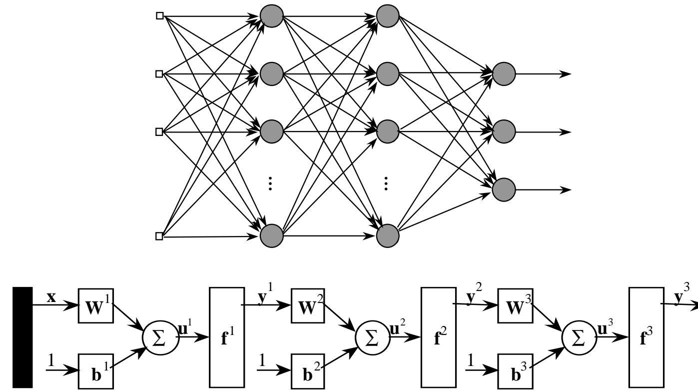


OPTIMISED TRAINING TECHNIQUES FOR FEEDFORWARD NEURAL NETWORKS

Leandro Nunes de Castro
lnunes@dca.fee.unicamp.br

Fernando José Von Zuben
vonzuben@dca.fee.unicamp.br

Technical Report

DCA-RT 03/98

July, 1998

State University of Campinas- UNICAMP

School of Electrical and Computer Engineering - FEEC

Department of Computer Engineering and Industrial Automation – DCA

SUMMARY

ABSTRACT **1**

1. INTRODUCTION **1**
- 1.1 NOTATION 3

2. FUNCTION APPROXIMATION **4**
- 2.1 EVALUATION OF THE APPROXIMATION LEVEL 4

3. NON-LINEAR UNCONSTRAINED OPTIMISATION TECHNIQUES **5**

4. EXAMPLE OF APPLICATION **8**

5. FIRST ORDER METHODS **8**
- 5.1 FIRST ORDER STANDARD BACKPROPAGATION WITH MOMENTUM (BPM) 9
  - *5.1.1 Matlab® source code* *9*
  - *5.1.2 Example of application* *10*
- 5.2 GRADIENT METHOD (GRAD) 11
  - *5.2.1 Matlab® source code* *11*
  - *5.2.2 Example of application* *13*

6. SECOND ORDER METHODS **14**
- 6.1 NEWTON'S METHOD 14
- 6.2 DAVIDON-FLETCHER-POWELL METHOD (DFP) 15
  - *6.2.1 Inverse construction* *15*
  - *6.2.2 Matlab® source code* *16*
  - *6.2.3 Example of application* *18*
- 6.3 BROYDEN-FLETCHER-GOLDFARB-SHANNO METHOD (BFGS) 18
  - *6.3.1 Matlab® source code* *18*
  - *6.3.2 Example of application* *20*
- 6.4 ONE-STEP SECANT METHOD (OSS) 21
  - *6.4.1 Matlab® source code* *21*
  - *6.4.2 Example of application* *23*
- 6.5 CONJUGATE GRADIENT METHOD 23
  - *6.5.1 The conjugate directions method* *24*
  - *6.5.2 Conjugate gradient method* *25*
- 6.6 NON-QUADRATIC PROBLEMS – POLAK-RIBIÈRE METHOD (PR) 25
  - *6.6.1 Matlab® source code* *26*
  - *6.6.2 Example of application* *27*
- 6.7 NON-QUADRATIC PROBLEMS – FLETCHER & REEVES METHOD (FR) 28
  - *6.7.1 Matlab® source code* *28*
  - *6.7.2 Example of application* *30*
- 6.8 SCALED CONJUGATE GRADIENT METHOD (SCGM) 30
  - *6.8.1 Exact calculation of the second order information* *31*
  - *6.8.2 Matlab® source code* *32*
  - *6.8.3 Example of application* *34*

7. LEARNING RATES **35**
- 7.1 INEXACT LINE-SEARCH 35
  - *7.1.1 Matlab® source code* *35*
- 7.2 EXACT LINE SEARCH – GOLDEN SECTION METHOD (GOLDSEC) 36
  - *7.2.1 Matlab® source code* *37*

8. SECONDARY FUNCTIONS **38**
- 8.1 RUNNING THE NET – (TESTNN) 38
  - *8.1.1 Example of application* *39*
- 8.2 CALCULATING THE PRODUCT H.V – (CALCHV) 39
- 8.3 CALCULATING THE SSE, GRADIENT VECTOR AND NET OUTPUT – (PROCESS) 40

9. REFERENCES **41**

OPTIMISED TRAINING TECHNIQUES FOR FEEDFORWARD NEURAL NETWORKS

LEANDRO NUNES DE CASTRO
lnunes@dca.fee.unicamp.br

FERNANDO JOSÉ VON ZUBEN
vonzuben@dca.fee.unicamp.br

Technical Report – DCA-RT 03/98 – July, 1998

Department of Computer Engineering and Industrial Automation, FEEC/UNICAMP, Brazil

Abstract

In this technical report we describe, analyse and present the source code for several non-linear unconstrained optimisation techniques applied to supervised training of feedforward networks. The functions and algorithms contained in this report were used in the simulations of the results presented in the Master thesis entitled: **Analyses and Synthesis of Artificial Neural Networks Training Strategies** (*Análise e Síntese de Estratégias de Treinamento de Redes Neurais Artificiais*). *All the codes presented were developed in Matlab® version 4.0 and some of them were updated for version 5.0. We start with a tutorial about the learning techniques and after each method is presented, its source code is given. It is not our goal to indicate directly the relative efficiency of these algorithms in an application, but analyse its main characteristics and present the Matlab® 4.0 source codes. We illustrate the default values specification for each algorithm presenting a simple example.*

Keywords: *non-linear optimisation, error backpropagation, Matlab®, artificial neural networks*.

1. Introduction

The training of multilayer (MLP) networks can be seen as a special case of function approximation, where no explicit model of the data is assumed [SHEPHERD, 1997]. We will review and present the source code for the following algorithms:

- standard backpropagation (BP);
- gradient method (GRAD);
- Fletcher & Reeves conjugate gradient (FR);
- Pollak-Ribière conjugate gradient (PR);

- MOLLER [1993] scaled conjugate gradient with the exact calculation of the second order information [PEARLMUTTER, 1994] (SCGM);
- BATTITI [1992] One-step Secant (OSS);
- Davidon-Fletcher-Powell quasi-Newton (DFP); and
- Broyden-Fletcher-Goldfarb-Shanno quasi-Newton (BFGS).

The error backpropagation has proven to be useful in the supervised training of feedforward multilayer networks when applied to several classification problems and non-linear static function mappings. Figure 2 illustrates the error backpropagation in a MLP network. There are cases in which the learning speed is a limiting factor to practical implementation of this kind of computational tool in the process of solving problems that require optimality and speed of convergence in the process of parameter adjustment.

In applications where real time results are not necessary, the time complexity of the algorithm can also result in a non-tractable problem. As an example, the intrinsic increase in complexity of the actual problems in the engineering field has produced a combinatorial explosion of the possible solution candidates, even when there are effective directions to exploring the solution space. Beyond that, among the search space methods, it is a common sense that there is no method superior to the others in every case. Though, several solutions achieved by specific techniques may not satisfy the constraints of the problem. One efficient way of dealing with this situation is exploring the computational processing potential available nowadays and start to operate with methods that present simultaneously multiple solution candidates, among which it can be chosen the best one according to a pre-specified criterion. When a solution is produced by means of artificial neural networks, the faster the learning speed, the more feasible this procedure. For example, a ten-time increase in the search speed for a solution allows finding ten times more solution candidates with the same computational effort. In this class, there are applications related to modelling, time series prediction and adaptive processes control [BATTITI, 1992].

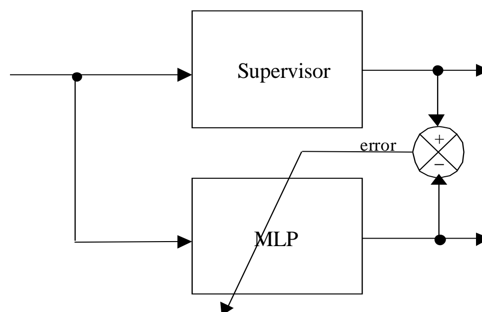


Figure 1: Comparison between the net output and the desired output of the system done by a supervisor (supervised training).

