A Procedure for Training Recurrent Networks

Manh C. Phan, Mark H. Beale and Martin T. Hagan

Abstract—In this paper, we introduce a new procedure for efficient training of recurrent neural networks. The new procedure uses a batch training method based on a modified version of the Levenberg-Marquardt algorithm. The information of gradients of individual sequences is used to mitigate the effect of spurious valleys in the error surface of recurrent networks. The method is tested on the modeling and control of several physical systems.

I. INTRODUCTION

Artificial Neural Networks can be categorized as feedforward neural networks (FNNs) or recurrent neural networks (RNNs). RNNs have at least one feedback loop, while in FNNs, the network output depends not only on the current inputs but also on past inputs, outputs and/or states of the networks.

FNNs can be trained efficiently using the Levenberg-Marquardt (LM) algorithm [1]—a Jacobian-based batch algorithm. For FNNs with reasonable sizes (a few hundred weights), the LM algorithm has been shown to be fast, accurate and consistent. It has been used for training neural networks in a variety of applications.

RNNs have rich dynamics and powerful computational capabilities. They are especially suitable for applications involving sequential or time-varying processing. Those applications include dynamic modeling, prediction, time series analysis, control systems, etc. (see the introduction in [2] for details). Despite their capabilities, however, it is well-known that RNNs are very difficult to train [3], [4]. A recent effort to improve RNN training [5] uses a Hessian-free optimization approach to deal with the problem of long-term dependencies. Other efforts for efficient RNN training can be found in the review paper [6].

One reason for the difficulties in RNN training is the existence of spurious valleys in the error surface [2], [7]. These valleys are not related to the true minimum of the surface, or to the problem the RNN is trying to solve. They depend on the input sequence in the training data. Any batch search algorithm, including LM, is very likely to be trapped in these spurious valleys.

In this paper, we propose a new training procedure for RNNs that could overcome the problem of spurious valleys. In [2] and [7] we proposed some modified training procedures that help mitigate the effects of the valleys, however, we demonstrated the operation of those procedures only on simple networks. The new procedure proposed in this paper will be applied for training practical RNNs.

Since the locations of the valleys depend on the input sequences, whenever the search is stuck, one possible solution is to remove the sequence (or sequences) that causes the valley where the search is stuck. In addition, the longer sequences produce steeper and more numerous valleys [2], [7]. Therefore, it would help to start the training with one-step-ahead prediction and then slowly predict further ahead. Details of this procedure will be presented in Section IV. To the best of our knowledge, the techniques in this procedure (removing sequences based on gradient information and incrementally increasing the prediction horizon) have not been used for practical training of RNNs. Section IV presents the main contribution of this paper. Other sections of this paper are as follows: We review some properties of the spurious valleys of RNNs in Section II and the LM algorithm in Section III. Section V gives some simulation results and Section VI concludes the paper.

II. BACKGROUND ON VALLEYS OF RECURRENT NETWORKS

In this section, we will review the properties of spurious valleys in the error surface of RNNs [2], [7]. It has been shown that the error surface of RNNs has spurious valleys that significantly disrupt the training of these networks. These valleys do not relate to the true minimum or to the problem the RNN is trying to solve. They depend on the input sequences, the initial conditions and the network architectures. These spurious valleys are features of networks that have feedback connections.

One property of these valleys is that they are caused by network instability and are related to the input sequence. In the unstable region of network, the output grows without bound (for linear networks) or saturates (for nonlinear networks). However, there are some locations in the unstable region (certain combinations of weights) where the output is still small for a particular input sequence. For example, in the first-order linear network [2], at locations where the feedback weight $w_2$ equals a root of the input sequence with magnitude bigger than 1, the output equals zero. Since the outputs nearby are much bigger, a valley appears. We would like to emphasize that the valleys occur because of the instability (the valleys are in the unstable region) and the input sequence. If the input sequence is modified, it will produce a valley in a different location.

