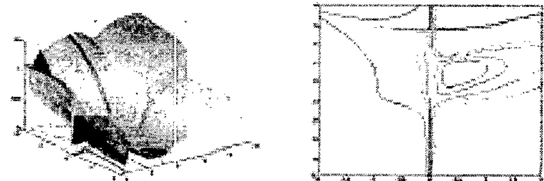

Figure 7: Error surface for first order nonlinear model for different input sequence.

This suggests another technique for improved training. We could use multiple input 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. Figure 8 presents an error surface for five sequences. This figure demonstrates how the spurious valleys are reduced in amplitude.

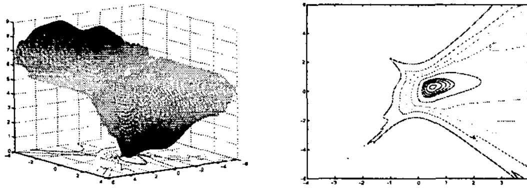

Figure 8: Error surface using sequence averaging.

Another method to move the valleys is to use random initial conditions. Figure 9 shows how the error surface is changed when we set the initial condition to $y(0) = 0.1$. The valley at $\mathbf{W}^1 = 0$, 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.

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).

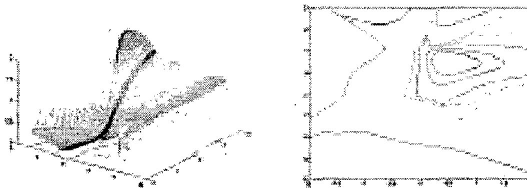

Figure 9: Error surface using $y(0) = 0.1$.

5 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 3 (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. We will define the results obtained with the gradient descent algorithm alone as 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 dynamic backpropagation method described in [6], [8] and [9].

5.1 First order nonlinear system

For the first order nonlinear system we generated training data using $\mathbf{W}^1 = 0.5$ and $\mathbf{W}^2 = 0.5$. The training was done using 25000 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 1 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. Each method provides some improvement on the baseline method. However, the multiple method is the only one that guarantees accurate convergence.

Table 1: Percentage of final weights within 0.001 of the optimal weights for the first order nonlinear network.

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

Figure 10 shows the relative final position of $\mathbf{W}^1$ vs. $\mathbf{W}^2$ for Baseline, SS, AS and IC. For the three first methods many tests finished along $\mathbf{W}^1 = 0$. That condition was removed when we set the initial conditions to 0.2. When we switch the sequences we avoided many cases where training may be trapped in the spurious valleys. The averaging of sequences did not improve our training results, resulting in worse results than the baseline method for 5 std.

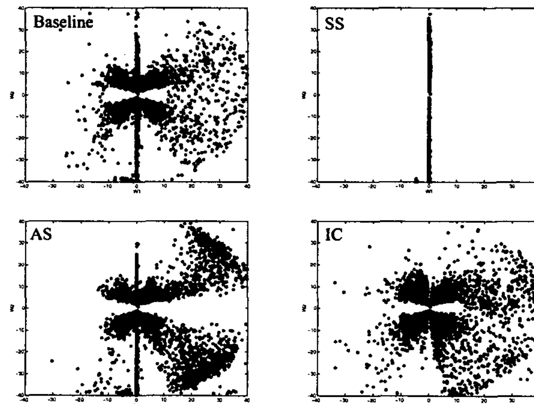

Figure 10: Relative final position of $\mathbf{W}^1$ vs. $\mathbf{W}^2$ for 5 std.

5.2 Two layer neural network

Figure 11 has a neural network with two layers, where each layer is feedback to the previous layers. This system will allow us to test the previous training procedure modifica-

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

$$W^2 = -0.5 \quad W^3 = 0.25$$
$$W^1 = 0.5 \quad W^4 = -0.3$$

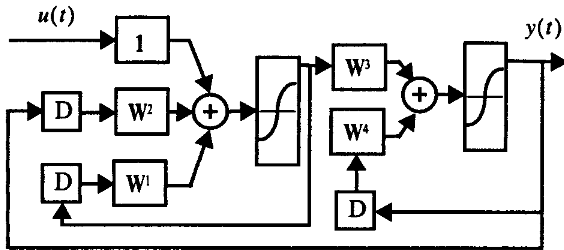

Figure 11: Two-layer nonlinear model.

