Gauss-newton Based Learning For Fully Recurrent Neural Networks

2004

Aniket Arun Vartak
University of Central Florida

GAUSS-NEWTON BASED LEARNING FOR FULLY RECURRENT NEURAL NETWORKS

by

ANIKET A. VARTAK
B.S. University of Mumbai, 2002

A thesis submitted in partial fulfillment of the requirements
for the degree of Master of Science
in the Department of Electrical and Computer Engineering
in the College of Engineering and Computer Science
at the University of Central Florida
Orlando, Florida

Summer Term
2004

ABSTRACT

The thesis discusses a novel off-line and on-line learning approach for Fully Recurrent Neural Networks (FRNNs). The most popular algorithm for training FRNNs, the Real Time Recurrent Learning (RTRL) algorithm, employs the gradient descent technique for finding the optimum weight vectors in the recurrent neural network. Within the framework of the research presented, a new off-line and on-line variation of RTRL is presented, that is based on the Gauss-Newton method. The method itself is an approximate Newton's method tailored to the specific optimization problem, (non-linear least squares), which aims to speed up the process of FRNN training. The new approach stands as a robust and effective compromise between the original gradient-based RTRL (low computational complexity, slow convergence) and Newton-based variants of RTRL (high computational complexity, fast convergence). By gathering information over time in order to form Gauss-Newton search vectors, the new learning algorithm, GN-RTRL, is capable of converging faster to a better quality solution than the original algorithm. Experimental results reflect these qualities of GN-RTRL, as well as the fact that GN-RTRL may have in practice lower computational cost in comparison, again, to the original RTRL.

ACKNOWLEDGMENTS

First of all, I would like to thank my academic advisors, Dr. M. Georgiopoulos and Dr. G. Anagnostopoulos. Their approach to the research process has helped me learn many new things required to carry out this kind of research. I also want to extend my warmest gratitude for being patient, and interested in my progress throughout this research effort. I am privileged working with them in this research effort, and I will always be thankful for this. I would also like to extend my sincere thanks to my committee members Dr. Kasparis, and Dr. Haralambous for their time and efforts for reviewing my work and providing with valuable feedback for this document to be as accurate and as through as possible. Finally I would like to thank my parents Arun Vartak and Vrushali Vartak and my sister Mrunmayee Vartak for always being a source of energy and support.

TABLE OF CONTENTS

LIST OF FIGURES

LIST OF TABLES

LIST OF ABBREVATIONS

J - Jacobian matrix

$g(X)$ - Gradient vector

G(X) - Hessian matrix

p - Search direction vector

U – Number of nodes in output layer

L – Number of observable nodes

V – Number of input layer nodes

I – Number of input nodes

H – Number of hidden nodes

W – Weight matrix

θ - Column vector of all adaptable parameters

$\phi(\cdot)$ - Objective function (sum of squared errors)

$e_k(t)$ - Error between desired and actual outputs of node ‘k’ at time‘t’

$d_k(t)$ - Desired output of node ‘k’ at time‘t’

$y_k(t)$ - Output of node ‘k’ at time‘t’

$r(t)$ - Residual vector

1: INTRODUCTION

Recurrent neural networks (RNN) are very effective in learning temporal data sequences, due to their feed back connections. These feed back connections make the RNN different from the feed forward networks. Due to the use of these recurrent connections we add another dimension to our network, which is ‘time’. The hidden units, whose output is of no immediate interest, act as dynamic memory units, and information about the previous time instances is used to calculate the information at present time instant. The following figure illustrates the structural differences between recurrent neural networks and their more popular counterparts, feedforward neural networks.

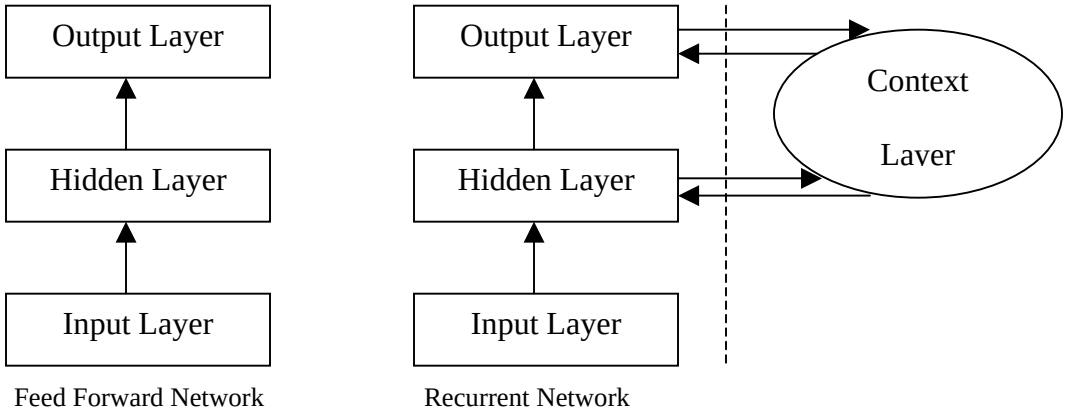
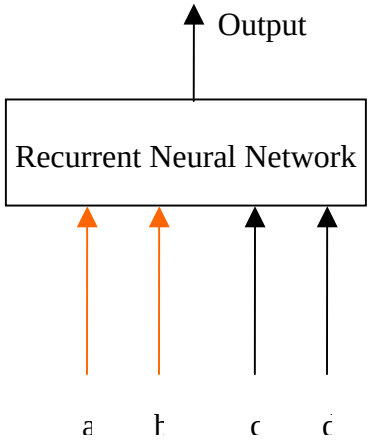


Figure 1: Structural differences in the feed forward and recurrent networks

In the structure of a recurrent neural network, depicted in Figure 1, the context layer acts as a

buffer that stores the past information about the data. Due to this structure the recurrent neural network is very effective when the data set is a time varying signal.

In the following we present an example, from the experiments with the Real Time Recurrent Learning (RTRL) paper (Williams, Zipser ‘89B), that demonstrates the power of the recurrent neural networks to learn a temporal task.



Time	**a**	**b**	**c**	**d**	Output
1	**1**	0	0	0	0
2	0	0	1	0	0
3	0	**1**	0	0	**1**
4	0	0	0	1	0
5	0	**1**	0	0	**0**
6	**1**	0	0	0	0
7	0	**1**	0	0	**1**

Figure 2: Example demonstrating power of RNN to learn temporal tasks

Consider the above system, whose output becomes 1 only when a particular pattern of lines appear in a specific order. As shown in the table, the output is 1 when line “b” is turned on after some time line “a” was on; other wise the output is off. The lines “c” and “d” are distractors. This task consists of recognizing events in a specific order (that is “a” then “b”), regardless of the

number of intervening events (that is “c” or “d”). Due to the feedback structure of the recurrent neural network, where the inputs consist of the external inputs as well as the delayed outputs, this task is accomplished efficiently. In order for the same task to be accomplished by a feed forward neural network a tapped delay line is required to store the past pattern information. Due to the finite number of tapped delay lines this network will fail to achieve its goal for arbitrary number of intervening “c”’s and “d”’s .in between a sequence of “a”, then “b”.

In general, recurrent neural networks can effectively address tasks that contain some sort of time element in them. Examples of these tasks include, but are not limited to, stock market prediction, speech recognition, learning formal grammar sequences, one-step-ahead prediction.

There are different architectures of recurrent neural networks. These architectures vary in the way they feed the outputs of the units (nodes) in the network, at a particular time instance, as inputs to the same units (nodes) in the network, at a future time instance. For example, the Elman recurrent neural network feeds back from each unit in its hidden layer to each other unit in the hidden layer. On the other hand, the fully recurrent neural network (FRNN) has the output of every node in the network connected to all the other nodes in the network (see figure 2 below for a block diagram of FRNN).

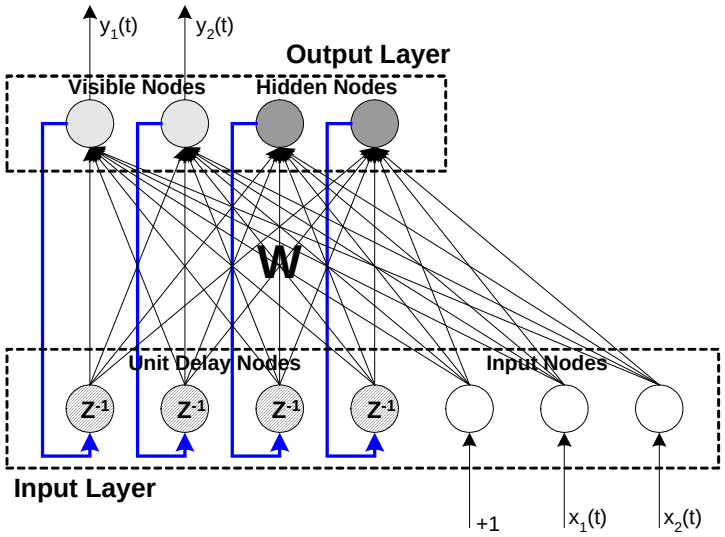


Figure 3: Block diagram of a Fully Recurrent Neural Network (FRNN)

The FRNN consists of an input layer and an output layer. The input layer is fully connected to the output layer via adjustable, weighted connections, which represent the system's training parameters (weights). The inputs to the input layer are signals from the external environment or unit-gain, unit-delay feedback connections from the output layer to the input layer. FRNNs accomplish their task by learning a mapping between a set of input sequences to another set of output sequences. In particular, the nodes in the input layer of an FRNN accept input sequences from the outside world, delayed output activations from the output nodes of the network and a constant-valued node that helps in serving as the bias node for all the output nodes in the network. On the other hand, the nodes in the output layer generate the set of output sequences. Typically, nodes in the output layer are distinguished as output nodes (that produce the desired outputs) and hidden nodes, whose activations are not related to any of the outputs of the task to

be learned, but act as a secondary, dynamic memory of the system. The feedback connections (from output layer to input layer) in a recurrent neural network is the mechanism that allows the network to be influenced not only by the currently applied inputs but also by past applied inputs; this feature gives recurrent networks the power to effectively learn relationships between temporal sequences.

