On-line Least-Squares Training For The Underdetermined Case

Roger L. Schultz, roger.schultz@halliburton.com, Halliburton Energy Services
Martin T. Hagan, mhagan@master.ceat.okstate.edu, Oklahoma State University

Abstract

In this paper we describe an on-line method of training neural networks which is based on solving the linearized least-squares problem using the pseudo-inverse for the underdetermined case. This Underdetermined Linearized Least Squares (ULLS) method requires significantly less computation and memory for implementation than standard higher-order methods such as the Gauss-Newton method or extended Kalman filter. This decrease is possible because the method allows training to proceed with a smaller number of samples than parameters. Simulation results which compare the performance of the ULLS algorithm to the recursive linearized least squares algorithm (RLLS) and the gradient descent algorithm are presented. Results showing the impact on computational complexity and squared-error performance of the ULLS method when the number of terms in the Jacobian matrix is varied are presented as well.

1. Introduction

First-order, stochastic gradient decent methods can be used in training neural networks adaptively, but they often exhibit poor performance when used with complex (i.e. nonlinear, recurrent) network structures. Standard higher-order optimization methods, such as the Gauss-Newton method or the extended Kalman filter, are used to solve the linearized least-squares problem using the pseudo-inverse for the overdetermined case. These methods generally perform much better than gradient descent methods, but involve numerical operations on square matrices which are proportional in size to the number of parameters in the network. In a typical filtering or control problem the tapped-delay line is used as the network input. If a fully connected neural network is used, the tapped-delay can easily contain enough taps to necessitate having a very large number of weights. The computational burden and memory requirements associated with training such a network can be prohibitive when implementing one of the standard higher-order optimization methods in a real-time system. Our first goal in this paper is to present an efficient ULLS method of training complex neural networks that is suitable for real-time implementation. Our second goal is to show a performance comparison between the ULS method and two other popular training methods. In the first section we give a summary of the linear least squares method for the overdetermined case and the underdetermined case. This development is then extended to the overdetermined and underdetermined cases for linearized least squares parameter estimation for nonlinear functions. In the next section we describe how to apply the ULLS algorithm to nonlinear neural networks. In the simulation section we apply the ULLS algorithm to a three-layer nonlinear network example. The same network is also trained using the recursive implementation of the overdetermined linearized least squares method and gradient descent method. The performance of all three training methods are then compared. The effect of varying the size of the Jacobian matrix is examined as well. The last section of the paper presents conclusions.

2. Linear Least Squares

One common method of estimating the unknown parameters of *linear* functions is the method of linear least squares [1]. In this method the function to be minimized is the familiar sum-of-squared-errors function given as:

$$F(\Theta) = \sum_{j=1}^{N} e_j^2(\hat{\underline{\Theta}}) = \underline{e}^T(\hat{\underline{\Theta}})\underline{e}(\hat{\underline{\Theta}}) \tag{1}$$

$$\text{where:} \qquad \underline{e}(\hat{\underline{\Theta}}) = [\underline{t} - \underline{a}(\hat{\underline{\Theta}})] \tag{2}$$

In Eq. (1) the vector $\underline{e}(\hat{\underline{\Theta}})$ represents the difference between a vector of desired outputs $\underline{t}$, and vector of outputs $\underline{a}(\hat{\underline{\Theta}})$, produced by a function of the form,

$$a(n) = \hat{\Theta}_1 p_1(n) + \hat{\Theta}_2 p_2(n) + \ldots + \hat{\Theta}_M p_M(n) \tag{3}$$

where $\hat{\underline{\Theta}}$ is a vector of parameter estimates. $p_1(n), p_2(n), \ldots, p_M(n)$ are multiplying entities which are simply scalar inputs at a specific index $\underline{n}$, but can be non-linear functions involving many system inputs. These functions cannot involve any of the parameters in $\hat{\underline{\Theta}}$. A radial basis function neural networks [2] is a common example of a function which fits this description. If we substitute Eq. (3) for all index values $(1 \ldots n \ldots N)$ into Eq. (2), and use Eq. (1) the following equation results:

$$F(\hat{\underline{\Theta}}) = [\underline{t} - \underline{P}\hat{\underline{\Theta}}]^T[\underline{t} - \underline{P}\hat{\underline{\Theta}}] \tag{4}$$

In Eq. (4) $\underline{P}$ is defined as:

$$\underline{P} = \begin{bmatrix} p_1(1) & p_2(1) & \dots & p_M(1) \\ p_1(2) & p_2(2) & \dots & p_M(2) \\ \vdots & \vdots & & \vdots \\ p_1(N) & p_2(N) & \dots & p_M(N) \end{bmatrix} \tag{5}$$

If an ideal a set of parameters exists, then,

$$\underline{P}\hat{\underline{\Theta}} = \underline{t} \tag{6}$$

and $\hat{\underline{\Theta}}$ represents a perfect solution to Eq. (6).