The multilayer artificial neural network supervised learning process is equivalent to a non-linear unconstrained optimisation problem, where a global error function is minimised from the parameters (weights and biases) adjustment of the neural net. This perspective of the supervised learning process leads to the development of training algorithms based upon results from conventional numerical analyses. The main numerical analyses procedures that can be implemented computationally use methods that require only the evaluation of the local gradient of the function, or methods that use also the second order derivatives. In the first case, the function is approximated by its first (constant) and second (linear) terms of its Taylor's expansion, and in the second case, the third (quadratic) term is also considered.

This report aims at describing and presenting the source code of some techniques that on average accelerate the convergence of the training process. As we are interested in the algorithms' speed of convergence, the generalisation capabilities acquired by the nets after convergence will not be studied. Some of these methods require few modifications in the standard algorithm, do not require choosing critic parameters of the net, like the learning rate and the momentum coefficient, and still result in high degrees of acceleration.

The multilayer network training can be viewed as a general problem of function approximation, though a brief introduction of this theory and the evaluation of the level of approximation will be presented. Then we are going to present optimisation techniques of the resulting approximation problem.

1.1 Notation

To standardise the functions implemented, we present the notation used for all the algorithms. Every function implemented is based upon matrix operation, and use *batch updating*.

The weights are initialised using a uniform distribution over the interval [-*val*, *val*].

Notation:

minerr	minimum value of the sum squared error (SSE) desired – stopping criteria
maxep	maximum number of epochs for training
ni	number of net input
nh	number of hidden units
no	number of output units
Nt	number of free parameters (weights)
np	number of samples (patterns)
P	matrix of input data – patterns ($np \times ni$)

T	matrix of desired output data – target ($np \times no$)
z	activation vector of the hidden units
y	activation vector of the output units
w1	weight matrix of the first layer ($ni + 1 \times nh$)
w2	weight matrix of the second layer ($nh + 1 \times no$)
alfa	step size (learning rate)
cm	momentum coefficient
dn	golden section (line search) threshold

2. Function Approximation

Consider the problem of approximating a function $g(.)$: $X \subset \Re^m \to \Re^r$ by an approximation model represented by the function $\hat{g}(.,\theta)$: $X \times \Re^{Nt} \to \Re^r$, where $\theta \in \Re^{Nt}$ (Nt finite) is the parameters vector.

The general approximation problem can be formally presented as follows [VON ZUBEN, 1996]: Consider the function $g(.)$: $X \subset \Re^m \to \Re^r$, that maps points in a compact subspace $X \subset \Re^m$ into points of another compact subspace $g[X] \subset \Re^r$. Based upon the input-output pairs $\{(\mathbf{x}_l, \mathbf{s}_l)\}_{l=1}^{np}$ sampled by a deterministic mapping defined by the function g as: $\mathbf{s}_l = g(\mathbf{x}_l) + \varepsilon_l$, $l = 1,\ldots,np$, and given the approximation model $\hat{g}(.,\theta): X \times \Re^{Nt} \to \Re^r$, determine the parameters vector $\theta^* \in \Re^{Nt}$ such as dist($g(.)$, $\hat{g}(.,\theta^*)$) $\le$ dist($g(.)$, $\hat{g}(.,\theta)$), for all $\theta \in \Re^{Nt}$, where the operator dist(.,.) measures the distance between two functions defined in space X. The vector ε_l represents the sampling error, and is assumed to be of zero mean and fixed variance. The solution of this problem if exists, is considered the best approximation and depends directly on the class of function that $\hat{g}$ belongs to.

2.1 Evaluation of the Approximation Level

In approximation problems using a finite number of sampled data and defined an approximation model $\hat{g}(.,\theta)$, the distance between the function to be approximated and its approximation dist($g(.)$, $\hat{g}(.,\theta)$) is a function only of the parameters vector $\theta \in \Re^{Nt}$. Taking the Euclidean norm as the distance measure, the following expression can be produced:

$$J(\theta) = \frac{1}{np} \sum_{l=1}^{np} \left(g(\mathbf{x}) - \hat{g}(\mathbf{x},\theta) \right)^2 . \qquad (1)$$

The functional J: $\Re^{Nt} \rightarrow \Re$ is called the error surface of the approximation problem, because it can be interpreted as a hyper-surface located "above" the parameters space $\Re^{Nt}$, in which each point $\theta \in \Re^{Nt}$ corresponds to the "height" $J(\theta)$.

Given the error surface, the approximation problem becomes an optimisation problem whose solution is the vector $\theta^* \in \Re^{Nt}$ that minimises $J(\theta)$:

$$\theta^* = \arg\min_{\theta \in \Re^{Nt}} J(\theta) . \tag{2}$$

During the approximation process of function $g(.)$ by the function $\hat{g}(.,\theta)$ obtained by the neural net, three kinds of errors must be considered [VAN DER SMAGT, 1994]: the representation error, the generalisation error and the optimisation error.

Representation error: let's first consider the case in which the whole sample set is available $\{(\mathbf{x}_l, \mathbf{s}_l)\}_{l=1}^{\infty}$. Assume also, that given $\{(\mathbf{x}_l, \mathbf{s}_l)\}_{l=1}^{\infty}$, it is possible to find an optimum weight vector θ^*. In this situation, the error depends on the flexibility of the approximation model $\hat{g}(.,\theta)$ and how adequate it is. This error is also known as the approximation error, or *bias*.

Generalisation error: in real world applications, only a finite number of samples is available or can be simultaneously used. Furthermore, the data can contain noise. The values of g for which no sample is available must be interpolated. A generalisation error, also known as estimation error or *variance*, can occur due to these factors.

Optimisation error: as the sample set is limited, the error is evaluated only upon the data that belong to this set.

Given the sample set $\{(\mathbf{x}_l, \mathbf{s}_l)\}_{l=1}^{np}$, the parameter vector $\theta = \theta^*$ must give the best approximation function based on a parametric representation $\hat{g}(.,\theta)$ and on the distance measure given by equation (1). If the error surface is continuous and differentiable with respect to the parameters vector (the parameters can assume any real value), then the most efficient non-linear unconstrained optimisation techniques can be applied to minimise $J(\theta)$.

3. Non-linear Unconstrained Optimisation Techniques

In the majority of the approximation models $\hat{g}(.,\theta)$, the optimisation problem presented in equation (2) has the disadvantage of being non-linear and non-convex, but the advantages of being unconstrained and allowing the application of variational calculus concepts in the process of obtaining the solution θ^*. These characteristics avoid the existence of an analytical solution, but

make it possible to obtain this solution by means of iterative processes, starting with an initial condition θ_0:

$$\theta_{i+1} = \theta_i + \alpha_i \mathbf{d}_i, \qquad i \geq 0, \tag{3}$$

where $\theta_i \in \Re^{Nt}$ is the parameters vector, $\alpha_i \in \Re^+$ is a scalar that defines the step size and $\mathbf{d}_i \in \Re^{Nt}$ is the search direction, all defined in iteration *i*. The optimisation algorithms revised in this report are applied in obtaining the step size and the search direction of the iterative process described in equation (3). The algorithms can be distinguished by the way in which they determine the step size and search direction [GROOT & WÜRTZ, 1994].

When the minimisation direction is available, it is necessary to define the step size $\alpha_i \in \Re^+$ in order to determine the parameters adjustment in that direction. Several line search procedures can be used to determine the step size. Though, we will be focused on determining the optimal direction. Usually, evaluations of the function are performed and its derivatives used for determining a minimum, global or local, and then finishing the learning process. There are methods available [BROMBERG & CHANG, 1992] that increase the chances of reaching the global minimum, but these methods require from hundreds to thousands function evaluations, though becoming highly computational intensive.

One usual way of classifying optimisation algorithms is according to the 'order' of information they use. By order we mean order of the derivatives of the objective (cost) function (in our case equation (1)). The first class of algorithms do not require more than the simple function evaluation in different points of the search space. No derivative is involved. These are called *methods with no differentiation*. The second class of algorithms uses the first derivative of the function to be minimised. These are called *first order methods*. The other class of algorithms that will be intensively studied in this report is the so-called *second order methods*, and make use of the second derivative of the cost function. One last division includes the algorithms whose parameters are adjusted in a heuristic way, i.e., through try and error procedures. These are classified as *heuristic methods*. In this work we will be focused on first and second order methods.

Figure 3 presents a diagram of the different training strategies that will be reviewed. The methods discussed in this report aim at determining local minima, which are points in a neighbourhood where the error function has the smallest value (see Figure 2). Theoretically, the second order methods are not more capable of reaching a global minimum than the first order ones.