Simple feed-forward neural networks are usually trained by the popular back-propagation algorithm (Rumelhart et al. ’86). The recurrent neural networks on the other hand, due to their dynamic processing require more complex algorithms for learning. An attempt to extend the back-propagation technique for learning in recurrent networks has led into a learning approach, termed *back-propagation through time* (Werbos ‘90). The implementation involved unfolding the recurrent network in time so that it grows one layer at each time step. This approach has the disadvantage of a requiring a memory size that grows large, especially for arbitrarily long training sequences. Another popular algorithm for learning in FRNN is the Real Time Recurrent Learning (RTRL) algorithm (Williams & Zipser ‘89). The main emphasis of the algorithm is to learn the temporal sequences by starting from a network topology that takes into consideration the knowledge about the temporal nature of the problem. RTRL is a gradient descent-based algorithm that is used for the adjusting the network’s interconnection weights. Williams & Zipser present two variations of RTRL, one for off-line (batch) and another one for on-line (incremental) learning. In both of its forms, RTRL has been successfully used to train FRNNs for a variety of applications, such as speech recognition and controller modeling.

2: ORGANIZATION OF THE THESIS

The thesis is organized as follows: The first section (Section 2.1) is the literature review. It focuses on recurrent neural networks and several approaches for learning the interconnection weights of recurrent neural networks. It also focuses on different variants of the RTRL introduced into the literature, as well as successful applications of the RTRL neural networks. In section 2.2 we discuss the contribution of this thesis in the field of recurrent neural network learning. It also discusses the motivation behind our approach for training these kinds of networks. In section 2.3 the theory behind the original RTRL (Williams & Zipser ‘89) is thoroughly discussed, including the formation of the direction vector, finding the adaptive learning rate etc. Furthermore, in section 2.4 we discuss the derivation of the RTRL algorithm based on the Gauss-Newton direction. In section 3 we elaborate on the data sets we used for the experimentation and the associated experimental results. In the last section of the thesis (section 4) conclusive remarks are provided.

2.1 Literature Review

There are several algorithms that have been developed to training recurrent neural networks, the principal ones being the real-time recurrent learning (RTRL) (Williams & Zipser ‘89), backpropagation-through-time (BPTT) (Werbos ’90) and the extended Kalman filter (Williams ’92). All these algorithms make use of the gradient of the error function with respect to the

weights to perform the weight updates. The specific method in which the gradient is incorporated in the weight updates distinguishes the different methods. The RTRL algorithm has several advantages as the gradient information is computed by integrating forward in time as the network runs, as apposed to the BPTT algorithm where the computation is done by integrating backward in time as the network takes a single step forward.

RTRL is a gradient-based algorithm that is used for the modification of the network's interconnection weights. Williams & Zipser present two variations of RTRL, one for off-line (batch) and one for on-line (incremental) learning. Using both variations, RTRL has been used to train FRNNs in a variety of applications, such as speech recognition, controller modeling, amongst others. Specifically, RTRL has been used for training a robust, manufacturing process controller in (Hambaba '00). The problem of speech enhancement and recognition is addressed in (Juang & Lin '01), where RTRL is used to construct adaptive fuzzy filters. RTRL has also been used to train FRNNs for next-symbol prediction in an English text processing application (Perez-Oritz et al. '01). The RTRL/FRNN combination has also been used in applications of communication systems. (Li, et al. '02) use FRNNs, trained by RTRL, for adaptive pre-distortion linearization of RF amplifiers and they have been shown to attain superior performance, in comparison with other well-known pre-distortion models. Furthermore, the authors show significant improvements in the Bit Error Rate (BER) performance as compared with linear techniques in the field of digital, mobile-radio systems. Finally, RTRL has been used to train FRNNs to effectively removing artifacts in EEG (Electroencephalograms) signals (Selvan &

Srinivasan ‘00).

A plethora of RTRL variants or substitutes have been suggested in the literature that aimed to enhance different aspects of the training procedure such as its computational complexity, especially when used in off-line mode, convergence speed/properties, and its sensitivity to the choice of initial weight values. In (Catfolis ‘93) a technique is presented for reinitializing RTRL after a specific time interval, so that weight changes depend on fewer past values and weight updates follow more precisely the error gradient. Also, the relationship between some inherent parameters, like the slope of the sigmoidal activation functions and the learning rate, has been taken into account to reduce the degrees of freedom of the associated non-linear optimization problems (Mandic & Chambers ‘99). In (Schmidhuber ‘92), the gradient calculation has been decomposed into blocks to produce an algorithm which is an order of magnitude faster than the original RTRL. Additional constraints have been imposed to the synaptic weight matrix to achieve reduced learning time, while the network forgetting is reduced (Druaux, et al. ‘98). In (Chang & Mak ‘98), a conjugate-gradient variation of RTRL has been developed. Other techniques suggested to improve the convergence rate include use of Normalized RTRL (Mandic & Chambers ‘00), and use of genetic algorithms (Mak, et al. ‘98) and (Blanco, et al. ‘01). It has been shown (Atiya & Parlos ‘00) that of the training approaches like backpropagation through time (BTT), RTRL, fast forward propagation approach (what training approaches?...be specific) are based on different computational ways to efficiently obtain the gradient of error function and can be generally grouped into five major groups. Furthermore, these five approaches are only

five different ways of solving particular matrix equation. Dynamic sub-grouping of processing elements has also been proposed to reduce the RTRL's computational complexity from $O(n^4)$ to $O(n^2)$. (Euliano & Principe '00). Finally, Newton-based approaches for small FRNNs have also been used to exploit the Newton's method quadratic rate of convergence (Coelho '00).

In this paper, we present a novel, on-line RTRL variation, namely the GN-RTRL. While the original RTRL training procedure utilizes gradient information to guide the search towards the minimum training error (and therefore we are going to refer to it as GD-RTRL), GN-RTRL uses the Gauss-Newton direction vector for the same purpose. The development of a GN-based training algorithm for FRNNs was motivated by the very nature of the optimization problem at hand. The function to be minimized is non-linear squared-error type, which makes it a Non-linear Least-Squares (NLS) optimization problem. While gradient descent methods are straightforward and easy to implement for NLS problems, their convergence rate is linear (Nocedal & Wright '99), which typically translates to long training times. The problem is further worsened when the model size (the number of interconnection weights) increases. On the other side of the spectrum, Newton-based methods attain a theoretical quadratic rate of convergence (Dennis & Schnabel '83), which makes them appealing from the perspective of achieving reduced training time. Nevertheless, Newton-based algorithms require second-order derivative information, such as the associated Hessian matrix or approximations to it. This requirement implies a high computational cost, which prohibits the usability of Newton-based learning for moderate or large size FRNN structures. We propose an RTRL scheme based on the Gauss-Newton method as a compromise between gradient descent and Newton's methods. The GN-

RTRL features a super-linear convergence profile (faster than GD-RTRL) and lower computational cost to second-order algorithms, which makes it practical for training small to moderate size FRNNs.

2.2 Contribution of the Thesis -Background Information

The thesis discusses a novel approach to learning in the fully recurrent neural networks. The most popular algorithm for learning, the RTRL employs the gradient descent technique of minimization for finding the optimum weight vectors in the recurrent neural network. By using an approximation to the Newton's method, i.e. Gauss-Newton method, which is suitable for non-linear least squares minimization problems we can speed up the process of learning the recurrent weights.

2.2.1 Minimization Techniques in non-linear Least Squares Problem

If an objective function is expressed as a sum of squares of other non-linear functions:

$$F(X) = \sum_{i=1}^{m} f_i^2(X), \qquad (1)$$

then it is possible to devise some efficient methods to minimize it.

If we gather the functions f_i in vector form as:

$$\mathbf{f}(X) = [f_1 f_2 \Lambda \ f_m]^T \text{ then } F(X) = \mathbf{f}^T(X)\mathbf{f}(X) \qquad (2)$$

To obtain the gradient vector of the objective function, the first partial derivative with respect to x_j can be written as:

$$g_j = 2\sum_{i=1}^{m} f_i \frac{\partial f_i}{\partial x_j} \tag{3}$$

Now if we define the Jacobian matrix as:

$$\mathbf{J} = \begin{bmatrix} \frac{\partial f_1}{\partial x_1} & \Lambda & \frac{\partial f_1}{\partial x_n} \\ \mathrm{M} & & \mathrm{M} \\ \frac{\partial f_m}{\partial x_1} & \Lambda & \frac{\partial f_m}{\partial x_n} \end{bmatrix} \tag{4}$$

then the gradient vector can be written as:

$$g(X) = 2\mathbf{J}^T(X)\mathbf{f}(X) \tag{5}$$

Now, if we differentiate equation (3) with respect to x_k gives the kj-element of the Hessian matrix:

$$G_{kj} = 2\sum_{i=1}^{m}\left\{\frac{\partial f_i}{\partial x_k}\frac{\partial f_i}{\partial x_j} + f_i \frac{\partial^2 f_i}{\partial x_k \partial x_j}\right\} \tag{6}$$

If we denote $\mathbf{T}_i$ be the Hessian matrix of function f_i

$$\mathbf{T}_i(X) = \nabla^2 f_i(X) \tag{7}$$

Then the complete Hessian matrix can be written as:

$$\mathbf{G}(X) = 2\mathbf{J}^T(X)\mathbf{J}(X) + \mathbf{S}(X) \tag{8}$$

where:

$$\mathbf{S}(X) = \sum_{i=1}^{m} f_i(X)\mathbf{T}_i(X) \tag{9}$$

This concludes our discussion about the basics of formation of various matrices in the case of least squares problems, needed for the further developments.

2.2.1.1 Steepest Descent

Now, to find minimum of a function $F(X)$ iteratively, the function should decrease in value from iteration to iteration, in other words,

$$F(X_{k+1}) < F(X_k) \tag{10}$$

Now we need to choose a direction $\mathbf{p}_k$, so that we can move "downhill". Let us consider the first-order Taylor series expansion about X_k:

$$F(X_{k+1}) = F(X_k + \Delta X_k) \approx F(X_k) + g_k^T \Delta X_k \tag{11}$$

where g_k is the gradient evaluated at the old guess X_k:

$$g_k = \nabla F(X)|_{X=X_k} \tag{12}$$

It is evident that the steepest descent would occur if we choose:

$$\mathbf{p}_k = -g_k \tag{13}$$

The original RTRL algorithm uses this direction for minimizing the objective function.