2.1 Overdetermined Linear Least Squares Solution

If there are more measurements available then there are unknown parameters for a system to be modeled, then the parameter estimation problem is said to be *overdetermined*. In this case we wish to minimize Eq. (4) by our choice of parameters. Eq. (4) can be rewritten as:

$$F(\hat{\underline{\Theta}}) = \underline{t}^T\underline{t} - 2\underline{t}^T\underline{P}\hat{\underline{\Theta}} + \hat{\underline{\Theta}}^T\underline{P}^T\underline{P}\hat{\underline{\Theta}} \tag{7}$$

Taking the gradient of Eq. (7) with respect to $\hat{\underline{\Theta}}$ yields:

$$\frac{\partial F(\hat{\underline{\Theta}})}{\partial\hat{\underline{\Theta}}} = -2[\underline{t}^T\underline{P}]^T + 2\underline{P}^T\underline{P}\hat{\underline{\Theta}} \tag{8}$$

To find the parameters which minimize Eq. (7), we set the equal to zero and solve for $\hat{\underline{\Theta}}$. The result is the formula below for computing the least squares estimate of the unknown parameters of a linear function in the overdetermined case.

$$\hat{\underline{\Theta}} = [\underline{P}^T\underline{P}]^{-1}\underline{P}^T\underline{t} \tag{9}$$

Notice that the matrix product appearing inside the inversion in Eq. (9) is of size M x M where M is number of parameters. The addition of a weighting matrix to Eq. (9) renders the *weighted* least squares [3] method, which can be described by:

$$\hat{\underline{\Theta}} = [\underline{P}^T\underline{W}\underline{P}]^{-1}\underline{P}^T\underline{W}\underline{t} \tag{10}$$

$\underline{W}$ is a diagonal matrix which causes the last measurements to be weighted more heavily than the preceding measurements. The weighting matrix is of the form,

$$\underline{W} = \begin{bmatrix} \ddots & \vdots & \vdots & \vdots \\ \dots & \Upsilon^2 & 0 & 0 \\ \dots & 0 & \Upsilon & 0 \\ \dots & 0 & 0 & 1 \end{bmatrix} \tag{11}$$

where Υ is a "forgetting" factor which is a positive number less than 1. The matrix inversion lemma [3] can be used to develop the familiar recursive least squares (RLS) method of computing the result of Eq. (10) recursively. This development will not be presented here but is the basis for the recursive Gauss-Newton method which will be used for comparison to the method of this paper. The RLS method is very useful in estimating unknown parameters in functions whose outputs are linearly related to the unknown parameters. This constraint renders this method of limited use in training neural networks, which often produce outputs which are not linearly related to the unknown network weights which are adjusted during training. In a later section the method of linearized least squares for use with nonlinear functions will be discussed.

2.1 Underdetermined Linear Least Squares Solution

If there are fewer measurements available then there are unknown parameters for a system to be modeled, then the parameter estimation problem is said to be *underdetermined*. When we have this situation, Eq. (9) cannot be used. The underconstrained nature of this problem dictates that a single unique solution does not exist. To remedy this we must constrain the problem sufficiently as to force a unique solution. A straight-forward way of accomplishing this is to minimize the sum of the squared parameters, while enforcing the following constraint:

$$\underline{P}\hat{\underline{\Theta}} = \underline{t} \tag{12}$$

In other words we want to perform the minimization with respect to $\hat{\underline{\Theta}}$, and $\underline{\lambda}$, where $\hat{\underline{\Theta}}$ is the vector of parameters and $\underline{\lambda}$ is a vector of Lagrange multipliers [4].

$$min\left\{\hat{\underline{\Theta}}^T\hat{\underline{\Theta}} + \underline{\lambda}^T[\underline{P}\hat{\underline{\Theta}} - \underline{t}]\right\} \tag{13}$$