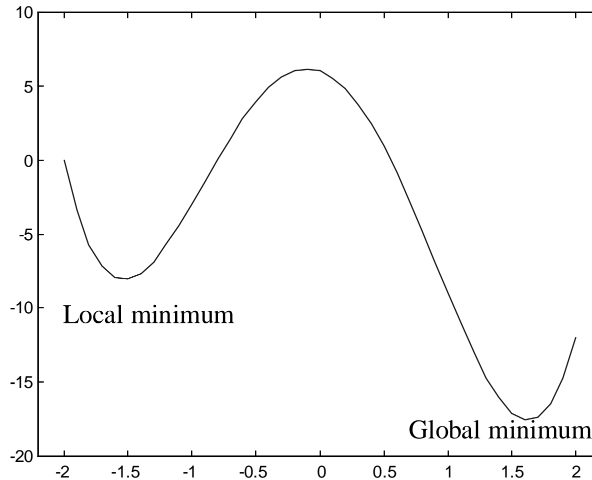


Figure 2: Scalar example of a function with one local and the global minimum.

The problem of determining the global minimum, even when a well-defined set of local minima is considered is difficult due to the fundamental impossibility of recognising a global minimum using only local information.

The key aspect of global optimisation is to know when to stop. Many efforts have been made directly in the problem of determining global minima. Recently, heuristic techniques like genetic algorithms (GA's) and simulated annealing (SA) have become very popular. However, none of these approaches, analytic or heuristic, guarantees reaching the global minimum of a smooth and continuous function in finite time and with limited computational resources.

Local minima are unique because of one of two reasons:

- the function is multi-modal;
- if the hessian matrix is singular in a local minimum, this minimum constitutes a compact set instead of an isolated point, i.e., the function value must be constant along a direction, a plane or a larger subspace [MCKEON & HALL, 1997].

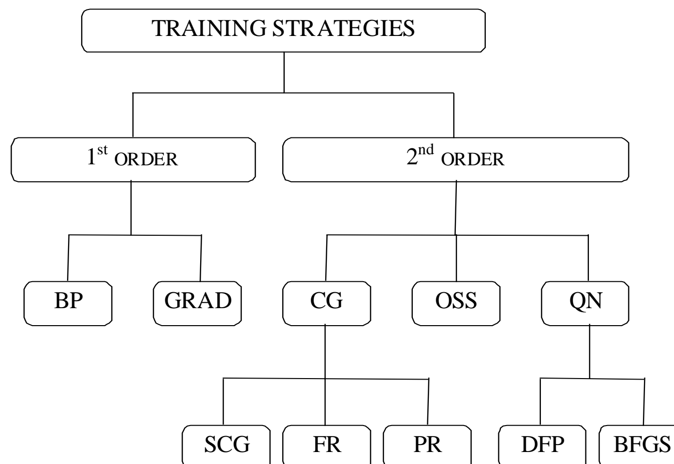


Figure 3: MLP neural network training strategies.

4. Example of Application

In this section we are going to present one example of application to illustrate how to specify the parameters for each algorithm presented in this work.

Consider the problem of approximating one period (2π) of the function $sin(x)\times cos(2x)$. Figure 4 presents the function to be approximated with the 42 uniformly distributed samples used.

For all the algorithms, the desired sum squared error is $SSE = 0.1$, the number of hidden units $nh = 10$, the maximum number of training epochs $maxep = 500$, the mean value of the final uncertainty interval for the golden section method is equal to 0.1%, and the weights were initialised uniformly over the interval [-0.5, 0.5]. Some parameters are particular for each algorithm and will be given only when the respective algorithm is presented.

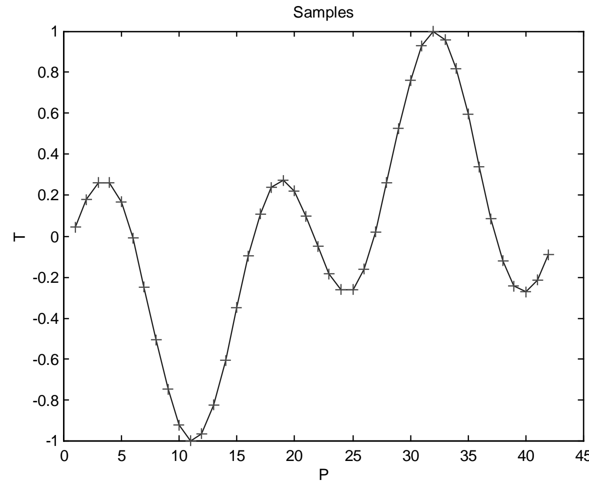


Figure 4: Function to be approximated. Training samples (+) uniformly distributed.

5. First Order Methods

The mean squared error (MSE) to be minimised can be presented considering its terms up to second order by equation (4):

$$J_{quad}(\theta) = J(\theta_i) + \nabla J(\theta_i)^T(\theta - \theta_i) + (\theta - \theta_i)^T \nabla^2 J(\theta_i)(\theta - \theta_i)\,, \tag{4}$$

where $\nabla J(\theta_i)$ is the gradient vector and $\nabla^2 J(\theta_i)$ is the hessian matrix of $J(\theta)$, both determined in the point $\theta = \theta_i$, and $J_{quad}(\theta)$ represents the $J(\theta)$ second order approximation.

In first order methods only the constant and linear terms in θ of the Taylor expansion are considered. These methods, where only the local gradient determines the minimising direction **d** (eq. (3)), are known as *steepest descent* or *gradient descent*.

5.1 First order standard backpropagation with momentum (BPM)

This method works as follows. When the net is in a state θ_i, the gradient $\nabla J(\theta_i)$ is determined and a minimising step in the opposite direction $\mathbf{d} = -\nabla J(\theta)$ is taken. The learning rule is given by equation (3).

In the standard backpropagation, the minimisation is performed using a fixed step α. Determining the step α is fundamental, because for very small values, the training time can become excessively high, and for very large values the parameters may diverge [HAYKIN, 1994]. The convergence speed is usually improved when a momentum term is added [RUMELHART *et. al.*, 1986].

$$\theta_{i+1} = \theta_i + \alpha_i \mathbf{d}_i + \beta_i \Delta\theta_{i-1}, \qquad i \geq 0. \tag{5}$$

This additional term usually avoids oscillation in the error behaviour, because it can be interpreted as the inclusion of an approximation of a second order information [HAYKIN, 1994].

5.1.1 Matlab® source code

The source code for this method is presented bellow:

```
function [w1, w2, y, sse] = bpm(P,T,nh,alfa,cm,minerr,maxep,val)
%
% BPM
% Main Program (function)
% MLP net with Backprop training
% Standard BP with Momentum
% Off-line Updating
% Author: Leandro Nunes de Castro
% Unicamp, January 1998
%

%-------------------------------------------------
% Definition and initialisation of the parameters
%-------------------------------------------------
P0 = P;
[np,ni] = size(P0);
[no] = size(T,2);
ep = 0;

w1 = 2.* val.*rand(ni+1,nh) - val;
w2 = 2.* val.*rand(nh+1,no) - val;

%------------------
% Network Training
%------------------
sse = 10; sseant = sse; veter = [];
P = [ones(np,1) P0];
fini = flops;  t0 = clock;

while (ep < maxep & sse > minerro)
   sseant = sse;   sse = 0;
   gdw1=zeros(ni+1,nh); gdw2=zeros(nh+1,no);
```

```
    %---------------
    % Forward Pass
    %---------------
    z0 = tanh(P*w1);
    z = [ones(np,1) z0];
    y = z*w2;                          % Linear output

    %--------------------------------
    % Correction and error calculus
    %--------------------------------
    dk = (T-y); gdw2 = z'*dk;          % Linear output
    w20 = reshape(w2(2:nh+1,:),nh,no);
    dj = (dk*w20').*(1-z0.^2);
    gdw1 = P'*dj;
    verr = (T-y); verr = reshape(verr,np*no,1);
    sse = verr'*verr;

    %--------------------------
    % Momentum update
    %--------------------------
    w1a = w1; w2a = w2;
    w1 = w1 + alfa*gdw1;
    w2 = w2 + alfa*gdw2;
    w1 = w1 + cm *(w1-w1a);
    w2 = w2 + cm *(w2-w2a);

    ep = ep + 1;
    vgrad = [reshape(gdw1,(ni+1)*nh,1); reshape(gdw2,(nh+1)*no,1)];
    ngrad = norm(vgrad);
    disp(sprintf('SSE: %f   Iteration: %u     ||GRAD||: %f',sse,ep,ngrad));
    veter = [veter sse];
 end;                              % end of stopping criteria
 fend = flops;  tflops = fend-fini;
 disp(sprintf('Flops total: %d Time: %d',tflops,etime(clock,t0)));

 % Ploting results
 figure(1); clf; plot (T,'r*'); hold on; plot(y,'g-'); drawnow;
 figure(2); semilogy(veter); title('BPM'); xlabel('Epochs'); ylabel('SSE');
```

5.1.2 Example of application

To run this algorithm for the problem presented in Section 4, we used the following command line:

```
>>  [w1, w2, y, sse] = bpm(P,T,10,0.001,0.9,0.1,500,0.5);
```

The result given by the net was:

```
>> SSE: 5.811590  Iteration: 500    ||GRAD||: 0.535431
>> Flops total: 12581289 Time: 1.741500e+001
```

Figure 5(a) and (b) presents the error behaviour and the resultant approximation, given by the BPM algorithm when applied to the $sin(x)\times cos(2x)$ problem, respectively.