2.2.1.2 Newton's method

The steepest descent algorithm considers first-order Taylor's series expansion. The next method we will discuss is based on the second-order Taylor's series expansion:

$$F(X_{k+1}) = F(X_k + \Delta X_k) \approx F(X_k) + g_k^T \Delta X_k + \frac{1}{2} \Delta X_k^T \mathbf{G}_k \Delta X_k \qquad (14)$$

Taking gradient of this quadratic function with respect to ΔX_k and set it equal to zero for X_{k+1} to be minimum we get:

$$g_k + \mathbf{G}_k \Delta X_k = 0 \Rightarrow \Delta X_k = -\mathbf{G}_k^{-1} g_k = \mathbf{p}_k \qquad (15)$$

if we use equations (5) and (8) to find out the gradient g_k, and Hessian $\mathbf{G}_k$ at the old guess X_k, and substitute in in equation (15) we get,

$$\mathbf{p}_k = -(\mathbf{J}_k^T \mathbf{J}_k + \mathbf{S}_k)^{-1} \mathbf{J}_k^T \mathbf{f}_k \qquad (16)$$

This is the Newton's method for the specialized, non-linear least squares objective function. The Newton's method involves computation of the $\mathbf{S}_k$ term which involves evaluation of $\frac{1}{2} mn(n+1)$ terms for an objective function comprising of sum of squares of 'm' functions and in 'n' dimensional space.

2.2.1.3 Gauss-Newton's method

By neglecting the $\mathbf{S}(k)$ term in the Newton's equation (16) becomes,

$$\mathbf{p}_k = -(\mathbf{J}_k^T \mathbf{J}_k)^{-1} \mathbf{J}_k^T \mathbf{f}_k \qquad (17)$$

This equation defines the Gauss-Newton method.

It is clear from (8), (9) & (16) that if $\mathbf{f}(X) \to 0$ as $X \to X^*$, then also $\mathbf{S}(X) \to 0$, and then the Gauss-Newton tends to Newton's method as the minimum is approached, which will have favorable consequences, such as quadratic convergence.

It has been shown by (Meyer '70) that the convergence constant K can be written as:

$$K \le \left\| \mathbf{S}(X^*) \right\| \left\| \left[\mathbf{J}^T (X^*) \mathbf{J}(X^*) \right]^{-1} \right\| \qquad (18)$$

So, the convergence is superlinear as $\|\mathbf{S}_k\| \to 0$ as $k \to \infty$, otherwise it is first-order and slower the larger $\|\mathbf{S}_k\|$ is. Thus, it can be concluded that Gauss-Newton's approximation to Newton's method would not perform as expected (i.e. with super-linear convergence) if the size of the $\mathbf{S}_k$ term $|f_i(X)|\|\mathbf{T}_i(X)\|$ is larger than eigen values of $\mathbf{J}^T\mathbf{J}$. Also at X^*, the vector $\mathbf{J}^T\mathbf{f}$, being proportional to the gradient vector, must be zero. Therefore if $\mathbf{f}(X^*) \neq 0$, then $\mathbf{J}(X^*)$ is rank deficient and $\mathbf{J}^T(X^*)\mathbf{J}(X^*)$ is singular, so Gauss-Newton cannot be expected to perform satisfactorily in large residual problems, where the residuals of the errors are going to be comparatively larger.

The original RTRL algorithm uses a steepest descent of gradient for minimization, which has linear rate of convergence. Our approach is to use the Gauss-Newton method for the minimization. As the objective function to be minimized is a sum of squares error function, and

we assume that the minimization method is already inside the close neighborhood of a local minimum, where the residuals will be small, and can be considered as a small residual problem. On the other hand if we are not close the local minimum, Gauss-Newton will be extremely slow and might produce inaccurate results in some cases.

2.2.2 Example Illustrating Working of Minimization Schemes

Let us take an example of minimization of a function $F(X) = \sin^2(X_1) + \sin^2(X_2)$ from an initial guess of $[0.5\ 0.5]^T$. Figure (4) illustrate surface of the function.

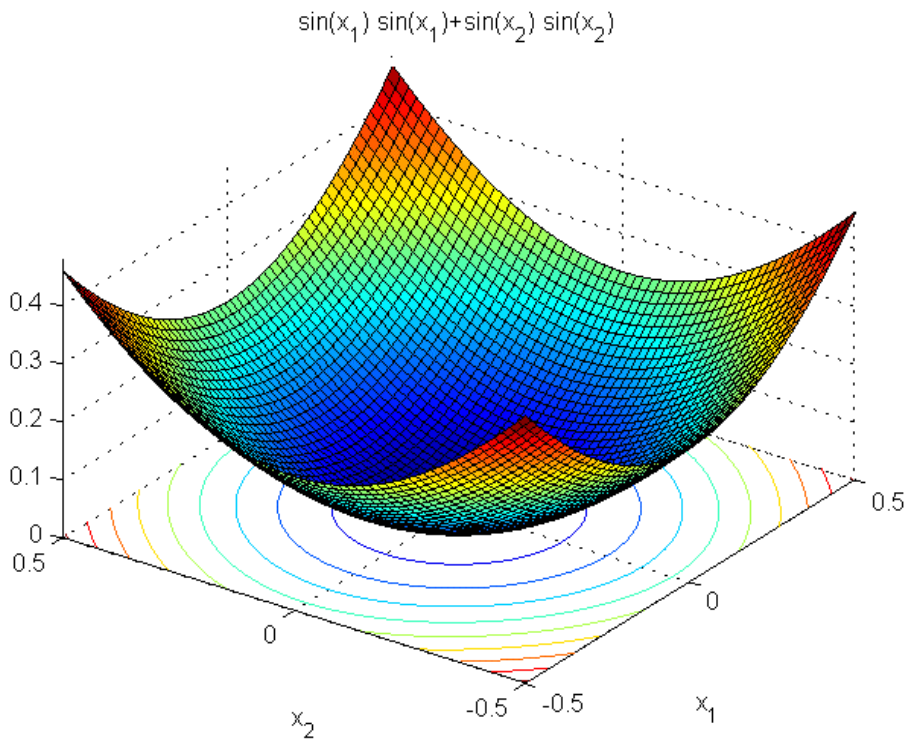


Figure 4: surface of the function to be minimized

2.2.2.1 Forming the Direction Vector

From equation (2), we can write: $\mathbf{f}(X) = [\sin X_1 \quad \sin X_2]^T$

The Jacobian can be written as: $\mathbf{J} = \begin{bmatrix} \cos X_1 & 0 \\ 0 & \cos X_2 \end{bmatrix}$

Now from equation (5), we can write the gradient as: $g(X) = 2\mathbf{J}^T(X)\mathbf{f}(X) = 2\begin{bmatrix} \sin X_1 \cos X_1 \\ \sin X_2 \cos X_2 \end{bmatrix}$

The Hessian is $\mathbf{G}(X) = 2\mathbf{J}^T(X)\mathbf{J}(X) + 2\mathbf{S}(X)$

The $\mathbf{S}(X)$ can be written as:

$$\mathbf{S}(X) = \sum_{i=1}^{m} f_i(X)\mathbf{T}_i(X) = \sin X_1 \begin{bmatrix} -\sin X_1 & 0 \\ 0 & 0 \end{bmatrix} + \sin X_2 \begin{bmatrix} 0 & 0 \\ 0 & -\sin X_2 \end{bmatrix} = \begin{bmatrix} -\sin^2 X_1 & 0 \\ 0 & -\sin^2 X_2 \end{bmatrix}$$

so the Hessian becomes:

$$\mathbf{G}(X) = 2\begin{bmatrix} \cos^2 X_1 & 0 \\ 0 & \cos^2 X_2 \end{bmatrix} + 2\begin{bmatrix} -\sin^2 X_1 & 0 \\ 0 & -\sin^2 X_2 \end{bmatrix} = 2\begin{bmatrix} \cos^2 X_1 - \sin^2 X_1 & 0 \\ 0 & \cos^2 X_2 - \sin^2 X_2 \end{bmatrix}$$

Now,

From a initial guess $X_0 = [0.5\ 0.5]^T$, we get the direction for gradient descent as,

$$\mathbf{p}^{GD} = \begin{bmatrix} 0.8415 & 0.8415 \end{bmatrix}^T$$

The Newton's direction is,

$$-\mathbf{G}^{-1}(X_0)g(X_0) = \begin{bmatrix} 0.7787 & 0.7787 \end{bmatrix}^T$$

and the Gauss-Newton's direction is,

$$-[\mathbf{J}^T(X_0)\mathbf{J}(X_0)]^{-1}\mathbf{J}(X_0)\mathbf{f}(X_0) = \begin{bmatrix} 0.5463 & 0.5463 \end{bmatrix}^T$$

These Directions are shown is figure (2).