Eq. (13) can be rewritten as:

$$min\left\{\hat{\underline{\Theta}}^T\hat{\underline{\Theta}} + [\underline{P}\hat{\underline{\Theta}} - \underline{t}]^T\underline{\lambda}\right\} \tag{14}$$

Taking the gradient of Eq. (14) with respect to $\hat{\underline{\Theta}}$ and $\underline{\lambda}$, respectively, and setting the result equal to zero yields:

$$\hat{\underline{\Theta}} + \underline{P}^T\underline{\lambda} = \underline{0} \tag{15}$$

and,

$$\underline{P}\hat{\underline{\Theta}} - \underline{t} = \underline{0} \tag{16}$$

Solving Eq. (15) for $\underline{\lambda}$ we get,

$$\underline{\lambda} = -[\underline{P}\underline{P}^T]^{-1}\underline{P}\hat{\underline{\Theta}} \tag{17}$$

but from Eq. (16) we know that:

$$\underline{P}\hat{\underline{\Theta}} = \underline{t} \tag{18}$$

Substituting Eq. (18) into Eq. (17) yields:

$$\underline{\lambda} = -[\underline{P}\underline{P}^T]^{-1}\underline{t} \tag{19}$$

Eq. (19) can now be plugged into Eq. (15) and the result can be solved for $\hat{\underline{\Theta}}$ to show that:

$$\hat{\underline{\Theta}} = \underline{P}^T[\underline{P}\underline{P}^T]^{-1}\underline{t} \tag{20}$$

Eq. (20) is the solution to the underdetermined linear least squares problem. Notice that the matrix product which appears inside the inversion of Eq. (20) is an M x M matrix where M is the number of sample contributions to the $\underline{P}$ matrix. The outer product form of Eq. (20) does not lend itself to the insertion of a traditional exponential weighting matrix or to the application of the matrix inversion lemma. However if a small window of sample contributions are used in forming the $\underline{P}$ matrix, the size of the resulting matrix to be inverted in Eq. (20) is small. This opens the possibility of computing parameter updates in "real-time" with fewer computations than with the RLS algorithm.

3.0 Linearized Least-Squares

The least-squares method may be used in minimizing non-linear functions by first linearizing the function about a nominal set of parameter values, then applying the standard least-squares equations to calculate an incremental parameter adjustment. This process is repeated until a specified convergence criteria has been satisfied. This modification of the least-squares method is known as the linearized least-squares method or the iterated least-squares method [5]. This modification of the linear least-squares method was developed to overcome the constraint of linearity between a function output and the unknown parameters in the function which is to be minimized. This method can be applied to overdetermined and underdetermined problems.

3.1 Overdetermined Linearized Least Squares Solution

A development of the iterated least-squares method begins with the familiar equation,

$$F(\underline{\Theta}) = \sum_{j=1}^{N} e_j^{\,2}(\underline{\hat{\Theta}}) = \underline{e}^T(\underline{\hat{\Theta}})\underline{e}(\underline{\hat{\Theta}}) \qquad (21)$$

where: $$\underline{e}(\underline{\hat{\Theta}}) = [\underline{t} - \underline{a}(\underline{\hat{\Theta}})] \qquad (22)$$

which has been explained in the preceding sections. Now suppose that $\underline{a}(\underline{\Theta})$ is a vector of outputs of a nonlinear function of known inputs and unknown function parameters $\underline{\Theta}$ which must be determined, and $\underline{t}$ is a vector of desired outputs corresponding to the known inputs to the function. Now we must linearize $\underline{a}(\underline{\Theta})$. The incremental change in the vector of function outputs for an incremental adjustment in the function parameters is represented by Eq. (23).

$$\underline{a}(\underline{\hat{\Theta}}_{n+1}) = \underline{a}(\underline{\hat{\Theta}}_n + \underline{\delta\hat{\Theta}}_n) \qquad (23)$$

In Eq. (23) $\underline{\delta\hat{\Theta}}_n$ is the vector of incremental adjustments of the estimated parameters. Taking the first-order Taylor series expansion of Eq. (23) produces Eq. (24).