Another feature of the spurious valleys is that they are very narrow and have steep slopes. (Some of the valleys were found to have widths on the order of $10^{-15}$.) This is understandable,
because of the sudden change in the output at the valleys. This also means that the norm of gradient of the performance index with respect to the weights would be very large. Furthermore, longer sequences produce steeper valleys (see Fig. 4 in [2]). Therefore, the norm of the gradient would indicate when the search reaches a valley.

As the RNNs become larger, with more layers, neurons, or feedback connections, the valleys become more numerous. An error profile for a practical network is shown in Fig. 1. The plot shows a cross section of the error surface (the MSE along the gradient direction where \( \alpha \) represents the fractional change in the weights). We can see that there are many valleys in such a small range. We definitely want to escape from this region during the training process.

III. LEVENBERG-MARQUARDT ALGORITHM: A SHORT DESCRIPTION

This section gives a brief description of the LM algorithm, which will be modified in a later section so as to avoid the spurious valleys. Details about LM can be found in [1] and [8]. The LM algorithm is a variation of Newton’s method where the performance index is sum squared error (SSE). The update rule for weights and biases \( x_k \) at the \( k^{th} \) iteration is

\[
\Delta x_k = - \left[ J^T(x_k)J(x_k) + \mu_k I \right]^{-1} J^T(x_k)e(x_k)
\]

where \( e \) is the network error and \( J \) is the Jacobian matrix of the network errors with respect to the weights. \( J \) can be computed using backpropagation. For RNNs, we need to use dynamic backpropagation. Jacobian calculations for a general dynamic network can be found in [9].

This algorithm has the very useful feature that as \( \mu_k \) is increased it approaches the steepest decent algorithm with a learning rate of \( 1/\mu_k \), while as \( \mu_k \) is decreased to zero the algorithm becomes Gauss-Newton. The algorithm begins with a small \( \mu_k \) (e.g., \( \mu_k = 0.01 \)). If a step does not yield a smaller value for the SSE, then the step is repeated with \( \mu_k \) multiplied by some factor \( \vartheta > 1 \) (e.g., \( \vartheta = 10 \)). Eventually the SSE should decrease, since we would be taking a small step in the steepest descent direction. If a step does produce a smaller value for SSE, then \( \mu_k \) is divided by \( \vartheta \) for the next step, so the algorithm will approach Gauss-Newton, which should provide faster convergence. The algorithm provides a nice compromise between the speed of Newton’s method and the guaranteed convergence of steepest descent. We can see that in the LM algorithm, the SSE is always decreasing from one iteration to the next.

One stopping criterion for the LM algorithm is \( \mu_k \) reaching a maximum value (e.g., \( \mu_{max} = 10^{10} \)) without the SSE decreasing. This rarely happens for FNN training. (FNN training usually stops when the norm of the gradient has reduced below some minimum value.) However, this is not the case for RNN training. In RNN training, \( \mu_k \) is quite likely to reach\( \mu_{max} \), especially for closed-loop training with long input sequences (see Section IV). As \( \mu_k \) reaches \( \mu_{max} \), we would take a very small step in the steepest descent direction. If the SSE cannot be reduced without making \( \mu_k \) very large, then there exists a very narrow valley at the current weights. This also means that the norm of the gradient at that weight location would be very large. Increasing \( \mu_{max} \) does not help in this case, since eventually we will get trapped in that narrow valley. Therefore, the idea is to escape from the valley. The procedure for implementing this idea will be presented in the next section.

IV. NEW TRAINING PROCEDURE

A. Rationale

We explained in Section II that the valleys are caused by instability and are related to the input sequences. Also, longer sequences produce steeper valleys. We also explained in Section III that \( \mu \) reaching \( \mu_{max} \) is an indication that the search algorithm has fallen inside a valley. Using this knowledge, we can modify the LM algorithm to improve convergence. The overall idea is to train with multiple sequences, and then if \( \mu \) reaches \( \mu_{max} \) to remove the sequence (or sequences) that are associated with the valley. The removed sequence should have larger gradient than the other sequences. Before introducing the training procedure, we need to explain some concepts that are commonly used in recurrent network training.