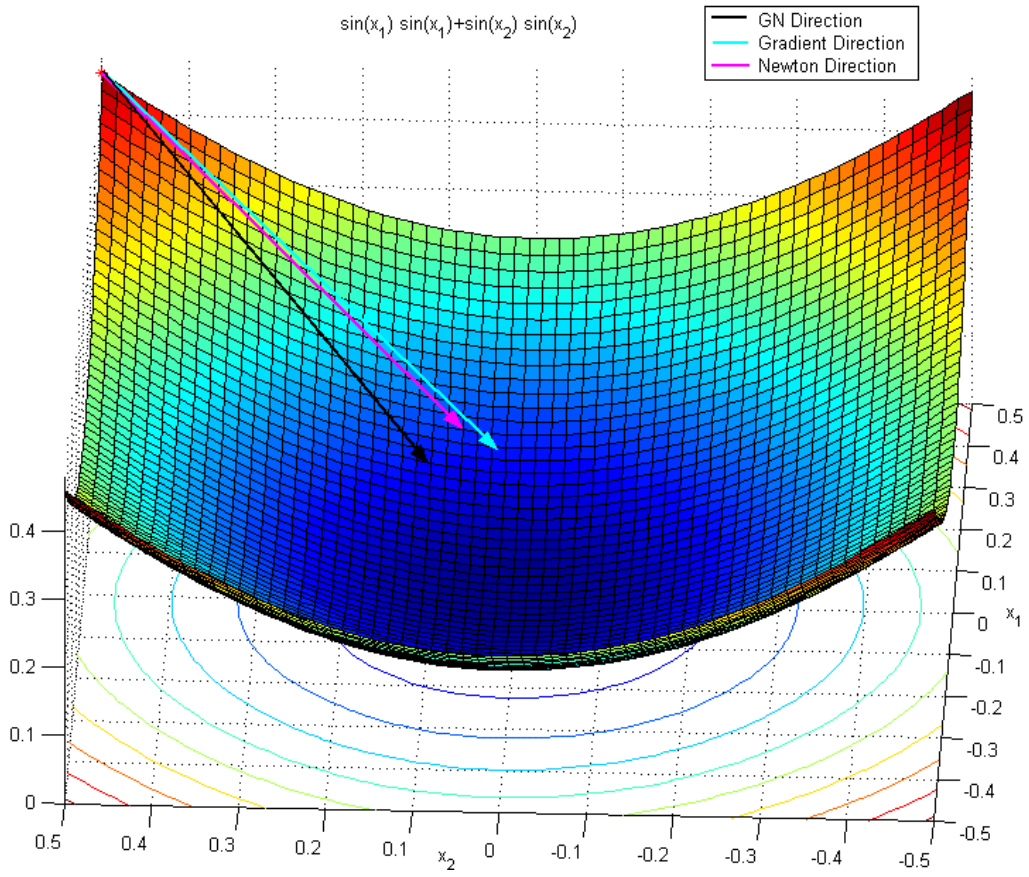


Figure 5: Direction vectors for different methods of minimization of a sum of squares function

2.2.2.2 Line Search

Once we find a minimizing direction the next step is to find a learning rate (step length along the search direction) that minimizes the function along the found direction. This process is called line search and is nothing else but a uni-dimensional minimization procedure.

There are several methods to perform a line search, the simplest being the the bisection method. In this method we start with a bracket of [a,b] = [0,1], and evaluate the function in the found direction in this bracket-end points. The function is also evaluated at the mid point c = (a+b)/2. Now the point with the maximum function value is excluded and a new bracket is formed. This iterative process is continued until the minimum within acceptable range is found.

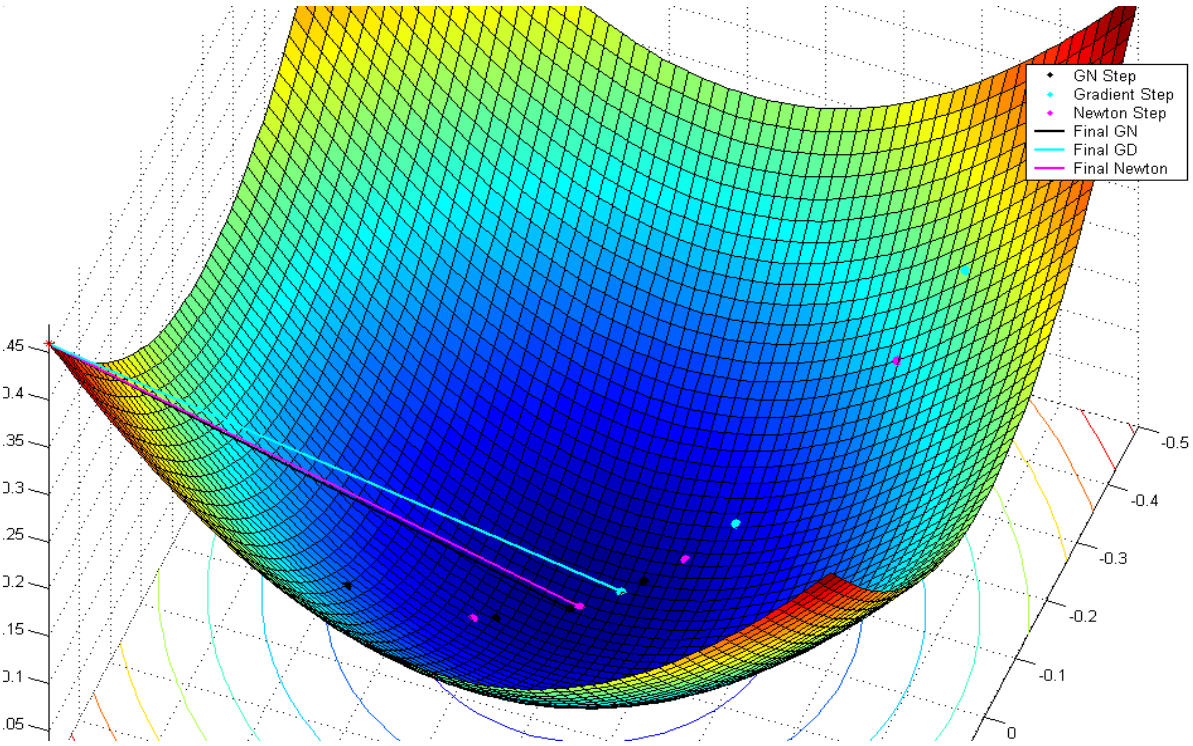


Figure 6: state of the solution after 4 backtracking steps.

The bisection performed on the 3 directions found yielded following results:

Table 1: Results of the line search on the found directions

Steps	Gradient method	Newton method	GN method
1	$F(X_0) = 0.2243$	$F(X_0) = 0.1514$	$F(X_0) = 0.0043$
2	$F(X_1) = 0.0125$	$F(X_1) = 0.0244$	$F(X_1) = 0.1012$
3	$F(X_2) = 0.0342$	$F(X_2) = 0.0141$	$F(X_2) = 0.0163$
4	**$F(X_3) = 0.0013$**	**$F(X_3) = 0.000354$**	**$F(X_3) = 0.000966$**

It can be seen from table (1) that for equal number of minimization steps the quality of the solution produced by the Gauss-Newton method is superior to the one provided by the steepest gradient method. On the other hand, when comparing the Gauss-Newton and Newton's methods, both solutions are comparable in accuracy. We have saved the computation of $\frac{1}{2}mn(n+1) = 6$ terms, by omitting the $\mathbf{S}(X)$ term from the Gauss-Newton calculation.

2.3 Off-Line RTRL Algorithm

In this section we present an algorithm for off-line training of FRNNs. The algorithm, which appears in (Williams & Zipser '89), is based on the Gradient Descent method, when the total *Sum of Squared Errors* (*SSE*) is being used as the objective function to be minimized.

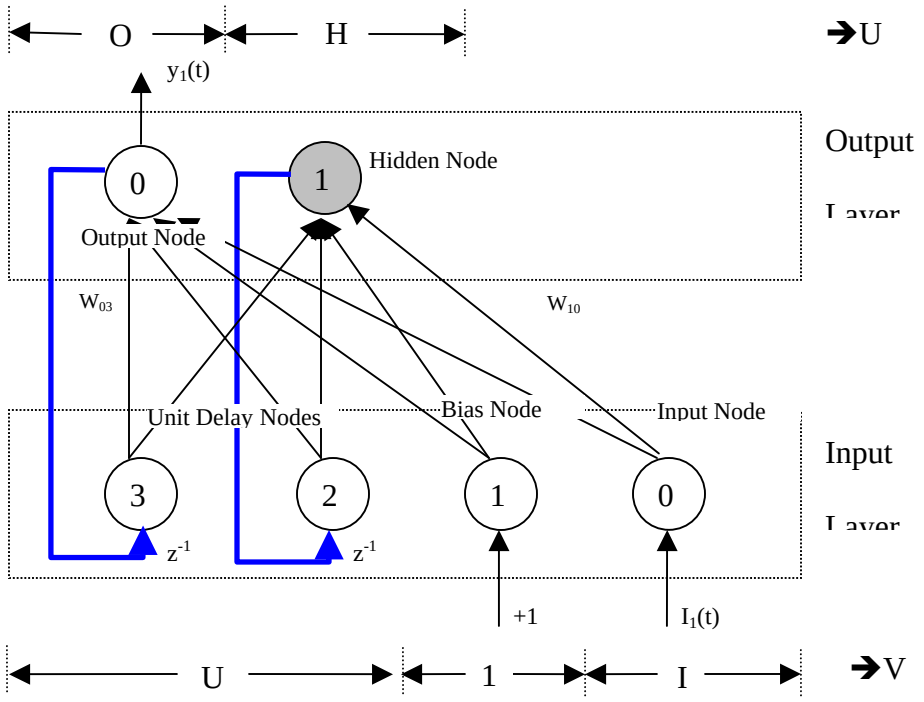


Figure 7: Fully recurrent neural network architecture

The above figure shows the notation used for ease of understanding the derivations ahead. The output layer consists of U nodes, L are the observable nodes and H are the hidden nodes. The input layer has a total of V nodes, comprising of I input nodes, 1 bias node and U unit-delay nodes. The indexing as shown inside the nodes is followed for ease of writing the equations. The weights are denoted as $W_{(\text{'to' node index})(\text{'from' node index})}$. Note that the recurrent connections do not have any weights associated with them and they just feed back the current output to the delay element. The outputs of the output nodes have some initial value at time t=-1, which is denoted by $y_k(-1)$ for k=0…U-1

The operations at a node in the output layer are depicted in the following figure:

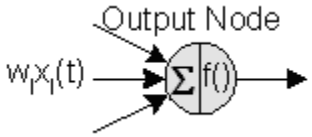


Figure 8: Operations at the output node

All the weighted inputs converging to the node are summed together and the output of the node is the output of a nonlinear function applied to that sum. Typical choices for this nonlinear function include the logistic and hyperbolic tangent functions, as shown below:

$$f_{\tanh}(s) = \tanh\left(\frac{s}{2}\right) = \frac{1-e^{-x}}{1+e^{-x}} \tag{19}$$

$$f_{\log}(s) = \frac{1}{1+e^{-s}} = \frac{1+\tanh\left(\frac{s}{2}\right)}{2} \tag{20}$$

In the performance phase of the network, let T be the number of test patterns with the first pattern being presented at time t=0. Let $y_k(t) \quad k = 0..O-1$ be the observed outputs and $y_k(t) \quad k = O..U-1$ be the H unobserved outputs of the context units.