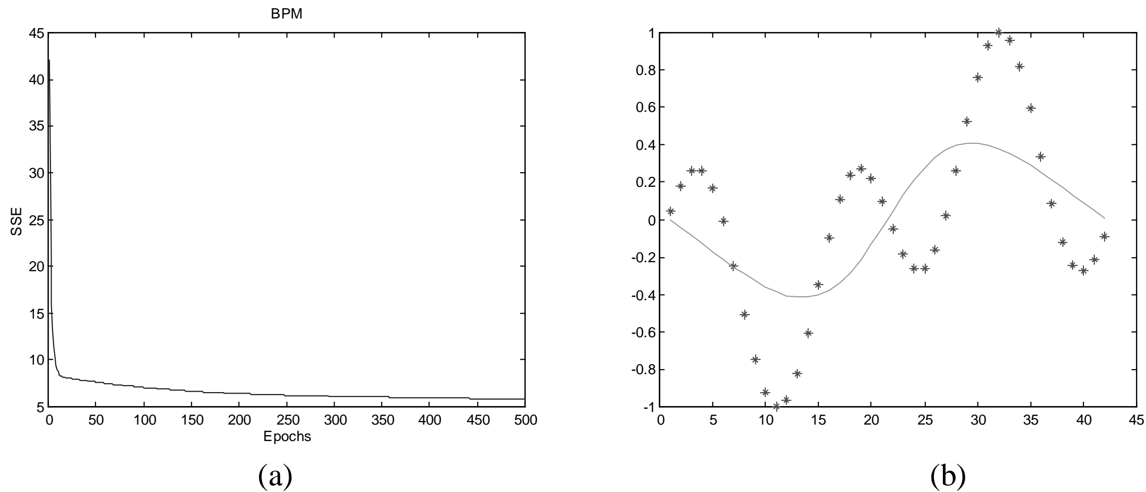


Figure 5: (a) Error behaviour. (b) Resultant approximation.

5.2 Gradient method (GRAD)

Among the methods that use search and differentiation, the gradient method is the simplest one at obtaining the search direction $\mathbf{d}_i$, because it uses only first order information. In iteration i, the direction $\mathbf{d}_i$ is defined as the greatest decreasing unit direction of function J.

$$\mathbf{d} = -\frac{\nabla J(\theta)}{\left\| \nabla J(\theta) \right\|}. \tag{6}$$

The adjustment rule is, then, given by:

$$\theta_{i+1} = \theta_i - \alpha_i \frac{\nabla J(\theta_i)}{\left\| \nabla J(\theta_i) \right\|}. \tag{7}$$

5.2.1 Matlab® source code

The source code for this method is as follows [VON ZUBEN, 1996]:

```
function [w1, w2, y, sse] = grad(P,T,nh,cm,minerr,maxep,val)
%
% GRAD
% Main Program (function)
% MLP net with Backprop training
% Gradient method
% Secondary functions: UNIDIM
% Off-line Updating
% Author: Leandro Nunes de Castro
% Unicamp, January 1998
%

%-------------------------------------------------
% Definition and initialisation of the parameters
%-------------------------------------------------
P0 = P;
[np,ni] = size(P0);
[no] = size(T,2);
```

```
ep = 0; alfa = 0.001;

w1 = 2.* val.*rand(ni+1,nh) - val;
w2 = 2.* val.*rand(nh+1,no) - val;

%-------------------
% Network Training
%-------------------
sse = 10; sseant = sse;
P = [ones(np,1) P0]; veter = []; vetalfa = [];
fini = flops;  t0 = clock; val = 0;
while (ep < maxep & sse > minerr)
   sseant = sse;   sse = 0;
   gdw1=zeros(ni+1,nh); gdw2=zeros(nh+1,no);

   %---------------
   % Forward pass
   %---------------
   z0 = tanh(P*w1);
   z = [ones(np,1) z0];
   y = z*w2;                           % linear output

   %---------------------------------
   % Correction and error calculus
   %---------------------------------
   dk = (T-y); gdw2 = z'*dk;
   w20 = reshape(w2(2:nh+1,:),nh,no);
   dj = (dk*w20').*(1-z0.^2);
   gdw1 = P'*dj;
   verr = (T-y); verr = reshape(verr,np*no,1);
   sse = verr'*verr;
   vgrad = [reshape(gdw1,(ni+1)*nh,1); reshape(gdw2,(nh+1)*no,1)];
   ngrad = norm(vgrad);

   alfa = unidim(gdw1,gdw2,w1,w2,alfa,cm,sse,sseant,T,P);

   %--------------------------
   % Momentum update
   %--------------------------
   w1a = w1; w2a = w2;
   w1 = w1 + alfa*gdw1/ngrad;
   w2 = w2 + alfa*gdw2/ngrad;
   w1 = w1 + cm *(w1-w1a);
   w2 = w2 + cm *(w2-w2a);

   ep = ep + 1;
   disp(sprintf('SSE: %f   Iteration: %u     ||GRAD||: %f      LR: 
%f',sse,ep,ngrad,alfa));
   veter = [veter sse]; vetalfa = [vetalfa alfa];
end;                             % end stopping criteria
fend = flops;  tflops = fend-fini;
disp(sprintf('Flops total: %d    Time: %d',tflops,etime(clock,t0)));

% Ploting results
figure(1); clf; plot (T,'r*'); hold on; plot(y,'g-'); drawnow;
figure(2); semilogy(veter); title('GRAD'); xlabel('Epochs'); ylabel('SSE');
figure(3); plot(vetalfa); title('Learning Rate');
xlabel('Epochs'); ylabel('Alfa');
```

The secondary functions' description will be presented later.