B. Useful concepts

1) Open-loop training and one-step-ahead prediction: We can consider the output of an RNN as an estimate of the output of the nonlinear dynamic system that we are trying to model. The output is fed back to the input of some layers in the network. Because the true output is available during the training of the network, we can use the true output instead of feeding back the estimated output. This is similar to the series-parallel configuration introduced in [10]. The advantage of this configuration is that the input to some layers of the network will be more accurate. Therefore it is easier for the algorithm to find the true minimum. Also, by removing feedback loops we reduce the chance for spurious valleys.
Because of the series-parallel configuration, we are doing one-step-ahead predictions. It turns out that doing one-step-ahead predictions is a very helpful initial step in RNN training. For this initial step, all feedback loops from the network output can be opened. After completing the one-step-ahead training, the feedback loops are closed for multi-step-ahead training.

2) Closed-loop training and multiple-step-ahead prediction: The original RNN is in the parallel configuration [10] or closed-loop form. We want to train this closed-loop network to perform an iterated prediction over many time steps (multiple-step-ahead prediction). (Note that for closed-loop training, we need to use dynamic backpropagation.) The time horizon of the prediction is determined by the length of the training sequences and the number of delays in the networks. For example, if the training sequences have a length of 5 time steps, and the maximum number of delays in the network is 2, then training the closed-loop network means that we are doing 3-step-ahead predictions.

Because of the relationship between the sequence length and the width of the spurious valleys mentioned above, we will start closed-loop training with short training sequences first, and then we will increase the prediction horizon. At each stage of this process, it requires that we segment the original long training sequences into shorter sequences.

C. Procedure

The following procedure could be used to train any arbitrary RNN:

1) Open-loop training (one-step-ahead predictions)
2) Closed-loop training with increasing prediction horizon:
   Do k-step-ahead prediction (k ≥ 2). This includes segmentation of original long sequences into small sequences.
3) At each iteration of the LM algorithm, if µ reaches µmax, remove the sequence with largest gradient. If the SSE does not decrease, keep removing the sequence with next largest gradient until the SSE decreases (the algorithm escapes from the valleys). Add the removed sequences back to the training data before proceeding to next iteration.
4) Increase the prediction horizon k (sequence length).
   If all sequences are removed, shorten the prediction horizon and go back to step 2.

In the original LM algorithm, the training stops if µ reaches µmax, and in these cases the resulting trained network performance is often very poor. In this modified LM algorithm, the training will continue, using new training data with some sequences (those with the biggest gradients) removed. This helps the algorithm avoid being trapped at the valleys in the error surface. We will demonstrate the performance of the modified algorithm on several test problems in the next section.

V. RESULTS

In this section, we will test the training procedure introduced in Section IV for two physical systems: magnetic levitation and double pendulum.

A. Magnetic Levitation

We will first develop a neural network plant model for a magnetic levitation system, and then we will use the neural network plant model to train a neural network controller. The control design will be based on the model reference adaptive control (MRAC) architecture (see Fig. 6), which was first introduced in [10].
The magnetic levitation system consists of a magnet suspended above an electromagnet, where the magnet is constrained so that it can only move in the vertical direction, as shown in Fig. 2.

The equation of motion of the magnet is

$$\frac{d^2 y(t)}{dt^2} = -g + \frac{\alpha}{M} i^2(t) \text{sgn}[i(t)] - \frac{\beta}{M} \frac{dy(t)}{dt}$$

(2)

where $y(t)$ is the distance of the magnet above the electromagnet, $i(t)$ is the current following in the electromagnet, $M$ is the mass of the magnet, $g$ is the gravitational constant, $\beta$ is a viscous friction coefficient and $\alpha$ is a field strength constant. Simulation parameters are given in Table I [11].