Table 2 shows the percentage of weights close to the final weights after the training process. For this neural network architecture, regularization resulted in a success rate of over ninety percent. However, it is again the multiple method that guarantees the best convergence.

Table 2: Percentage of final weights within 0.5 of the optimal weights for the two-layer nonlinear network.

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

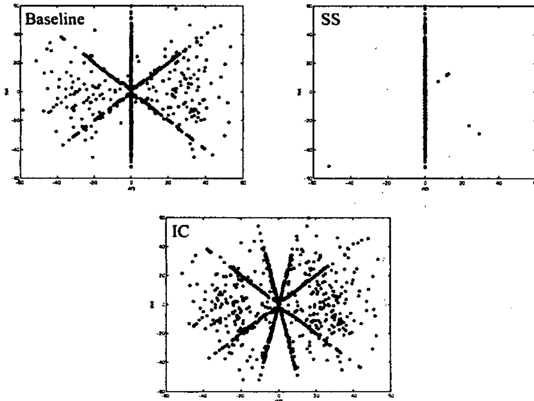

Figure 12: Relative final position of W^3 vs. W^4 for 20 std.

Figure 12 presents the final weight positions in the W^3 vs. W^4 plane for the Baseline, SS and IC training methods. For the Baseline training method, we notice the presence of three axes or valleys where the training converged. From the middle figure we can see that the SS method can eliminate the diagonal final condition. However, the axis along $W^3 = 0$ remains. When we set the initial layer conditions to 0.2, we can see from the last figure that two new axes appear. This demonstrates that setting the initial conditions to nonzero values does not necessarily remove spurious valleys. It may just move them to new locations. This suggests that we should vary the initial conditions whenever we switch the training sequence.

Figure 13 shows how the final distance to the optimal weights is affected by the switching sequence interval. While training for 10000 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 with the same sequence, we could be caught in a spurious valley, resulting in failed training.

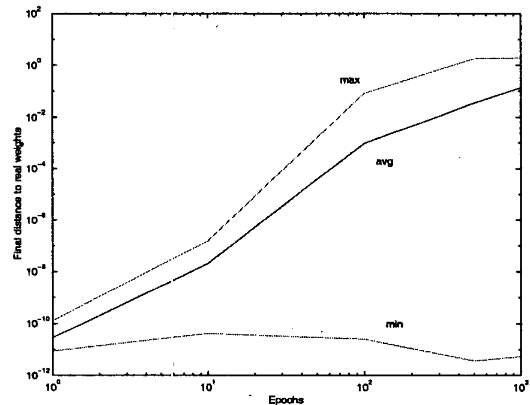

Figure 13: Final distance to optimal weights for different switching sequence intervals.

Figure 14 shows the average performance for three different switching intervals. We obtain substantial improvement when the sequence is switched more frequently. 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. The constants we used were 1.01, 1.2 and 2. Figure 15 shows the average performance when α is divided by 1.01 and 1.2. 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 concluded that α must be decreased in some way to obtain the best training results.

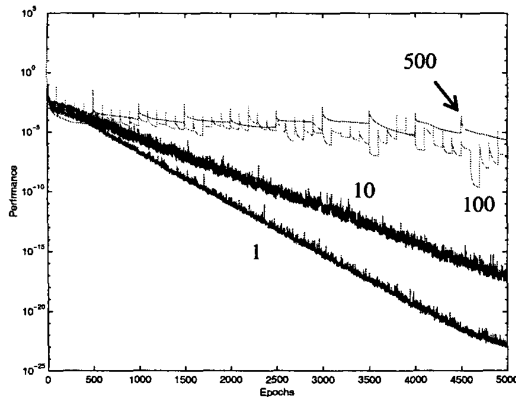

Figure 14: Average performance for different switching sequence interval.