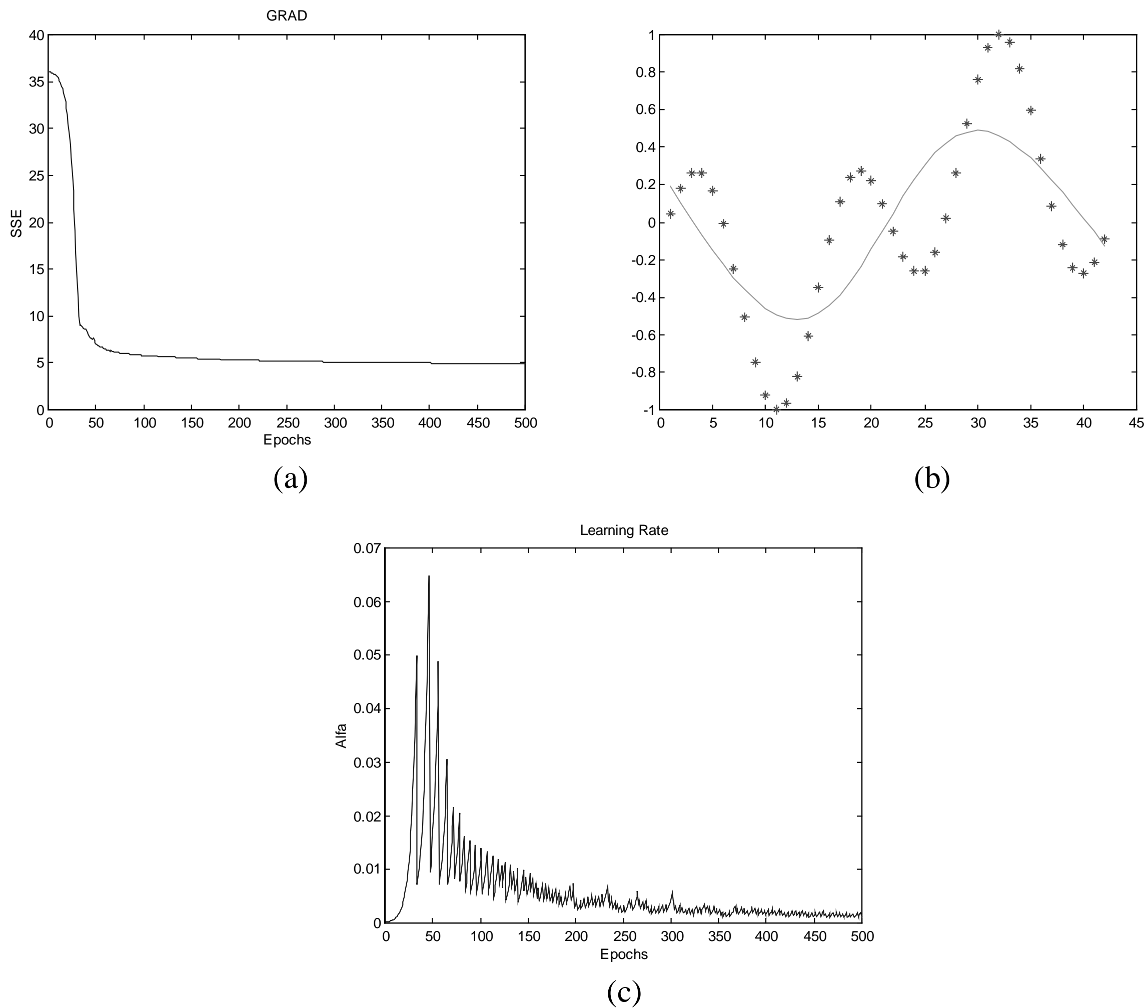
The stopping criterion adopted can also force the norm of the gradient vector to be smaller than a pre-specified value ε, i.e. $\|\nabla J(\theta_i)\| < \varepsilon$, instead of choosing the sum-squared error. The user can easily define it.

Whenever $J(\theta)$ has at least one minimum, the gradient method associated to this line search procedure is certainly going to reach a solution θ*, local minimum of problem (2). The inclusion of a momentum term in equation (3), as:

$$\theta_{i+1} = \theta_i - \alpha_i \frac{\nabla J(\theta_i)}{\|\nabla J(\theta_i)\|} + \beta_i(\theta_i - \theta_{i-1}), \tag{8}$$

is not recommended (but can be used) in this case because it makes the local minimum guaranteed convergence condition more difficult to be satisfied.

5.2.2 Example of application



(a) (b) (c)

Figure 6: (a) Error behaviour. (b) Resultant approximation. (c) Learning rate behaviour.

To run this algorithm for the problem presented in Section 4, we used the following command line:

```
>>  [w1, w2, y, sse] = grad(P,T,10,0.9,0.1,500,0.5);
```

The result given by the net was:

```
>> SSE: 4.923375  Iteration: 500    ||GRAD||: 0.670527      LR: 0.001176
>> Flops total: 14191068      Time: 2.742000e+001
```

Figure 6(a) and (b) presents the error behaviour and the resultant approximation, given by the GRAD algorithm when applied to the $sin(x)\times cos(2x)$ problem, respectively. Figure 6(c) presents the learning rate behaviour for the problem proposed.

6. Second Order Methods

Nowadays these methods are considered the most efficient way of training MLP neural networks [SHEPHERD, 1997]. These algorithms make use of mathematical fundamentals based upon non-linear unconstrained optimisation techniques, and though do not represent a natural connection with the biological inspiration initially proposed for the artificial neural networks (ANN's).

6.1 Newton's method

In this report we are not going to present the Newton's method source code, but we are going to make a brief introduction to it in order to present the basic concepts of second order techniques required for the comprehension of the following strategies. The practical application of the Newton's method to multilayer perceptrons is not recommended due to the fact that the exact calculation of the hessian matrix, its inversion, spectral analyses and storage, are very computational intensive. The hessian matrix is of order $Nt \times Nt$, where Nt is the net number of free parameters (weights and biases) to be adjusted [BATTITI, 1992; LUENBERGER, 1989; BAZARAA *et. al.*, 1993].

The vector θ_{i+1}, is the solution that exactly minimises $J(\theta)$ given by equation (4), though satisfying the optimality condition

$$\frac{\partial J_{quad}(\theta_{i+1})}{\partial \theta_{i+1}} = 0. \tag{9}$$

Applying equation (9) to equation (4) results

$$\theta_{i+1} = \theta_i - \left[\nabla^2 J(\theta_i)\right]^{-1} \nabla J(\theta_i), \tag{10}$$

where $\nabla^2 J(.)$ is the hessian matrix and $\nabla J(.)$ is the gradient vector.

Like the gradient method, as the function $J(\theta)$ is not necessarily quadratic, its quadratic approximation minimisation $J_{quad}(\theta)$ given by equation (4) may not lead to a solution θ_{i+1} such that $J(\theta_{i+1}) < J(\theta_i)$. The adjustment rule (10) becomes:

$$\theta_{i+1} = \theta_i - \alpha_i \left[\nabla^2 J(\theta_i)\right]^{-1} \nabla J(\theta_i) \,. \tag{11}$$

Detailed information about how to determine the step size α_i will be presented in a later section.

In the way the Newton's method was presented above, the convergence can not be guaranteed, because nothing can be said about the Hessian's sign, and it has to be a positive definite matrix for two reasons: to guarantee that the quadratic approximation has a minimum and the inverse existence. The latter is the necessary condition for solving equation (10) or (11) at each iteration.

6.2 Davidon-Fletcher-Powell method (DFP)

This method, like BFGS that will be presented later, is classified quasi-Newton method. The idea of the quasi-Newton methods is to iteratively approximate the inverse Hessian, such as:

$$\lim_{i \to \infty} \mathbf{H}_i = \nabla^2 J(\boldsymbol{\theta})^{-1} \tag{12}$$

These are, theoretically, considered the most sophisticated methods for solving non-linear unconstrained optimisation problems and represent the apices of the algorithm development through quadratic problem analyses.

For quadratic problems, they generate the conjugate directions of the conjugate gradient methods (that will be reviewed later) at the same time that constructs the inverse Hessian approximation. At each iteration the inverse Hessian is approximated by the sum of two rank 1 symmetric matrices, procedure usually called rank 2 correction.

6.2.1 Inverse construction

The idea is constructing the inverse Hessian, using first order information obtained along the learning iteration process. The actual approximation $\mathbf{H}_i$ is used at each iteration to define the next descent direction of the method. Ideally, the approximations converge to the inverse hessian matrix.

Suppose that the error functional $J(\theta)$ has continuous partial derivative up to the second order. Taking two points θ_i and θ_{i+1}, define $\mathbf{g}_i = \nabla J(\theta_i)^T$ e $\mathbf{g}_{i+1} = \nabla J(\theta_{i+1})^T$. If the Hessian, $\nabla^2 J(\boldsymbol{\theta})$, is constant, then we have:

$$\mathbf{q}_i \equiv \mathbf{g}_{i+1} - \mathbf{g}_i = \nabla^2 J(\theta)\mathbf{p}_i\,, \tag{13}$$

$$\mathbf{p}_i = \alpha_i \mathbf{d}_i\,. \tag{14}$$

We can then realise that the gradient evaluation in two points presents information about the hessian matrix $\nabla^2 J(\boldsymbol{\theta})$. $\theta \in \Re^{Nt}$, taking Nt linearly independent directions $\{\mathbf{p}_0, \mathbf{p}_1, \ldots, \mathbf{p}_{Nt-1}\}$, it is possible to uniquely determine $\nabla^2 J(\boldsymbol{\theta})$ if $\mathbf{q}_i$, $i = 0, 1, \ldots, Nt - 1$ is known. To do so, we have to iteratively apply equation (15) that follows, with $\mathbf{H}_0 = \mathbf{I}_{Nt}$ (dimension Nt identity matrix).

$$\mathbf{H}_{i+1} = \mathbf{H}_i + \frac{\mathbf{p}_i\mathbf{p}_i^T}{\mathbf{p}_i^T\mathbf{q}_i} - \frac{\mathbf{H}_i\mathbf{q}_i\mathbf{q}_i^T\mathbf{H}_i}{\mathbf{q}_i^T\mathbf{H}_i\mathbf{q}_i},\; i = 0, 1, \ldots, Nt - 1. \tag{15}$$

After Nt successive iterations, if $J(\theta)$ is a quadratic function, then $\mathbf{H}_{Nt} = \nabla^2 J(\boldsymbol{\theta})^{-1}$. As we are not usually dealing with quadratic problems, at each Nt iterations the algorithm re-initialisation must be done, i.e., take the minimisation direction like the direction opposite to the gradient vector direction and the hessian matrix as the identity matrix again.