The outputs of the output units can be written as:

$$y_k(t) = f\left(s_k(t)\right) \tag{21}$$

where:

$$s_k(t) = \sum_{l=0}^{I-1} w_{k,l} x_l(t) + w_{k,I} + \sum_{l=0}^{U-1} w_{k,l+I+1} y_l(t-1) \quad \text{t=0… T-1, k=0…U-1} \tag{22}$$

Now, learning in the RNN architecture can be performed by minimizing a suitable objective function $\varphi(\ )$. Furthermore, the minimization can be achieved by finding appropriate values for all the free parameters of the network. While up to this point we have considered only the weight matrix **W** as a network parameter, the *U* initial values $y_k(-1) \quad k = 0..U-1$ are arbitrary and, therefore, we can consider them too as network parameters that can be tuned during the training process. These values can be summarized in the output state vector $\mathbf{y}(-1)$. Moreover, both the output state vector and the weight matrix can be summarized in a single column vector of $U*(U+I+2)=(O+H)*(O+I+H+2)$ elements

$$\boldsymbol{\theta} = [\theta_r]_{r=0..U(V+1)-1} = \begin{bmatrix} vec[\mathbf{W}] \\ \mathbf{y}(-1) \end{bmatrix} \tag{23}$$

here *vec*(.) is used to indicate the weight matrix **W** is arranged in a single column vector. The relationship between θ_r and $w_{i,j}$, y_k(-1) parameters is given below:

$$\begin{aligned} & w_{i,j}\big|_{i=0..U-1,\, j=0..V-1} = \theta_r \\ & y_k(-1)\big|_{k=0..U-1} = \theta_r \end{aligned} \quad \text{where} \quad \begin{aligned} & r = Vi + j \\ & r = k + UV \end{aligned}$$

$$\begin{aligned} & \theta_r\big|_{r=0..UV-1} = w_{i,j} \\ & \theta_r\big|_{r=UV..U(V+1)-1} = y_k(-1) \end{aligned} \quad \text{where} \quad \begin{aligned} & i = \lfloor r/V \rfloor,\ j = r - Vi \\ & k = r - UV \end{aligned} \tag{24}$$

To understand the above mapping let us better consider an example, where we have a network

with 3 input nodes (I = 3), 4 hidden nodes (H = 4) and 2 output nodes (O = 2). The weight matrix will have dimensionality 6x10 and y(-1) will have dimensionality 1x6.

Now the θ vector can be written as follows:

$$\begin{array}{l}\theta = [w_{00}\, w_{01}\, \Lambda\ w_{09}\, w_{10}\, \Lambda\ w_{59}\, y_0(-1)\, y_1(-1)\, \Lambda\ y_5(-1)]^T_{1\times 66} \\ \quad\downarrow\ \downarrow\quad\ \downarrow\ \downarrow\quad\ \downarrow\ \downarrow\quad\ \downarrow\quad\ \downarrow \\ \quad\theta_0\ \ \theta_1\, \Lambda\ \ \theta_9\ \ \theta_{10}\, \Lambda\ \theta_{59}\quad \theta_{60}\quad \theta_{61}\quad\ \ \theta_{65}\end{array}$$

2.3.1 Finding the Direction Vector

Continuing with our presentation, our objective function (Sum of Squared Errors (SSE)), which for the RNN is defined as

$$SSE = \phi(\boldsymbol{\theta}) = \sum_{t=0}^{T-1} \phi(\boldsymbol{\theta}, t) \tag{25}$$

where we define the instantaneous SSE $\varphi(\boldsymbol{\theta},t)$ as

$$\phi(\boldsymbol{\theta}, t) = \frac{1}{2} \sum_{k=0}^{U-1} e_k^2(t) \tag{26}$$

From Equation (26) we can see that the SSE is proportional to the average of instantaneous SSE over the time period *t=0..T-1*. The instantaneous error $e_k(t)$ for output neuron *k* depends

implicitly on $\boldsymbol{\theta}$ and is given as

$$e_k(t) = \begin{cases} d_k(t) - y_k(t) & \text{if } \exists\, d_k(t) \quad k = 0..O-1 \\ 0 & \text{otherwise} \end{cases} \tag{27}$$

In other words, $e_k(t)$ is zero, if there is no specific desired response $d_k(t)$ for output k at time instance t. We can combine Equations (25) and (26) into

$$SSE = \phi(\boldsymbol{\theta}) = \frac{1}{2}\sum_{t=0}^{T-1}\sum_{k=0}^{U-1} e_k^2(t) \tag{28}$$

In this section we are going to apply Gradient Descent to minimize the SSE. This is the off-line learning procedure, since we consider errors over a period of all the time instances simultaneously. The gradient is given as

$$\nabla_{\boldsymbol{\theta}}\phi = \left[\frac{\partial\phi}{\partial\theta_r}\right]_{r=0..U(V+1)-1} = \begin{bmatrix} \nabla_{\mathbf{W}}\phi \\ \nabla_{\mathbf{y}(-1)}\phi \end{bmatrix} \tag{29}$$

where we define the following column vectors

$$\nabla_{\mathbf{W}}\phi = vec\left[\frac{\partial\phi}{\partial w_{i,j}}\right]_{i=0..U-1,\ j=0..U+I} \tag{30}$$

$$\nabla_{\mathbf{y}(-1)}\phi = vec\left[\frac{\partial\phi}{\partial y_i(-1)}\right]_{i=0..U-1} \tag{31}$$

In order to specify the gradient completely we need to find expressions for the partial derivatives in Equations (30) and (31). From Equation (26) the partial derivative with respect to the parameter θ_r (which might be a specific $w_{i,j}$ or a specific $y_i(-1)$) is given as

$$\frac{\partial \phi}{\partial \theta_r} = \sum_{t=0}^{T-1} \sum_{k=0}^{U-1} e_k(t) \frac{\partial e_k(t)}{\partial \theta_r} \qquad r = 0..U(V+1)-1 \tag{32}$$

Utilizing Equation (27), Equation (32) can be rewritten as

$$\frac{\partial \phi}{\partial \theta_r} = -\sum_{t=0}^{T-1} \sum_{k=0}^{U-1} e_k(t) \frac{\partial y_k(t)}{\partial \theta_r} \qquad r = 0..U(V+1)-1 \tag{33}$$

In order to proceed further we need the expressions for the partial derivatives inside the summations. These can be calculated via Equation (21).

$$\frac{\partial y_k(t)}{\partial \theta_r} = f'\left(s_k(t)\right) \frac{\partial s_k(t)}{\partial \theta_r} \qquad r = 0..U(V+1)-1 \tag{34}$$

We can express the derivative of the activation function in Equation (19) as

$$f'_{\tanh}(s_k(t)) = 1 - \left(y_k(t)\right)^2 \tag{35}$$

$$f'_{\log}(s_k(t)) = y_k(t)\left[1 - y_k(t)\right] \tag{36}$$

The form of the derivative in Equations (35) and (36) is convenient from an implementation perspective, since they require only a few floating-point operations. At this point we still need to find expressions for the partial derivatives in Equation (34). For clarity, we need to differentiate now between θ_r being a specific $w_{i,j}$ or a specific $y_i(-1)$. Thus, Equation (34) becomes

$$\frac{\partial y_k(t)}{\partial w_{i,j}} = f'\left(s_k(t)\right) \frac{\partial s_k(t)}{\partial w_{i,j}} \qquad k = 0..U-1, i = 0..U-1, j = 0..V-1 \tag{37}$$

$$\frac{\partial y_k(t)}{\partial y_i(-1)} = f'\left(s_k(t)\right) \frac{\partial s_k(t)}{\partial y_i(-1)} \qquad k = 0..U-1, i = 0..U-1 \tag{38}$$

The above partial derivatives can be calculated with the help of Equation (22), so that Equations (37) and (38) become

$$\frac{\partial y_k(t)}{\partial w_{i,j}} = f'(s_k(t))\left[\sum_{l=0}^{U-1} w_{k,l+I+1}\frac{\partial y_l(t-1)}{\partial w_{i,j}} + \begin{cases} x_j(t) & \text{if } j = 0..I\text{-}1 \\ 1 & \text{if } j = I \\ y_{j-I-1} & \text{if } j = I+1..V-1 \end{cases}\right] \quad k = 0..U-1, i = 0..U-1, j = 0..V-1 \tag{39}$$

$$\frac{\partial y_k(t)}{\partial y_i(-1)} = f'(s_k(t))\left[\sum_{l=0}^{U-1} w_{k,l+I+1}\frac{\partial y_l(t-1)}{\partial y_i(-1)}\right] \qquad k = 0..U-1, i = 0..U-1 \tag{40}$$

If we proceed to define the training state quantities

$$p_{i,j}^k(t) = \frac{\partial y_k(t)}{\partial w_{i,j}} \qquad k = 0..U-1, i = 0..U-1, j = 0..U+I \tag{41}$$

$$q_i^k(t) = \frac{\partial y_k(t)}{\partial y_i(-1)} \qquad k = 0..U-1, i = 0..U-1 \tag{42}$$

then we can rewrite Equations (41) and (34) as

$$p_{i,j}^k(t) = f'(s_k(t))\left[\sum_{l=0}^{U-1} w_{k,l+I+1}p_{i,j}^k(t-1) + \delta_{k,i}\begin{cases} x_j(t) & \text{if } j = 0..I\text{-}1 \\ 1 & \text{if } j = I \\ y_{j-I-1} & \text{if } j = I+1..V-1 \end{cases}\right] \quad k = 0..U-1, i = 0..U-1, j = 0..V-1 \tag{43}$$

$$q_i^k(t) = f'(s_k(t))\left[\sum_{l=0}^{U-1} w_{k,l+I+1}q_i^k(t-1)\right] \qquad k = 0..U-1, i = 0..U-1 \tag{44}$$

where $\delta_{k,i}$ is the Kronecker delta symbol. Due to their definition in Equations (41) and (42), the training state quantities are initialized to

$$p_{i,j}^k(-1) = \frac{\partial y_k(-1)}{\partial w_{i,j}} = 0 \qquad k = 0..U-1, i = 0..U-1, j = 0..U+I \tag{45}$$

$$q_i^k(-1) = \frac{\partial y_k(-1)}{\partial y_i(-1)} = \delta_{k,i} \qquad k = 0..U-1, i = 0..U-1 \tag{46}$$