Figure 16 shows the number of 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.

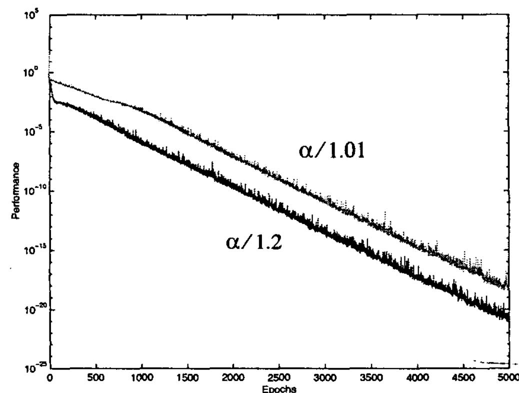

Figure 15: Average performance for $\alpha/1.01$ and $\alpha/1.2$.

6 Summary

This paper has presented an analysis of some problems that we may encounter when training recurrent neural networks. We found that the error surface for recurrent neural networks contain spurious valleys that make the training more difficult for gradient descent 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.

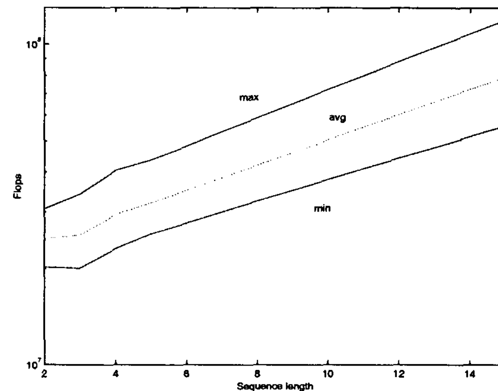

Figure 16: Flops for different sequence length.

7 References

[1] Atiya, A.F.; Parlos, A.G., "New Results on Recurrent Network Training: Unifying the Algorithms and Accelerating Convergence," *IEEE Transactions on Neural Networks* Vol. 11, 2000, pp. 697 - 709.

[2] Bengio, Y.; Simard, P.; Frasconi, P., "Learning Long-Term Dependencies with Gradient Descent is Difficult," *IEEE Transactions on Neural Networks* Vol. 5, 1994, pp. 157 - 166.

[3] Bianchini, M.; Gori, M.; Maggini, M., "On the Problem of Local Minima in Recurrent Neural Networks," *IEEE Transactions on Neural Networks* Vol. 5, 1994, pp. 167 - 177.

[4] De Jesús, O.; Pukrittayakamee, A.; Hagan, M.T., "A Comparison of Neural Network Control Algorithms," To be publish in: *Proceedings of the INNS-IEEE International Joint Conference on Neural Networks*, Washington DC, July 2001.

[5] Hagan,M.T.; Demuth, H.B., "Neural Networks for Control," *Proceedings of the 1999 American Control Conference*, San Diego, CA, 1999, pp. 1642-1656.

[6] Hagan, M.T.; De Jesus, O.; Schultz, R., "Training Recurrent Networks for Filtering and Control," Chapter 12 in *Recurrent Neural Networks: Design and Applications*, L. Medsker and L.C. Jain, Eds., CRC Press, 1999, pp. 311-340.

[7] Poggio, T.; Girosi, F., "Networks for Approximation and Learning," *Proceedings of the IEEE*, Vol. 78, 1990, pp. 1481-1497.

[8] Yang, W., "Neurocontrol Using Dynamic Learning," *Doctoral Thesis*, Oklahoma State University, Stillwater, 1994.

[9] Yang, W.; Hagan, M.T., "Training Recurrent Networks", *Proceeding of the 7th Oklahoma Symposium on Artificial Intelligence*, Stillwater, 1993, pp. 226-233.