1) System description: The magnetic levitation system consists of a magnet suspended above an electromagnet, where the magnet is constrained so that it can only move in the vertical direction, as shown in Fig. 2.

2) System identification: First, we need a set of training data. We apply random inputs consisting of a series of pulses of random amplitude and duration. An example of training data is shown in Fig. 3. We use a nonlinear autoregressive with exogenous inputs (NARX) network (Fig. 4) to model this system. For this task, we use 3 input delays and 2 output delays (so the prediction begins with the fourth data point) and 10 hidden neurons. In the open-loop (one-step-ahead prediction) training phase, there are two inputs to the series-parallel network: the input sequence and the target sequence.

After finishing the open-loop training, we come back to the original NARX network for multiple-step-ahead prediction. The prediction horizon $k$ is gradually increased, based on the following list:


where $a : m : b$ means we go from $a$ steps to $b$ steps with the jump distance of $m$ steps. The selection of this list depends on how difficult the fitting problem is. In this particular example, at the beginning (the coarse tuning phase), we slowly increase $k$. Later, in the fine tuning phase, we could increase $k$ faster to save the training time.

Fig. 5 shows the results of the 1997-step-ahead prediction. We hardly see any difference between the actual position of the magnet and the position predicted by the NARX network, since the error is so small. We will use this model to train the controller in the next section.

3) Controller training: We will use the model reference adaptive control architecture ([10], [11]) in this section. There are two neural networks: the previously trained NARX network plant model and another network for the controller. The controller is trained so that the plant model output follows
the reference model output. The detailed MRAC structure is shown in Fig. 6.

There are three sets of controller inputs: delayed reference inputs, delayed controller outputs (plant inputs), and delayed plant outputs. The chosen number of delays for all three inputs is 2. (Typically, the number of delays increases with the order of the plant.) We use 10 hidden neurons for the controller. For the open-loop training phase, we use the targets (reference model outputs) wherever the plant model outputs are fed back. In other words, we open two feedback connections from the last (fourth) layer to the first layer and the third layer. Notice that we still have a feedback connection in the controller subnetwork. Since the plant model network has already been trained, its weights are fixed during the controller training process. We also set the initial output weights of the controller to zero, so it gives the plant zero initial input.

After successful open-loop training, we can follow the same steps we used for plant training (given in Section IV-C) to train the closed-loop network.

The result of controller training is shown in Fig. 7. We can see that the plant output matches the reference model output very well. (To verify the advantage of our new procedure, i.e., how incrementally increasing the prediction horizon can improve convergence, we trained the MRAC closed-loop network directly with the full-length training sequence (3997-step-ahead prediction). The training result is shown in Fig. 8, in which we can see that the search fails to converge to an accurate controller.)

Now we can test the operation by applying an arbitrary input to the trained MRAC network. We can see from Fig. 9 that the plant model output does follow the reference model input (and matches the reference model output). The result is good in both transient and steady-state regions, even though the input sequence is not the same as the input sequence in the training data.

B. Double Pendulum

Next, we will train a recurrent network to model a double pendulum. In this section, we only consider system identification and not controller design.

1) System description: A double pendulum (Fig. 10) is a physical system that exhibits rich dynamics and is sensitive to initial conditions. It is known to be chaotic.

If we define the state vector to be

\[ x = \begin{bmatrix} x_1 \\ x_2 \\ x_3 \\ x_4 \\ \end{bmatrix}, \quad y = \begin{bmatrix} q_1 \\ q_2 \\ f_1 \\ f_2 \end{bmatrix} \]

then equations of motion for the double pendulum can be written (from [12])

\[ \dot{x} = f(x) \] (4)

where \( f \) is given by

\[ f = \begin{bmatrix} x_3 \\ x_4 \\ f_3 \\ f_4 \end{bmatrix} \]