6.2.2 Matlab® source code

The source code for this method is presented bellow:

```
function [w1, w2, y, sse] = dfp(P,T,nh,minerr,maxep,dn,val)
%
% DFP
% Main Program (function)
% MLP net with Backprop training
% Davidon-Fletcher-Powell quasi-Newton method
% Off-line Updating
% Author: Leandro Nunes de Castro
% Unicamp, January 1998
%

%--------------------------------------------------
% Definition and initialisation of the parameters
%--------------------------------------------------
P0 = P;
[np,ni] = size(P0);
[no] = size(T,2);
ep = 0; alfa = 0.001;

w1 = 2.* val.*rand(ni+1,nh) - val;
w2 = 2.* val.*rand(nh+1,no) - val;

%------------------
% Network Training
%------------------
sse = 10; sseant = sse; veterr = []; vetalfa = [];
Nt = (ni+1)*nh + (nh+1)*no; vgrad = zeros((ni+1)*nh + (nh+1)*no,1);
P = [ones(np,1) P0];  H = eye(Nt); p = vgrad;
fini = flops;  t0 = clock; val = 0;
while (ep < maxep & sse > minerr)
```

```
   sseant = sse;   sse = 0;
   gdw1=zeros(ni+1,nh); gdw2=zeros(nh+1,no);

   %---------------
   % Passo forward
   %---------------
   z0 = tanh(P*w1);
   z = [ones(np,1) z0];
   y = z*w2;                           % Linear output

   %---------------------------------
   % Correction and error calculus
   %---------------------------------
   dk = (T-y); gdw2 = z'*dk;           % Linear output
   w20 = reshape(w2(2:nh+1,:),nh,no);
   dj = (dk*w20').*(1-z0.^2);
   gdw1 = P'*dj;
   verr = (T-y); verr = reshape(verr,np*no,1);
   sse = verr'*verr;

   %---------------------------------
   % Gradient and search direction
   %---------------------------------
   vgrada = vgrad;
   vgrad = [reshape(gdw1,(ni+1)*nh,1); reshape(gdw2,(nh+1)*no,1)];
   ngrad = norm(vgrad);
   d = H*vgrad; d = d/norm(d);
   if rem(ep+1,Nt) == 0,
         d = vgrad; H = eye(Nt); disp('Restart');
   end;
   gdw1 = reshape(d(1:(ni+1)*nh),ni+1,nh);
   gdw2 = reshape(d((ni+1)*nh+1:(ni+1)*nh+(nh+1)*no),nh+1,no);

   %---------------------------------------------
   % Line search and inverse Hessian construction
   %---------------------------------------------
   alfa = goldsec(w1,w2,gdw1,gdw2,T,P,dn);
   pa = p; p = alfa*d; q = vgrad - vgrada;
   q=q/norm(q);
   if (p'*q) <= 0;                     % first-order necessary condition
         p = pa;
   end;
   H = H + ((p*p')/(p'*q)) - ((H*q*q'*H)/(q'*H*q));

   %-------------------------
   % Update
   %-------------------------
   w1 = w1 + alfa*gdw1;
   w2 = w2 + alfa*gdw2;

   ep = ep + 1;
   disp(sprintf('SSE: %f   Iteration: %u     ||GRAD||: %f      LR:
%f',sse,ep,ngrad,alfa));
   veter = [veter sse]; vetalfa = [vetalfa alfa];
end;                              % end stopping criteria
fend = flops;  tflops = fend-fini;
disp(sprintf('Flops Total: %d    Time: %d',tflops,etime(clock,t0)));

% Ploting results
figure(1); clf; plot(T,'r*'); hold on; plot(y,'g'); drawnow;
figure(2); semilogy(veter); title('DFP'); xlabel('Epochs'); ylabel('SSE');
```

6.2.3 Example of application

To run this algorithm for the problem presented in Section 4, we used the following command line:

```
>>  [w1, w2, y, sse] = dfp(P,T,10,0.1,500,0.0001,0.5);
```

The result given by the net was:

```
>> SSE: 0.099725  Iteration: 259    ||GRAD||: 2.723345      LR: 0.118034
>> Flops Total: 58900635      Time: 5.574000e+001
```

Figure 7(a) and (b) presents the error behaviour and the resultant approximation, given by the DFP algorithm when applied to the $sin(x)\times cos(2x)$ problem, respectively.

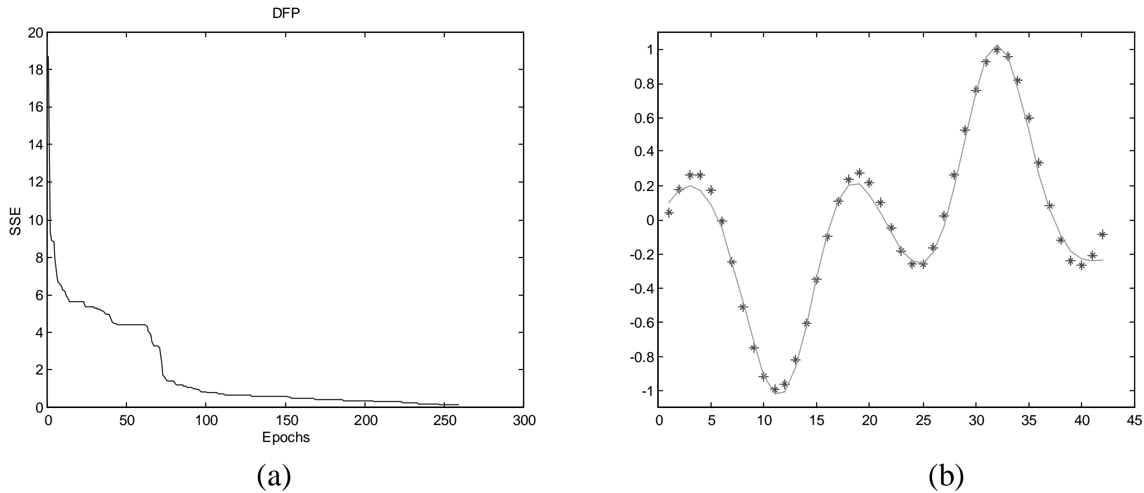


Figure 7: (a) Error behaviour. (b) Resultant approximation.

6.3 Broyden-Fletcher-Goldfarb-Shanno method (BFGS)

The basic difference between this method and the method presented in the last section (DFP) is in the way the inverse Hessian is constructed. The expression that allows determining the approximation inverse Hessian of BFGS method is presented in equation (16).

$$\mathbf{H}_{i+1} = \mathbf{H}_i + \frac{\mathbf{p}_i\mathbf{p}_i^T}{\mathbf{p}_i^T\mathbf{q}_i}\left[1 + \frac{\mathbf{q}_i^T\mathbf{H}_i\mathbf{q}_i}{\mathbf{p}_i^T\mathbf{q}_i}\right] - \frac{\mathbf{H}_i\mathbf{q}_i\mathbf{p}_i^T + \mathbf{p}_i\mathbf{q}_i^T\mathbf{H}_i}{\mathbf{p}_i^T\mathbf{q}_i} \tag{16}$$

The vectors $\mathbf{q}_i$ and $\mathbf{p}_i$ are determined as in expression (13) and (14), respectively.