$$\underline{a}(\underline{\hat{\Theta}}_{n+1}) \approx \underline{a}(\underline{\hat{\Theta}}_n) + \left[\nabla a(\underline{\hat{\Theta}}_n)\Big|_{\underline{\Theta} = \underline{\Theta}_n}\right]^T \underline{\delta\hat{\Theta}}_n \qquad (24)$$

where:

$$\nabla\underline{a}(\underline{\hat{\Theta}}_n) = \begin{bmatrix} \frac{\partial}{\partial\hat{\Theta}_{1,n}}a_1(\underline{\hat{\Theta}}_n) & \frac{\partial}{\partial\hat{\Theta}_{1,n}}a_2(\underline{\hat{\Theta}}_n) & \cdots & \frac{\partial}{\partial\hat{\Theta}_{1,n}}a_N(\underline{\hat{\Theta}}_n) \\ \frac{\partial}{\partial\hat{\Theta}_{2,n}}a_1(\underline{\hat{\Theta}}_n) & \frac{\partial}{\partial\hat{\Theta}_{2,n}}a_2(\underline{\hat{\Theta}}_n) & \cdots & \frac{\partial}{\partial\hat{\Theta}_{2,n}}a_N(\underline{\hat{\Theta}}_n) \\ \vdots & \vdots & & \vdots \\ \frac{\partial}{\partial\hat{\Theta}_{M,n}}a_1(\underline{\hat{\Theta}}_n) & \frac{\partial}{\partial\hat{\Theta}_{M,n}}a_2(\underline{\hat{\Theta}}_n) & \cdots & \frac{\partial}{\partial\hat{\Theta}_{M,n}}a_N(\underline{\hat{\Theta}}_n) \end{bmatrix}^T \qquad (25)$$

In Eq. (25) M is the number of unknown parameters, N is the number of input/target-output data pairs, and n is the index which corresponds to the iteration of the data set presentation. Using Eq. (24), Eq. (21) can be written as:

$$F(\underline{\hat{\Theta}}) \approx [\underline{t} - \underline{a}(\underline{\hat{\Theta}}_n) + \nabla\underline{a}(\underline{\hat{\Theta}}_n)\underline{\delta\hat{\Theta}}_n]^T [\underline{t} - \underline{a}(\underline{\hat{\Theta}}_n) + \nabla\underline{a}(\underline{\hat{\Theta}}_n)\underline{\delta\hat{\Theta}}_n] \qquad (26)$$

Rewriting Eq. (26) in terms of the function output error we have:

$$F(\underline{\hat{\Theta}}) \approx [\underline{e}(\underline{\hat{\Theta}}_n) + \nabla\underline{a}(\underline{\hat{\Theta}}_n)\underline{\delta\hat{\Theta}}_n]^T [\underline{e}(\underline{\hat{\Theta}}_n) + \nabla\underline{a}(\underline{\hat{\Theta}}_n)\underline{\delta\hat{\Theta}}_n] \qquad (27)$$

Rearranging Eq. (27) we have:

$$F(\underline{\hat{\Theta}}) \approx \underline{e}(\underline{\hat{\Theta}}_n)^T\underline{e}(\underline{\hat{\Theta}}_n) + 2\underline{e}(\underline{\hat{\Theta}}_n)^T\nabla\underline{a}(\underline{\hat{\Theta}}_n)\underline{\delta\hat{\Theta}}_n + \underline{\delta\hat{\Theta}}_n^T\nabla\underline{a}(\underline{\hat{\Theta}}_n)^T\nabla\underline{a}(\underline{\hat{\Theta}}_n)\underline{\delta\hat{\Theta}}_n \qquad (28)$$

Taking the vector derivative of Eq. (28) with respect to $\underline{\delta\hat{\Theta}}_n$ yields:

$$\frac{\partial F(\underline{\hat{\Theta}}_n)}{\partial\underline{\hat{\Theta}}_n} \approx 2[\underline{e}(\underline{\hat{\Theta}}_n)^T\nabla\underline{a}(\underline{\hat{\Theta}}_n)]^T + 2\nabla\underline{a}(\underline{\hat{\Theta}}_n)^T\nabla\underline{a}(\underline{\hat{\Theta}}_n)\underline{\delta\hat{\Theta}}_n \qquad (29)$$

To find the changes in the parameters which minimize Eq. (28), we set the vector derivative given by Eq. (29) equal to zeros and solve for $\underline{\delta\hat{\Theta}}_n$. The result is Eq. (30) for computing the least-squares estimate of the incremental parameter adjustment needed to minimize the function. The function has been linearized about the current parameter values.