By introducing the training state quantities the calculation of the gradient components in

Equation (33) is performed as follows

$$\frac{\partial \phi}{\partial w_{i,j}} = -\sum_{t=0}^{T-1}\sum_{k=0}^{U-1} e_k(t)\, p_{i,j}^{k}(\mathrm{t}) \qquad i = 0..U-1,\ j = 0..V-1 \tag{47}$$

$$\frac{\partial \phi}{\partial y_k(-1)} = -\sum_{t=0}^{T-1}\sum_{k=0}^{U-1} e_k(t) q_i^{k}(\mathrm{t}) \qquad k = 0..U-1 \tag{48}$$

Up to this point we managed to calculate the gradient vector. Training using the Gradient Descent method produces parameter updates that are of the form

$$\begin{aligned} \Delta \mathbf{W} &= -\lambda \nabla_{\mathbf{W}} \phi \\ \Delta \mathbf{y}(-1) &= -\lambda \nabla_{\mathbf{y}(-1)} \phi \end{aligned} \tag{49}$$

where λ>0 is the learning rate.

The update equation thus can be written as:

$$\mathbf{W}^{new} = \mathbf{W}^{old} + \lambda\, \mathbf{p} \tag{50}$$

Where **p** is an appropriate direction vector and λ is the step length in the direction of **p** (also referred to as the learning rate). Our direction vector here is $\mathbf{p} = \mathbf{p}^{GD} = -\nabla_{\mathbf{W}} \phi$, so

$$\mathbf{W}^{new} = \mathbf{W}^{old} - \lambda \nabla_{\mathbf{W}\phi} \tag{51}$$

Before training commences it is important to normalize (adjust the range of values) the training patterns.

2.3.2 Line Searches

After finding the direction vector we need to find the learning rate adaptively, as a constant

learning rate would not necessarily minimize the sum of the squared errors. So for the given direction vector we find the learning rate. The problem of finding the learning rate for minimizing the SSE becomes a one-dimensional minimization problem, and this problem can be solved by several methods.

2.3.2.1 Bisection

One of the simplest methods is the backtracking. In backtracking we start with a bracket of learning rates, such as [0, maximum learning rate]. The function (to be minimized) is evaluated at the bracket mid point, and the point with the maximum value of function is eliminated, and this process is continued until we reach to the minimum of the function with the 2 points separated with the distance of machine's floating point precision. The bracket of learning rates, for our case, would be always bounded by [0, maximum learning rate], so our initial guess for the bracketing minimum would always be this bracket.

2.3.2.2 Parabolic Interpolation (Brent's Method)

Assuming that our function is parabolic near the minimum, a parabola fitted through any three points would take us to the minimum in a single step, or at least very near to it.

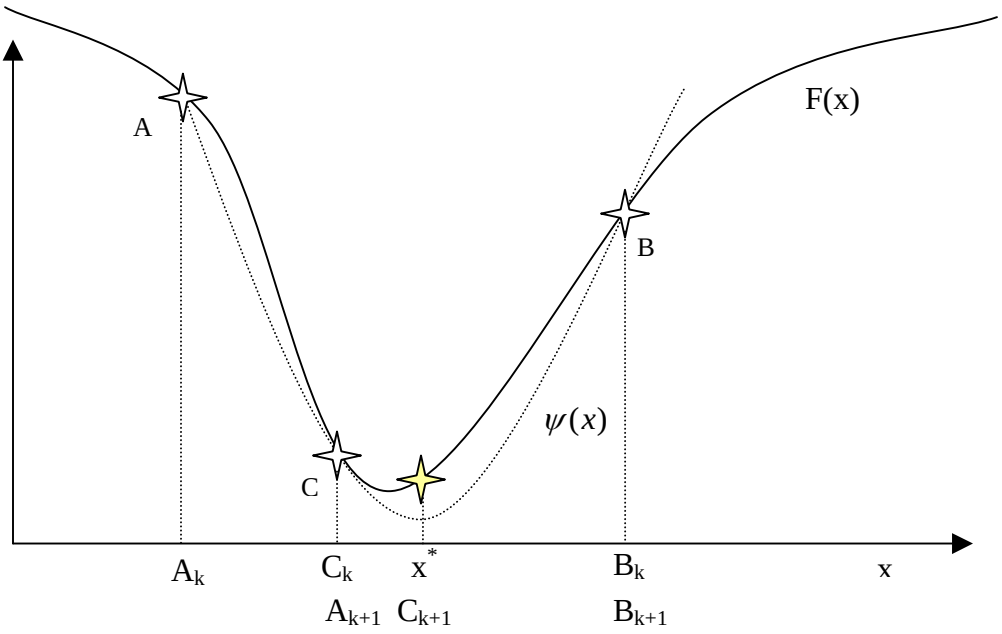


Figure 8: Parabolic interpolation (Brent's method)

If we denote equation of the parabola by:

$$\psi(x) = ax^2 + bx + c\,, \tag{52}$$

to fit a parabola through out function, we can use the bracket end points to find out the coefficients a, b and c. In fact we do not need the complete form of $\psi(x)$, as only the position of the minimum of the parabola x^* is required. Differentiating equation (52) with respect to x, and equating it to 0, we get,

$$\psi'(x^*) = 2ax + b = 0 \Rightarrow x^* = -\frac{b}{2a} \tag{53}$$

so ratio of b/a is needed. Now we have the bracket as [A,B] and another internal point C such that $\psi(C) < \psi(A)$ and $\psi(C) < \psi(B)$.

We solve three simultaneous linear equations:

$$\begin{bmatrix} A^2 & A & 1 \\ B^2 & B & 1 \\ C^2 & C & 1 \end{bmatrix} \begin{bmatrix} a \\ b \\ c \end{bmatrix} = \begin{bmatrix} \psi(A) \\ \psi(B) \\ \psi(C) \end{bmatrix} \tag{54}$$

The x^* is then obtained as:

$$x^* = \frac{1}{2} \frac{(B^2 - C^2)\psi(A) + (C^2 - A^2)\psi(B) + (A^2 - B^2)\psi(C)}{(B - C)\psi(A) + (C - A)\psi(B) + (A - B)\psi(C)} \tag{55}$$

As shown in figure (8), A_{k+1} becomes C_k, C_{k+1} becomes the new found point x^* for the next iteration k+1.

2.4 Off-Line GN-RTRL Algorithm

In this section we present our new version of the RTRL algorithm by using the Gauss Newton's technique for minimizing the least squares objective function. The Gauss Newton method replaces the direction method $\mathbf{p}_k$ in the original approach described above with the Gauss-Newton direction. The learning rate is adaptively calculated the same way using the Brent method (based on parabolic interpolation technique) as described in the last section. The function to be minimized is the function in equation (20), with respect to the adaptable parameters (i.e. the weights). Let us first define $\mathbf{r}(t) = \left[e_o(t)\, e_1(t) \ldots e_{L-1}(t)\, e_o(t-1) \ldots e_{L-1}(t-T+1)\right]^T$, where r(t) is the vector

containing the errors for different (visible) outputs across T consecutive time instances (the current time instance t and the T-1 previous time instances; so $r(t) \in \Re^{L \cdot T}$). Also if $\boldsymbol{\theta} \hat{=} vec[\mathbf{W}]$, then the ($m$,$n$) entry of the Jacobian matrix can be written as,

$$\mathbf{J}(t;m,n) = \frac{\partial r_m(t)}{\partial \theta_n} \tag{52}$$

The gradient vector of $\phi(\boldsymbol{\theta},t) = \frac{1}{2}\sum_{t=0}^{T-1}\sum_{k=0}^{U-1} e_k^2(t)$ is then defined as,

$$\nabla\phi(\boldsymbol{\theta},t) = \mathbf{J}^T(t)r(t) \tag{53}$$

and the Hessian of $\phi(\boldsymbol{\theta},t)$ as,

$$\nabla^2\phi(\boldsymbol{\theta},t) = \mathbf{J}(t)^T\mathbf{J}(t) + \sum_{j=0}^{t-T+1} r_j(t)\nabla^2 r_j(t) \tag{54}$$

Note here that after calculating the gradient g of the objective function, we get the first part of equation (54) without any further evaluations. The Gauss-Newton method assumes that the minimization problem at hand is of small residual and, therefore, ignores the second part of the Hessian calculation.

Let g_k denote the gradient vector of $\phi(\boldsymbol{\theta},t)$, and G_k denote the Hessian of $\phi(\boldsymbol{\theta},t)$; G is defined in the following equation

$$g_k = \nabla\phi(\theta,t) \mid \theta = \theta_k \tag{55}$$

$$G_k = \nabla^2\phi(\theta,t) \mid \theta = \theta_k \tag{56}$$

where $\nabla\phi(\theta,t)$ and $\nabla^2\phi(\theta,t)$ are defined in equations (53) & (54).