6.3.1 Matlab® source code

The source code of this method is as follows:

```
function [w1, w2, y, sse] = dfp(P,T,nh,minerr,maxep,dn,val)
%
% DFP
% Main Program (function)
% MLP net with Backprop training
% Davidon-Fletcher-Powell quasi-Newton method
```

```
% Off-line Updating
% Author: Leandro Nunes de Castro
% Unicamp, January 1998
%

%-----------------------------------------------
% Definition and initialisation of the parameters
%-----------------------------------------------
P0 = P;
[np,ni] = size(P0);
[no] = size(T,2);
ep = 0; alfa = 0.001;

w1 = 2.* val.*rand(ni+1,nh) - val;
w2 = 2.* val.*rand(nh+1,no) - val;

%------------------
% Network Training
%------------------
sse = 10; sseant = sse; veterr = []; vetalfa = [];
Nt = (ni+1)*nh + (nh+1)*no; vgrad = zeros((ni+1)*nh + (nh+1)*no,1);
P = [ones(np,1) P0];  H = eye(Nt); p = vgrad;
fini = flops;  t0 = clock; val = 0;
while (ep < maxep & sse > minerr)
   sseant = sse;   sse = 0;
   gdw1=zeros(ni+1,nh); gdw2=zeros(nh+1,no);

   %---------------
   % Forward pass
   %---------------
   z0 = tanh(P*w1);
   z = [ones(np,1) z0];
   y = z*w2;                          % Linear output

   %--------------------------------
   % Correction and error calculus
   %--------------------------------
   dk = (T-y); gdw2 = z'*dk;          % Linear output
   w20 = reshape(w2(2:nh+1,:),nh,no);
   dj = (dk*w20').*(1-z0.^2);
   gdw1 = P'*dj;
   verr = (T-y); verr = reshape(verr,np*no,1);
   sse = verr'*verr;

   %--------------------------------
   % Gradient and search direction
   %--------------------------------
   vgrada = vgrad;
   vgrad = [reshape(gdw1,(ni+1)*nh,1); reshape(gdw2,(nh+1)*no,1)];
   ngrad = norm(vgrad);
   d = H*vgrad; d = d/norm(d);
   if rem(ep+1, Nt) == 0,
         d = vgrad; H = eye(Nt); disp('Restart');
   end;
   gdw1 = reshape(d(1:(ni+1)*nh),ni+1,nh);
   gdw2 = reshape(d((ni+1)*nh+1:(ni+1)*nh+(nh+1)*no),nh+1,no);

   %-----------------------------------------------
   % Line search and inverse Hessian construction
   %-----------------------------------------------
   alfa = goldsec(w1,w2,gdw1,gdw2,T,P,dn);
```

```
   pa = p; p = alfa*d; q = vgrad - vgrada;
   q=q/norm(q);
   if (p'*q) <= 0;                     % first-order necessary condition
         p = pa;
   end;
   H=H+((p*p')/(p'*q))*(1+(q'*H*q)/(q'*p))-((H*q*p'+p*q'*H)/(q'*p));


   %-------------------------
   % Update
   %-------------------------
   w1 = w1 + alfa*gdw1;
   w2 = w2 + alfa*gdw2;

   ep = ep + 1;
   disp(sprintf('SSE: %f   Iteration: %u     ||GRAD||: %f      LR:
%f',sse,ep,ngrad,alfa));
   veter = [veter sse]; vetalfa = [vetalfa alfa];
end;                                % end stopping criteria
fend = flops;  tflops = fend-fini;
disp(sprintf('Flops Total: %d    Time: %d',tflops,etime(clock,t0)));

% Ploting results
figure(1); clf; plot(T,'r*'); hold on; plot(y,'g'); drawnow;
figure(2); semilogy(veter); title('BFGS'); xlabel('Epochs'); ylabel('SSE');
```

6.3.2 Example of application

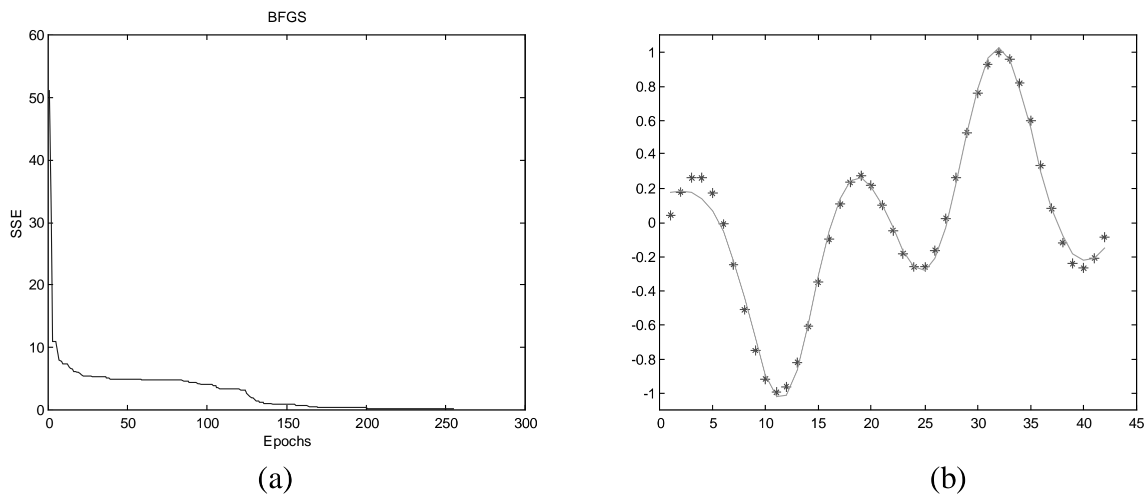
To run this algorithm for the problem presented in Section 4, we used the following command line:

```
>>  [w1, w2, y, sse] = bfgs(P,T,10,0.1,500,0.0001,0.5);
```

The result given by the net was:

```
>> SSE: 0.096948  Iteration: 255    ||GRAD||: 0.509881      LR: 0.072949
>> Flops Total: 59924742      Time: 5.287600e+001
```

Figure 8(a) and (b) presents the error behaviour and the resultant approximation, given by the BFGS algorithm when applied to the $sin(x)\times cos(2x)$ problem, respectively.



(a) (b)

Figure 8: (a) Error behaviour. (b) Resultant approximation.

6.4 One-Step Secant method (OSS)

The term one-step secant comes from the fact that the derivatives are approximated by secants evaluated in two points of the function (in this case the function is the gradient). One advantage of this method presented by BATTITI [1992; 1994] is that it has order $O(Nt)$ complexity, i.e., it is linear in relation to the number Nt of parameters, while the methods DFP and BFGS have order $O(Nt^2)$ complexity.

The main reason for the computational effort reduction, when compared to the previous methods (DFP e BFGS), is that the updating (search) direction (eq. (3)) is calculated only based upon vectors determined by the gradients, and there is no further storage of the approximation of the inverse Hessian. The new search direction $\mathbf{d}_{i+1}$ is obtained as follows:

$$\mathbf{d}_{i+1} = -\mathbf{g}_i + A_i\mathbf{s}_i + B_i\mathbf{q}_i, \tag{17}$$

where:

$$\mathbf{s}_i = \theta_{i+1} - \theta_i = \mathbf{p}_i, \tag{18}$$

$$A_i = -\left[1 + \frac{\mathbf{q}_i^T\mathbf{q}_i}{\mathbf{s}_i^T\mathbf{q}_i}\right]\frac{\mathbf{s}_i^T\mathbf{g}_i}{\mathbf{s}_i^T\mathbf{q}_i} + \frac{\mathbf{q}_i^T\mathbf{g}_i}{\mathbf{s}_i^T\mathbf{q}_i}; \qquad B_i = \frac{\mathbf{s}_i^T\mathbf{g}_i}{\mathbf{s}_i^T\mathbf{q}_i}. \tag{19}$$

The vectors $\mathbf{q}_i$ and $\mathbf{p}_i$ are determined by the expression (13) and (14), respectively.