$$\underline{\delta\hat{\Theta}}_n = -[\nabla\underline{a}(\underline{\hat{\Theta}}_n)^T\nabla\underline{a}(\underline{\hat{\Theta}}_n)]^{-1}\nabla\underline{a}(\underline{\hat{\Theta}}_n)^T\underline{e}(\underline{\hat{\Theta}}_n) \qquad (30)$$

In essence, the linear least-squares solution to the linearized system is computed about the current set of parameter estimates to find the appropriate vector of parameter adjustments which are then used in Eq. (31) to compute an updated vector of parameter estimates.

$$\underline{\hat{\Theta}}_{n+1} = \underline{\hat{\Theta}}_n + \underline{\delta\hat{\Theta}}_n \qquad (31)$$

Eq. (30) can be inserted into Eq. (31) to obtain:

$$\underline{\hat{\Theta}}_{n+1} = \underline{\hat{\Theta}}_n - [\nabla\underline{a}(\underline{\hat{\Theta}}_n)^T\nabla\underline{a}(\underline{\hat{\Theta}}_n)]^{-1}\nabla\underline{a}(\underline{\hat{\Theta}}_n)^T\underline{e}(\underline{\hat{\Theta}}_n) \qquad (32)$$

The same result can be derived using the Gauss-Newton approach [6]. Upon close examination of the Gauss-Newton and iterated least-squares methods it can be seen that the two methods are in fact equivalent. A recursive method of implementing Eq. (32) can be developed in much the same way as the RLS algorithm. The recursive linearized least squares method will be referred to as the RLLS method henceforth.

3.1 Underdetermined Linearized Least Squares Solution

A linearized least squares solution can be obtained for underdetermined nonlinear optimization problems. A development similar to that given in section 2.2 can be used with the perturbation equations introduced in section 3.1 to develop a linearized least squares solution for the underdetermined case. This derivation will not be shown here, but the results can be summarized as:

$$\underline{\delta\hat{\Theta}}_n = -\nabla\underline{a}(\underline{\hat{\Theta}}_n)[\nabla\underline{a}(\underline{\hat{\Theta}}_n)\nabla\underline{a}(\underline{\hat{\Theta}}_n)^T]^{-1}\underline{e}(\underline{\hat{\Theta}}_n) \qquad (33)$$

$$\underline{\hat{\Theta}}_{n+1} = \underline{\hat{\Theta}}_n + \underline{\delta\hat{\Theta}}_n \qquad (34)$$

$$\underline{\hat{\Theta}}_{n+1} = \underline{\hat{\Theta}}_n - \nabla\underline{a}(\underline{\hat{\Theta}}_n)[\nabla\underline{a}(\underline{\hat{\Theta}}_n)\nabla\underline{a}(\underline{\hat{\Theta}}_n)^T]^{-1}\underline{e}(\underline{\hat{\Theta}}_n) \qquad (35)$$

As with the underdetermined linear least squares method the size of the matrix which must be inverted in Eq. (35) is dependent not on the number parameters, but on the number of measurements being considered. This method will be applied in the next section to the training of nonlinear multi-layer neural networks.

3. Incremental ULLS Network Training

In this section a method of using the underdetermined linearized least squares (ULLS) approach to incrementally train nonlinear neural networks will be presented. The method relies on applying Eq. (35), where the parameters $\underline{\Theta}$ are the network weights $\underline{w}$, and gradient matrix $\nabla \underline{a}(\hat{\underline{\Theta}}_n)$ is a windowed Jacobian matrix [6] $\underline{J}$ for the network. The equation for a single update of the network weights can be written as:

$$\underline{w}_{i+1} = \underline{w}_i - [\underline{J}_i[\underline{J}_i\underline{J}_i^T]^{-1}\underline{e}_i]\alpha \tag{36}$$

In Eq. (36), α is a small positive number less than one, which serves as a weight adjustment step size or damping factor. The Jacobian matrix used in Eq. (36) is updated at each index in time to reflect a window of past measurements. This update can be described by Eq. (37) below where $\underline{j}_i$ is the most recently computed row in the Jacobian matrix.