Now, a Taylor series gives the expansion of the gradient of a quadratic function at

$$\theta_{k+1} = \theta_k + \mathbf{p}_k \text{ as } g_{k+1} = g_k + \mathbf{G}\,\mathbf{p}_k . \quad (57)$$

If we take a the gradient of this quadratic function with respect to p_k and set it to zero we get,

$$g_k + \mathbf{G}\mathbf{p}_k = 0 \text{ or } \mathbf{p}_k = -\mathbf{G}^{-1} g_k \quad (58).$$

Now from (53), (54) & (58),

$$p_k = -\left[\mathbf{J}^T(t)\mathbf{J}(t) + \mathbf{S}(t)\right]^{-1} \mathbf{J}^T(t)\mathbf{r}(t) \quad (59)$$

The main problem in applying this method is the computation of the **S**(t) term. This calculation involves evaluation of $\frac{1}{2}mn(n+1)$ terms. This difficulty leads us to the Gauss-Newton modification of the Newton's method. If we ignore the **S**(t) term in equation (52) we get the Gauss-Newton equation for the optimal direction vector, defined as,

$$\mathbf{p} = \mathbf{p}^{GN} = -\left[\mathbf{J}^T(t)\mathbf{J}(t)\right]^{-1} \mathbf{J}^T(t)\mathbf{r}(t) \quad (60)$$

This approximation in equation (60) can be viewed as a low computational cost approximation to the Hessian matrix associated to the SSE minimization, which is sufficiently accurate for small-residual problems. (Scales '85).

In out case, $SSE = \phi(\boldsymbol{\theta}, t) = \frac{1}{2}\sum_{t=0}^{T-1}\sum_{k=0}^{U-1} e_k^2(t)$, is to be minimized with respect to the adaptable weight vector $\boldsymbol{\theta}$.

The residual vector $r(t)$ can be written as:

$$\mathbf{r}(t) = \left[(d_0(t) - y_0(t))\,(d_1(t) - y_1(t)) \ldots (d_{L-1}(t) - y_{L-1}(t))\,(d_o(t-1) - y_0(t-1)) \ldots (d_{L-1}(t-T+1) - y_{L-1}(t-T+1))\right]^T \quad (61)$$

so, $\frac{\partial r(t)}{\partial \theta_r} = -\frac{\partial y_k(t)}{\partial \theta_r} = -p^k_{i.j}(t) = \mathbf{J}(t;m,n)$ where: $t = \lfloor m/L \rfloor, k = m - Lt, i = \lfloor n/V \rfloor, j = n - Vi$ (62)

This equation links the Jacobian matrix to the output sensitivities $p^k_{i,j}(t)$. This equation also underlines the fact that GN-RTRL is closely related to GD-RTRL, since both approaches utilize output sensitivity information for the computation of their corresponding search directions. Nevertheless, GN-RTRL promises shorter training time without the need of computing second order derivatives.

In order to solve the equation to get the Gauss-Newton direction, we avoid the formation of the term $-\left[\mathbf{J}^T(t)\mathbf{J}(t)\right]^{-1}\mathbf{J}^T(t)$ directly as it is prone to considerable, numerical errors during its computation. Instead we solve:

$$\mathbf{J}^T\mathbf{J}p^{GN} = -\mathbf{J}^T r \tag{63}$$

using a numerically stable approach, namely the Singular Value Decomposition (SVD), as presented in (Golub & Van Loan, 1996). We decompose the Jacobian matrix so that,

$$\mathbf{J} = \mathbf{U}\begin{bmatrix} S \\ 0 \end{bmatrix}\mathbf{V}^T = [\mathbf{U}_1\mathbf{U}_2]\begin{bmatrix} S \\ 0 \end{bmatrix}\mathbf{V}^T = \mathbf{U}_1\mathbf{S}\mathbf{V}^T, \tag{64}$$

Where,

U is m x m orthogonal; $\mathbf{U}_1$ contains first n columns of **U**, $\mathbf{U}_2$ the last m-n columns;

V in n x n orthogonal;

S is n x n diagonal, with diagonal elements $\sigma_1 \geq \sigma_2 \geq \Lambda \geq \sigma_n > 0.$ for a m x n Jacobian matrix. The matrices **U**, **V** should not be confused with the sizes of the input and output layers of the FRNN.

Now using this decomposition we can write,

$$\mathbf{p}^{GN} = -\mathbf{V}\mathbf{S}^{-1}\mathbf{U}_1^T\mathbf{r}(t)\,. \tag{65}$$

In terms of the left- ($\mathbf{u}_i$), right-eigenvectors ($\mathbf{v}_i$) and singular values (σ_i) of the Jacobian matrix the direction vector can be written as,

$$\mathbf{p}^{GN} = -\sum_i \frac{\mathbf{u}_i^T\mathbf{r}(t)}{\sigma_i}\mathbf{v}_i \tag{66}$$

In the above summation terms that correspond to relatively small singular values are omitted to increase robustness in the direction vector calculations. This phenomenon may occur when **J**(t) is close to being rank deficient and the unaltered search vector may not represent a descent direction. Another alternative would be using the negative gradient as in GD-RTRL for that particular time step until **J**(t) becomes full rank again.

Some of the advantages in using the Gauss-Newton method for the minimization in the least squares solutions problem can be listed as follows. First, we ignore the second order Hessian term from the Newton's method, which saves us the trouble of computing the individual Hessians. Secondly our minimization task is a *small residual* problem as the residual vector **r**(t) is the error between the desired output and the actual output, and tends to zero as the learning progresses. In the *small residual* problems the first term in equation (49) is much more significant than the second term, and the Gauss-Newton method gives performance similar to the Newton's method, with reduced computational effort (Nocedal & Wright, 1999).

The original RTRL algorithm comes in 2 versions: 1) Batch mode (off-line training) and 2) Continuous mode (on-line learning).

In continuous mode the weights are updated as the network is running. The advantage with this approach is that you don't have to specify any epoch boundaries and that leads to a conceptual and implementation-wise simple algorithm. The disadvantage is that now the algorithm is no longer the exact negative gradient of the total error along the trajectory. In the case of GN-RTRL for the super-linear convergence, the Jacobian matrix must be full column rank. In an effort to fulfill the last requirement, GN-RTRL must compute its direction vector based on $T \geq UV/L$ time instances, which contrasts GD-RTRL that needs only current time instance information. Although this very fact might be considered as a disadvantage of GN-RTRL from a computational or memory-storage perspective, utilizing information of T instead of just one time instance to perform on-line learning may cause a smoothing/averaging effect in time and allow GN-RTRL to converge faster in an on-line mode.

3: EXPERIMENTS

In order to demonstrate the merits of the Gauss-Newton RTRL we have chosen to compare it to the original algorithm (GD-RTRL) on one-step-ahead prediction problems. For both GN- & GD-RTRL we used the same parabolic interpolation technique for approximate line minimization, the same set of initial weights and the same, maximum learning rate. For each dataset the two algorithms performed training using 100 different initial configurations. Once the algorithms converged, they were tested on all available data and the produced SSE was measured. The datasets we considered were:

Santa-Fe Time Series Competition dataset: The dataset consists of a computer-generated, 1-dimensional temporal sequence of 500 time instances. The data set was generated by numerical integration of equation of motion of a damped particle. The data set looks as follows:

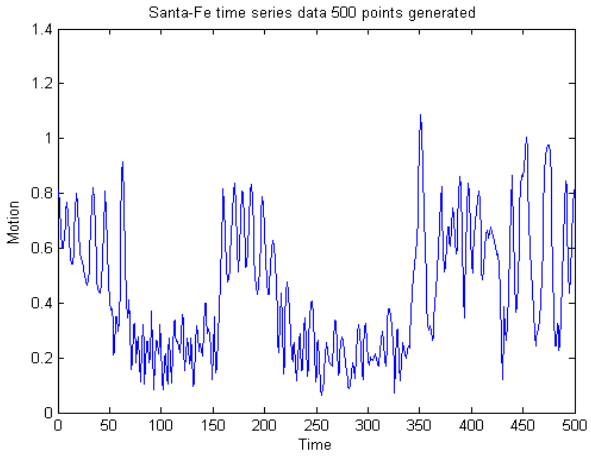


Figure 9: Santa-Fe time series data set

[Link: http://www-psych.stanford.edu/~andreas/Time-Series/SantaFe.html]

Sunspot dataset: The dataset consists of a temporal sequence representing the annual, average number of sunspot activity as measured in the interval 1749 to 1915. The sequence is 1-dimensional and contains 2000 samples. [Link: http://science.msfc.nasa.gov/ssl/pad/solar/greenwch.htm]

The results obtained are summarized in Tables 1, 2 depicted below. TSUC denotes time steps until convergence, while KFlops denotes thousands of floating point operations performed during training. In the online learning case, the condition of convergence is, the L-infinity norm of the gradient vector must be smaller than an accuracy value ε, for that particular time instant.

$$\left|g_k\right|_\infty < \varepsilon \text{ where } \varepsilon > 0 \text{ is the accuracy threshold.} \qquad (67)$$

The experimental results for both datasets verify that GN-RTRL is superior in terms of convergence as expected. In the first dataset, GN-RTRL achieved convergence (on average) in only 7% of the time steps required by GD-RTRL, while in the second one in 35%. In terms of computational effort (Kflops), GN-RTRL seems to be (on the average) better than the GD-RTRL method for the Santa-Fe series (96 vs. 710 Kflops) and comparable for the Sun-spot series (128 vs. 154 KFlops).

Table 2: Performance - Santa-Fe Time Series

	TSUC			KFlops			SSE		
	Min.	Mean	Max.	Min.	Mean	Max.	Min.	Mean	Max.
GD-RTRL	163	**750.43**	1596	153.01	**710.74**	1517.35	10.86	90.58	1016
GN-RTRL	16	**55.92**	472	13.72	**96.95**	941.41	1.86	109.27	1016

Table 3: Performance - Sunspot Time Series

	TSUC			KFlops			SSE		
	Min.	Mean	Max.	Min.	Mean	Max.	Min.	Mean	Max.
GD-RTRL	39	**168.17**	186	36.16	**153.86**	170.37	6.88	35.95	101.44
GN-RTRL	16	**58.48**	1992	19.28	**127.93**	5262.52	1.65	18.08	1013.5

Figure 10 shows boxplots describing the distribution of computational effort required by the 2 algorithms in terms of kiloflops for the computer generated dataset. The boxplot has blue lines at lower and upper quartiles, while the red line indicates the median of the Kflops for each of the methods. The lines extending from the box indicate the rest of the points. These plots show that GN-RTRL is relatively more computationally efficient in comparison to GD-RTRL. Despite the fact that GD-RTRL has small computational overhead per time instance, when compared to GN-RTRL that incorporates an SVD step, the latter method compensates by performing significantly less iterations.