| TABLE II
| DOUBLE PENDULUM PARAMETERS |
|-----|-----|-----|-----|-----|-----|
| \( l_1 \) | \( l_2 \) | \( m_1 \) | \( m_2 \) | \( I_1 \) | \( I_2 \) |
| 1 m | 2 m | 1 kg | 2 kg | 0.1 kg m\(^2\) | 0.7 kg m\(^2\) |
where

\[
\begin{align*}
    f_3 &= \frac{1}{\gamma(x_2)} \left\{ c_1 c_3 (x_3 + x_4)^2 \sin(x_2) - c_2 c_4 g \sin(x_1) 
                        + c_3^2 x_3^2 \cos(x_2) \sin(x_2) + c_3 c_5 g \sin(x_1 + x_2) \cos(x_2) \right\} \\
    f_4 &= \frac{1}{\gamma(x_2)} \left\{ -[c_2 + c_3 \cos(x_2)](2x_3 x_4 + x_4^2) \sin(x_2) 
                        + c_2 c_4 g \sin(x_1 + x_2) - c_3 c_5 g \cos(x_2) \sin(x_1 + x_2) 
                        - [c_1 + c_2 + 2c_3 \cos(x_2)] x_3^2 c_3 \sin(x_2) \right\} \\
    \gamma(x_2) &= c_1 e_2 - c_3^2 \cos^2(x_2) \\
    c_1 &= m_1 l_1^2 + m_2 l_2^2 + I_1 \\
    c_2 &= m_2 l_1^2 + I_2 \\
    c_3 &= m_2 l_2^2 \\
    c_4 &= m_1 l_1 c_1 + m_2 l_2 \\
    c_5 &= m_2 l_2 c_2
\end{align*}
\]

The parameters for the double pendulum are given in Table II [12]. We assume that \( l_{c1} = l_1/2 \) and \( l_{c2} = l_2/2 \).

2) System identification: Because of the sensitivity of the double pendulum to initial conditions, we need to collect “enough” training data to adequately represent most dynamic behaviors. In this experiment, we used 20 different initial conditions in the range \([-\pi/2, \pi/2]\) for the two angles \( q_1 \) and \( q_2 \). Also, angles \( q_1 \) and \( q_2 \) are forced to be in the range \([-\pi, \pi]\). We use a NARX network with no input, 2 outputs, 4 output delays, and 10 hidden neurons for this task.

In the closed-loop training phase, \( \mu \) reached \( \mu_{\text{max}} \) quite often. Fig 11(a) shows an error profile (a cross section of the error surface along the gradient direction) when \( \mu \) reached \( \mu_{\text{max}} \). Clearly, there are many valleys in the error surface. At this point, we were doing 65-step-ahead prediction. There were 140 subsequences (which were segmented from the 20 original
sequences) in the training data pool. By checking the norm of gradients of individual sequences, we obtained the plot shown in Fig. 12. We can see that there is a sequence (sequence 14) whose gradient dominates the gradients of others. It turns out that this sequence contributes the most to the valleys in the error surface. By removing this sequence from the training data and plotting the error profile again at the same location, we obtained the new profile shown in Fig. 11(b). All the valleys disappear now. (Note that this new profile is very smooth, and may seem to be too good to be true. That is because we are plotting the profile over a very small region. Even in that small region, however, the original profile has many spurious valleys. This result is typical of the performance of the new proposed algorithm.) Because we have eliminated many of the spurious valleys, it will be easier for the algorithm to escape from this region.

Fig. 13 shows the performance of the modified LM algorithm. We can see that the SSE increases at certain epochs. (In the original LM algorithm, the SSE always decreases.) This occurs when \( \mu \) reaches \( \mu_{\text{max}} \) and the algorithm gets trapped in a valley. Since some sequences are removed at this point, the next step could produce weights where the SSE on the full data set does not decrease. The algorithm does escape from the valleys and eventually converges, as shown in Fig. 13. (Note that the standard LM algorithm would stop whenever \( \mu \) reached \( \mu_{\text{max}} \). There would be no way to continue with additional iterations. The algorithm would be permanently stuck, and would almost always be stuck at a weight location where the network fit is very poor.)