$$\underline{J}_{i+1} = \begin{bmatrix} \leftarrow & \underline{j}_{i-M+1} & \rightarrow \\ & \vdots & \\ \leftarrow & \underline{j}_{i-1} & \rightarrow \\ \leftarrow & \underline{j}_i & \rightarrow \end{bmatrix} \tag{37}$$

The oldest row of the previous Jacobian matrix is removed in Eq. (37). This windowing effect causes the algorithm to “forget” the effects of older measurements, making the method suitable for adaptive network training. As stated previously, the required matrix inversion in Eq. (36) is of order M, which is the number of measurements considered in the current Jacobian matrix. This is of particular importance in training neural networks. A fully-connected neural network can easily contain a large number of weights. This is especially true in applications in which tapped-delay inputs are used. A large number of weights can be required to accommodate the inputs from even a modest length tapped-delay line. Unfortunately, tapped-delay lines are commonly used in real-time operations where conventional recursive linearized least squares training of networks containing a large number of weights is impractical because of computational requirements. These methods do not require direct inversion of large matrices, but do require multiplication and addition operations on matrices which are of order M, the number of network parameters. The ULLS method of training networks requires a direct matrix inversion of order N, the number of measurements to be considered in the Jacobian matrix window. As we will see in the following section, the number of measurements considered for each weight update does have an impact on the algorithm performance as well as computational requirements.

4. Simulation Results

In this section simulation results for training a typical neural network using three different training methods will be compared. There will also be results presented which show the impact on performance when various numbers of past measurements are considered when training using the ULLS method. The three methods which will be compared are gradient descent, recursive linearized least squares and the ULLS method. The performance of the methods will be evaluated and compared based on speed of convergence, squared-error performance after convergence, and computational requirements. The network used in the comparison was a three-layer nonlinear network with 20 inputs, 10 log-sigmoid neurons in the first hidden layer, 3 log-sigmoid neurons in second hidden layer, and a single linear output neuron. The network was fully connected and had a total of 247 weights including biases. Training data was obtained by fixing the network weights at random values and running a forward simulation using a tapped delay structure of random inputs. This data was then used in training simulations. For each training simulation the network weights were initialized to the same set of small random numbers. The performance of the fixed learning rate gradient descent method was optimized by running simulations with many different learning rates. The comparisons of the methods which follow are based on the gradient descent results when using the optimum learning rate. The implementation of the ULLS method used a Jacobian matrix based on a window of 10 past measurements. All simulation results shown here reflect this implementation unless otherwise noted. Figure 1 shows the mean squared error for a 200 point window computed at each index in the 3000 training points.

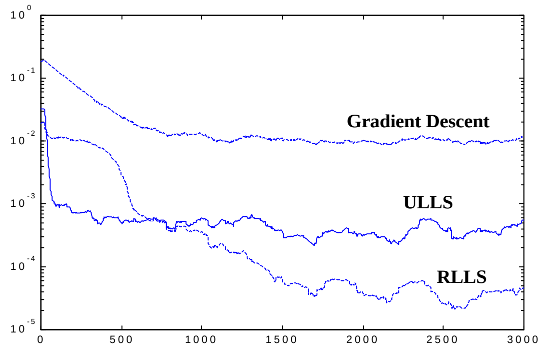

Figure 1 Mean-Squared Error Performance

We can see from Figure 1 that the gradient descent method converges more slowly and finds a poorer solution than either the RLLS method or the ULLS method. It can also be seen the RLLS method converges with a lower mean-squared error than the ULLS method. However it is very significant that the ULLS method converges much more quickly than the RLLS method. This is of great importance in adaptive systems where the model is constantly changing. The superior squared-error performance of the RLLS and ULLS methods is paid for in computational complexity as indicated in Table 1.

Training Method	Mean Squared Error (Converged)	Flops/Sample
Gradient Descent	1.00×10^{-2}	1.87×10^{3}
ULLS	1.97×10^{-4}	1.19×10^{5}
RLLS	3.77×10^{-5}	2.28×10^{7}

Table 1 Training Method Performance Comparison

These results indicate that with the methods considered there is improved ultimate squared-error performance as the computational complexity increases. Most significantly, we can see that the ULLS method exhibits the fastest convergence performance on a sample by sample basis. Figure 2 is plot of squared-error vs. floating point operations.