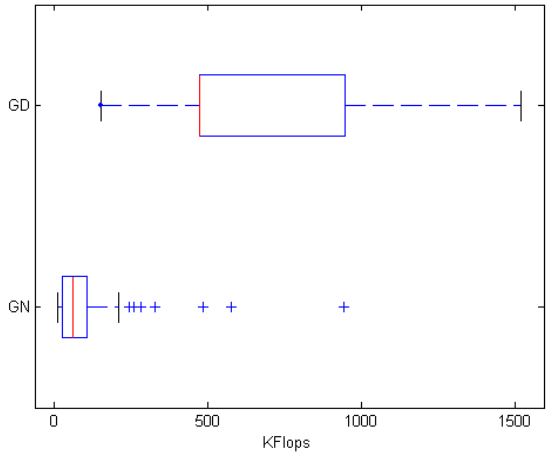


Figure 10: Boxplot of KFlops for GN-RTRL and GD-RTRL for the Santa-Fe Time Series

In terms of solution quality, Tables 1 and 2 clearly emphasize the superiority of GN-RTRL. The SSE of GN-RTRL is by almost an order of magnitude smaller than the one achieved by GD-RTRL. Figure 11 shows a representative plot of SSE computed during the testing phase versus the TSUC for both methods, which demonstrates that GN-RTRL converges much faster to a solution and, simultaneously, the solution is of higher quality, when compared to the GD-RTRL solution. Additionally, Figure 11 indicates cases, where it seems that GN-RTRL may have terminated training rather prematurely, which resulted in high SSE values. This may be attributed to the initialization of the weight matrix, and in these cases the algorithm may have converged to the local minima. To avoid these random initializations which converge to local minima, a method may be devised where at the initial stages of the algorithm the GN direction is watched and if it is observed to be diverging away from the steepest descent, this initialization is

discarded and another random weight matrix is selected.

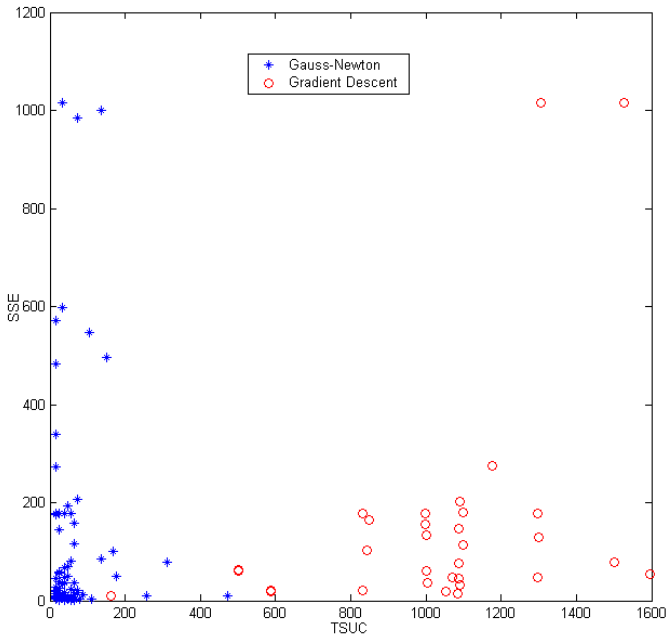


Figure 11: The SSE versus TSUC results of the FRNN trained with the GN-RTRL and GD-RTRL methods

4: SUMMARY, CONCLUSIONS AND FUTURE RESEARCH

We have presented a Gauss-Newton variation of the Real Time Recurrent Learning algorithm (Williams & Zipser, ’89) for the on-line training of Fully Recurrent Neural Networks. The modified algorithm, GN-RTRL, performs error minimization using Gauss-Newton direction vectors that are computed from information collected over a period of time rather than only using instantaneous gradient information. GN-RTRL is a robust and effective compromise between the original, gradient-based RTRL (low computational complexity, slow convergence) and Newton-based variants of RTRL (high computational complexity, fast convergence). Experimental results were reported that reflect the superiority of GN-RTRL over the original version in terms of speed of convergence and solution quality. Furthermore, the results indicate that, in practice, GN-RTRL features a lower-than-expected computational cost due to its fast convergence: GN-RTRL required fewer computations than the original RTRL to accomplish its learning task. This is indicated by the results obtained on the tests performed on the sunspot and santa-fe time series data sets.

The formation of the Gauss-Newton direction requires computation of singular value decomposition of the Jacobian matrix, in order to circumvent the problem of inverting that matrix. The majority of computational load is due to this computation of the SVD. Now if the Jacobian matrix is rank deficient the direction vector may not point to the descent direction, and in that case we have to take the gradient descent direction. This is one of the

shortcomings of the method that the quality of the GN direction depends upon the rank of the Jacobian matrix.

LIST OF REFERENCES

[Werbos ‘90] – Werbos, “Backpropagation through time: What it does and how to do it”, *Proceedings of the IEEE*, 78(10), 1990.

[Williams & Zipser ‘89] – Williams R.J., Zipser D., “A learning algorithm for continually running fully recurrent learning algorithm,” *Neural Computation*, 1, 270, 1989.

[Williams, Zipser ‘89B] - Williams R.J., Zisper D., “Experimental analysis of the real-time recurrent learning algorithm”, Connection Science, Vol. 1, No. 1, 87-111, 1989.

[Scales ’85] – Scales, Introduction to Nonlinear Optimization, Springer-Verlag New York Inc. 1985.

[Dennis & Schnabel ‘83] - Dennis J.E., Schnabel R. B., Numerical Methods for Unconstrained Optimization and Nonlinear Equations, *Prentice-Hall*, NJ, 1983.

[Nocedal & Wright ‘99] - Nocedal, J., Wright, J.S., Numerical Optimization, Springer-Verlag, New York, NY, 1999.

[Golub & Van Loan, 1996] - Golub G.H., Van Loan C.F., Matrix Computations, 3rd Ed., *John Hopkins University Press*, Baltimore, MD, 1996.

[Williams ’92] – Williams R.J., “Training recurrent networks using the extended Kalman filter,” in *International Joint Conference on Neural Networks*, Baltomore, Vol. 4, 241, 1992.

[Catfolis ‘93] - Catfolis T., “a method for improving the real-time recurrent learning algorithm”, Neural Networks, Vol.6, pp. 807-821, 1993.

[Druaux, et al. ‘98] - Druaux F., Rogue E., Faure A., “Constrained RTRL to reduce learning rate and forgetting phenomenon”, Neural Processing Letters, 7, 161-167, 1998.

[Mak et al. ‘98] - Mak M.W., Ku K.W., Lu Y.L., “On the improvement of the real time recurrent learning algorithm for recurrent neural networks”, Neurocomputing, 24: (1-3), 13-26, 1998.

[Mandic & Chambers ‘99] - Mandic D.P., Chambers J.A., “Relating the slope of the activation function and the learning rate with a recurrent neural network”, Neural Computation, 11, 1069-1077, 1999.

[Mandic & Chambers ‘00] - Mandic D.P., Chambers J.A., “A normalized real time recurrent

learning algorithm", Signal Processing, 80, 1909-1916, 2000.

[Atiya & Parlos '00] - 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, No. 3, 697-709, 2000.

[Schmidhuber '92] - Schmidhuber J., "A Fixed Size Storage O (n^3) time complexity learning algorithm for fully recurrent continually running networks", Neural Computation, 4, pp. 243-248, 1992.

[Coelho '00] - Coelho P.H.G., "An extended RTRL training algorithm using Hessian Matrix", IEEE Proceedings on Neural Networks, Vol. 1, 563-568, 2000.

[Chang & Mak '98] - Chang W.F., Mak M.W., "A conjugate gradient learning algorithm for recurrent neural networks", NEUROCOMPUTING 24 (1-3): 173-189, 1998.

[Euliano & Principe '00] - Euliano N.R., Principe J.C., "Dynamic Sub grouping in RTRL providing a faster O (N^2) algorithm", Acoustics, Speech, and Signal Processing, Vol. 6, 3418-3421 2000.

[Hambaba '00] - Hambaba A, Robust hybrid architecture to signals from manufacturing and machine monitoring, *Journal Of Intelligent & Fuzzy Systems*, 9 (1-2): 29-41, 2000.

[Juang & Lin '01] - Juang C.F., Lin C.T., Noisy speech processing by recurrently adaptive fuzzy filters, *IEEE Transactions On Fuzzy Systems*, 9 (1): 139-152, 2001.

[Perez-Ortiz et al. '01] - Perez-Ortiz J.A., Calera-Rubio J., Forcada M.L., Online symbolic-sequence prediction with discrete-time recurrent neural networks A*rtifical Nneural Networks-icann 2001, proceedings lecture notes in computer science*, 2130, 719-724, 2001.

[Li et al. '02] - Li C.G., He S.B., Liao X.F., Yu J.B.,Using recurrent neural network for adaptive predistortion linearization of RF amplifiers, *International Journal Of Rf And Microwave Computer-Aided Engineering* , 12 (1), 125-130, 2002.

[Rumelhart et al. '86] – Rumelhart D.E., Hinton G.E., Williams R.J., "Learning representations by back-propagating errors," Nature, vol. 323, pp. 533-536, 1986.

[Selvan & Srinivasan '00] - Selvan S., Srinivasan R., "Recurrent neural network based efficient adaptive filtering technique for the removal of ocular artefacts from EEG", IETE Technical Review, 17 (1-2): 73-78, 2000.

[Blanco et al. ‘01] - Blanco A, Delgado M, Pegalajar MC, A real-coded genetic algorithm for training recurrent neural networks, *Neural Networks*, 14 (1): 93-105, 2001.

[Meyer ‘70] -Meyer, “ Theoretical and computational aspects of nonlinear regression,” in J.B. Rosen, O.L. Mangasarian and K. Ritter (eds), Nonlinear Programming, Academic Press, London and New York, 1970.