Table III shows the number of sequences that were removed at certain epochs when we were doing 65-step-ahead predictions. In this training stage, \( \mu \) reached \( \mu_{\text{max}} \) 9 times, and the algorithm had to remove at most 5 (out of 140) sequences
to escape from spurious valleys. Table IV summarizes the operation of the modified LM algorithm at different stages (different prediction horizons) when the search encountered spurious valleys. The table shows how many times $\mu$ reached $\mu_{\text{max}}$ and how many sequences (at most) were removed.

The trained network responses for 2 out of 20 sequences in the training data are shown in Fig. 14. We can see that even for a chaotic system, the 497-step-ahead prediction results are quite good. (Since we were using batch training, we were able to achieve a good fit for all 20 sequences.) The performance for a testing sequence is shown in Fig. 15. The network outputs still track the actual outputs quite well even for a new sequence (with new initial conditions).

### VI. Conclusions

In this paper, we have presented a new procedure for efficiently training general recurrent neural networks. The new procedure is a modified version of the Levenberg-Marquardt batch training algorithm. This procedure helps to avoid the numerous spurious valleys that appear in the error surface of recurrent networks. There are two features of the proposed procedure. First, the procedure detects when a search enters a spurious valley and removes sequences in the data set that cause the valley. Second, the procedure begins with short sequences (short prediction horizons) and then increases the sequence length during the training process. We have tested the procedure for two physical systems. In the first test, we developed a precise NARX model for a magnetic levitation system and trained an accurate MRAC neural controller. The second test dealt with a more difficult task, in which we trained a NARX model for a chaotic system - the double pendulum. We have shown in this task how the modified LM algorithm can handle the valleys in the error surface to achieve convergence. We believe that the problem of spurious valleys is one of the key difficulties that recurrent network training has to overcome. This paper suggests a possible solution to this problem.

### REFERENCES


### Table III

<table>
<thead>
<tr>
<th>Epoch</th>
<th>4</th>
<th>6</th>
<th>7</th>
<th>9</th>
<th>10</th>
<th>15</th>
<th>17</th>
<th>19</th>
<th>25</th>
</tr>
</thead>
<tbody>
<tr>
<td>No. seq</td>
<td>1</td>
<td>2</td>
<td>1</td>
<td>5</td>
<td>2</td>
<td>1</td>
<td>2</td>
<td>2</td>
<td>1</td>
</tr>
</tbody>
</table>

### Table IV

<table>
<thead>
<tr>
<th>$k$</th>
<th>Total no. sequences</th>
<th>No. times $\mu \rightarrow \mu_{\text{max}}$</th>
<th>Max no. sequences removed</th>
</tr>
</thead>
<tbody>
<tr>
<td>27</td>
<td>900</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>58</td>
<td>180</td>
<td>3</td>
<td>2</td>
</tr>
<tr>
<td>65</td>
<td>140</td>
<td>9</td>
<td>5</td>
</tr>
<tr>
<td>75</td>
<td>120</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>95</td>
<td>100</td>
<td>2</td>
<td>1</td>
</tr>
<tr>
<td>125</td>
<td>60</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>145</td>
<td>60</td>
<td>16</td>
<td>1</td>
</tr>
<tr>
<td>150</td>
<td>60</td>
<td>21</td>
<td>1</td>
</tr>
<tr>
<td>150</td>
<td>60</td>
<td>31</td>
<td>2</td>
</tr>
<tr>
<td>160</td>
<td>60</td>
<td>19</td>
<td>3</td>
</tr>
<tr>
<td>200</td>
<td>20</td>
<td>4</td>
<td>3</td>
</tr>
<tr>
<td>300</td>
<td>20</td>
<td>13</td>
<td>1</td>
</tr>
<tr>
<td>497</td>
<td>20</td>
<td>1</td>
<td>1</td>
</tr>
</tbody>
</table>