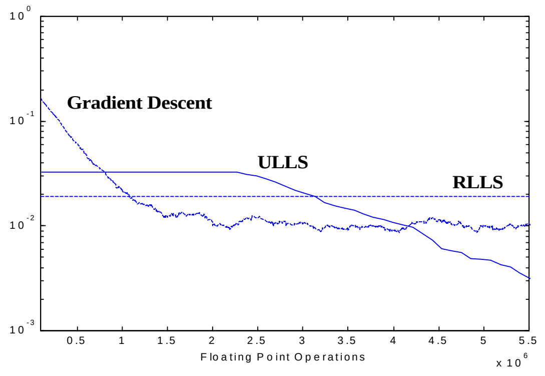

Figure 2 Squared-Error vs. Flops

This plot shows that the gradient descent method exhibits the fastest convergence on a floating point operations basis. Notice that the ULLS algorithm can drive the mean squared error much lower than the minimum error achieved by the steepest descent algorithm with less than an order of magnitude increase in required floating point operations.

The next set of simulation results reflects the effect of changing the size of the window of measurements considered in the ULLS algorithm. The network structure and data set used for this simulation were the same as for the previous simulations. The size of the Jacobian matrix was varied from run to run. Figure 3 shows the squared-error performance when considering different Jacobian matrix sizes. In this example there was little improvement in the squared-error performance when more than about 10 measurements were used in the Jacobian matrix window. Table 2 contains a summary of the ULLS algorithm performance for these simulations. There is a tremendous increase in computational burden each time the number of terms in the Jacobian is doubled. If only two terms are used there is a dramatic increase in performance for only a four-fold increase in computational burden. This might be an attractive improvement for systems with limited computational capacity which currently use gradient descent training.

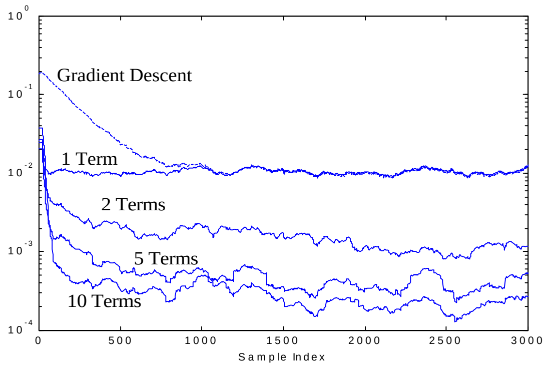

Figure 3 Effect of Jacobian Size in ULLS Performance

Jacobian Terms	Mean Squared Error (Converged)	Flops/Sample
Gradient Descent	1.00×10^{-2}	1.87×10^{3}
1	1.02×10^{-2}	3.08×10^{3}
2	1.03×10^{-3}	7.45×10^{3}
5	3.38×10^{-4}	3.22×10^{4}
10	1.97×10^{-4}	1.19×10^{5}

Table 2 Jacobian Terms Performance Comparison

5. Conclusions

A method for training neural networks on-line, using the solution to the underdetermined linearized least-squares problem, has been presented. Our work shows that, for training feed-forward multi-layer neural networks on a sample-by-sample basis, the ULLS method we have presented here exhibits faster convergence than two of the standard real-time training methods. The ULS method also exhibits good final convergence performance, although not as good as the recursive linearized least squares (RLLS) method. Additionally, we have shown that the ULLS method requires far fewer computations per input sample than the RLLS method. Because we have employed the solution of the underdetermined least-squares problem in our method, it is possible to consider fewer input samples than network parameters in calculating weight updates. This dramatically reduces the computational burden and memory requirements over second-order on-line training methods.

6. References

[1] A. Bjorck, *Numerical Method for Least Squares Methods*, Siam, 1996.

[2] S. Haykin, *Adaptive Filter Theory*, 3rd ed., Prentice Hall, Upper Saddle River, N.J., 1996.

[3] J. M. Mendel, *Lessons in Digital Estimation Theory*, 2nd ed., Prentice-Hall, Englewood CLiffs, N.J., 1995

[4] G.H. Golub, C.F. Van Loan, Matrix Computations, 3rd ed., John Hopkins, Baltimore and London

[5] L. Ljung, *System Identification: Theory for the User*, Prentice Hall, Engelwood Cliffs, N.J.

[6] M. T. Hagan, H. B. Demuth and M. Beale, *Neural Network Design*, Boston: PWS Publishing Co., 1996.