6.4.1 Matlab® source code

The source code for this method is given bellow:

```
function [w1, w2, y, sse] = oss(P,T,nh,minerr,maxep,dn,val)
%
% OSS
% Main Program (function)
% MLP net with Backprop training
% One-Step Secant method
% Off-line Updating
% Author: Leandro Nunes de Castro
% Unicamp, January 1998
%

%---------------------------------------------------
% Definition and initialisation of the parameters
%---------------------------------------------------
P0 = P;
[np,ni] = size(P0);
[no] = size(T,2);
ep = 0; alfa = 0.001;

w1 = 2.* val.*rand(ni+1,nh) - val;
w2 = 2.* val.*rand(nh+1,no) - val;

%------------------
% Network Training
%------------------
```

```
 sse = 10; sseant = sse; val = 0;
 P = [ones(np,1) P0]; Ac = 0; Bc = 0;
 Nt = (ni+1)*nh + (nh+1)*no; vgrad = zeros((ni+1)*nh + (nh+1)*no,1);
 beta = 0; d = vgrad; veterr = []; vetalfa = [];
 fini = flops; vw = [reshape(w1,(ni+1)*nh,1); reshape(w2,(nh+1)*no,1)];
 t0 = clock;
 while (ep < maxep & sse > minerr)
   sseant = sse;   sse = 0;
   gdw1=zeros(ni+1,nh); gdw2=zeros(nh+1,no);

   %---------------
   % Passo forward
   %---------------
   z0 = tanh(P*w1);
   z = [ones(np,1) z0];
   y = z*w2;                           % Linear output

   %--------------------------------
   % Correction and error calculus
   %--------------------------------
   dk = (T-y); gdw2 = z'*dk;           % Linear output
   w20 = reshape(w2(2:nh+1,:),nh,no);
   dj = (dk*w20').*(1-z0.^2);
   gdw1 = P'*dj;
   verr = (T-y); verr = reshape(verr,np*no,1);
   sse = verr'*verr;

   %--------------------------------
   % Gradient and search direction
   %--------------------------------
   vgrada = vgrad; vwa = vw;
   vgrad = [reshape(gdw1,(ni+1)*nh,1); reshape(gdw2,(nh+1)*no,1)];
   vw = [reshape(w1,(ni+1)*nh,1); reshape(w2,(nh+1)*no,1)];
   ngrad = norm(vgrad); vgrad = vgrad/ngrad;
   p = alfa * d; q = vgrad - vgrada;
   d = vgrad + Ac * p + Bc * q;
   if ep >= 1,
         p = p/norm(p);
         Ac = (1+(q'*q)/(p'*q))*((p'*vgrad)/(p'*q)) - (q'*vgrad)/(p'*q);
         Bc = -(p'*vgrad)/(p'*q);
   end;
   if rem(ep+1,round(sqrt(Nt))) == 0,
         d = vgrad; disp('Restart');
   end;
   gdw1 = reshape(d(1:(ni+1)*nh),ni+1,nh);
   gdw2 = reshape(d((ni+1)*nh+1:(ni+1)*nh+(nh+1)*no),nh+1,no);
   alfa = goldsec(w1,w2,gdw1,gdw2,T,P,dn);

   %-------------------------
   % Update
   %-------------------------
   w1 = w1 + alfa*gdw1;
   w2 = w2 + alfa*gdw2;

   ep = ep + 1;
   disp(sprintf('SSE: %f   Iteration: %u     ||GRAD||: %f      LR:
%f',sse,ep,ngrad,alfa));
   veter = [veter sse]; vetalfa = [vetalfa alfa];
end;                            % end stopping criteria
fend = flops;  tflops = fend-fini;
disp(sprintf('Flops Total: %d    Time: %d',tflops,etime(clock,t0)));
```

```
% Ploting results
figure(1); clf; plot(T,'r*'); hold on; plot(y,'g'); drawnow;
figure(2); semilogy(veter); title('OSS'); xlabel('Epochs'); ylabel('SSE');
```

6.4.2 Example of application

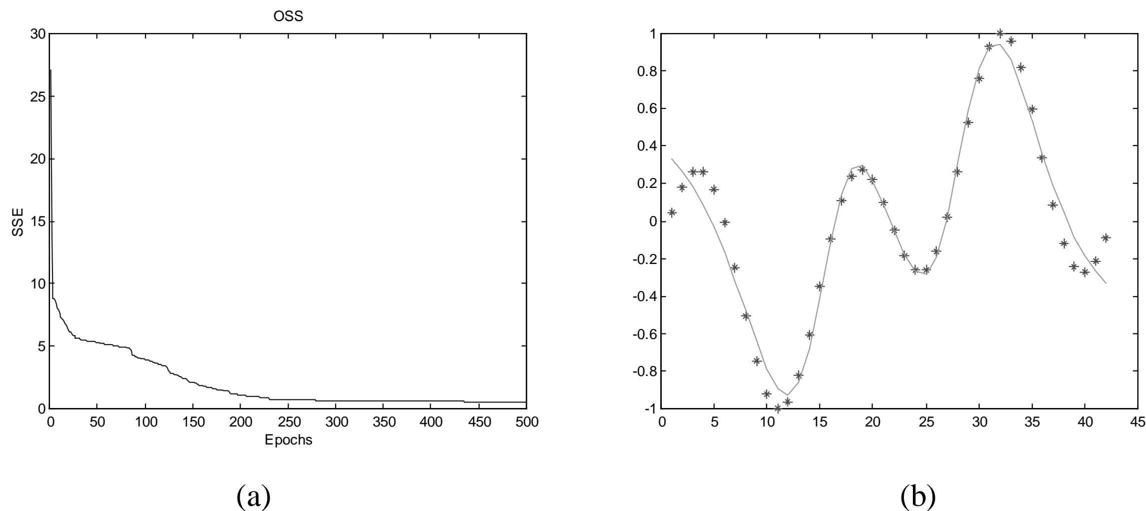
To run this algorithm for the problem presented in Section 4, we used the following command line:

```
>>  [w1, w2, y, sse] = oss(P,T,10,0.1,500,0.0001,0.5);
```

The result given by the net was:

```
>> SSE: 0.551896  Iteration: 500    ||GRAD||: 0.136418      LR: 0.004065
>> Flops Total: 80304227      Time: 8.851700e+001
```

Figure 9(a) and (b) presents the error behaviour and the resultant approximation, given by the OSS algorithm when applied to the *sin*(*x*)×*cos*(2*x*) problem, respectively.



(a) (b)

Figure 9: (a) Error behaviour. (b) Resultant approximation.

6.5 Conjugate Gradient method

It is general agreement of the numerical analyses community that the class of optimisation methods called *conjugate gradient*, deal effectively with large-scale problems [VAN DER SMAGT, 1994].

Conjugate gradient methods have their strategies based upon the general model presented in the standard algorithm, but choose the search direction $\mathbf{d}_i$, the step size α_i and the momentum coefficient β_i (equation (5)) more efficiently using second order information. They are designed to demand less calculation than the Newton's method and present higher convergence rates than the gradient method.

Before presenting the conjugate gradient method it is necessary to introduce an intermediate result called *conjugate directions method*.

6.5.1 The conjugate directions method

The adaptation law of the processes under study are like equation (3), and, if convergence is achieved, the optimal solution $\theta^* \in \Re^{Nt}$ can be expressed by:

$$\theta^* = \alpha_0 \mathbf{d}_0 + \alpha_1 \mathbf{d}_1 + \ldots = \sum_i \alpha_i \mathbf{d}_i \ .$$

Assuming as hypotheses that the set $\{\mathbf{d}_0, \mathbf{d}_1, \ldots, \mathbf{d}_{Nt\text{-}1}\}$ forms a base of $\Re^{Nt}$ and $\alpha = [\alpha_0 \ldots \alpha_{Nt\text{-}1}]^T$ is the representation of θ^* in this base, then it is possible to obtain θ^* in Nt iterations of equation (3)

$$\theta^* = \alpha_0 \mathbf{d}_0 + \alpha_1 \mathbf{d}_1 + \ldots + \alpha_{Nt-1} \mathbf{d}_{Nt-1} = \sum_{i=0}^{Nt-1} \alpha_i \mathbf{d}_i \ . \tag{20}$$

Given a symmetric matrix $\mathbf{A}$ of dimension $Nt \times Nt$, the directions $\mathbf{d}_i \in \Re^{Nt}$, $i = 0, \ldots, Nt\text{-}1$, are said to be $\mathbf{A}$-conjugate if: $\mathbf{d}_j^T \mathbf{A} \mathbf{d}_i = 0$, for $i \neq j$ and $i,j = 0, \ldots, Nt\text{-}1$.

If matrix $\mathbf{A}$ is definite-positive, the set of Nt $\mathbf{A}$-conjugate directions forms a base of $\Re^{Nt}$. In this way, the coefficients $\alpha_j^*, j = 1, \ldots, Nt - 1$, can be determined by the following procedure.

Given a symmetric matrix $\mathbf{A}$, positive-definite and of dimension $Nt \times Nt$, left multiplying equation (20) by $\mathbf{d}_j^T \mathbf{A}$, with $0 \leq j \leq Nt\text{-}1$, results:

$$\mathbf{d}_j^T \mathbf{A} \theta^* = \sum_{i=0}^{Nt-1} \alpha_i^* \mathbf{d}_j^T \mathbf{A} \mathbf{d}_i, \qquad j = 1, \ldots, Nt - 1 \ . \tag{21}$$

Choosing the directions $\mathbf{d}_i \in \Re^{Nt}$, $\mathbf{A}$-conjugate, it is possible to apply the results presented above to obtain:

$$\alpha_j^* = \frac{\mathbf{d}_j^T \mathbf{A} \theta^*}{\mathbf{d}_j^T \mathbf{A} \mathbf{d}_j}, \qquad j = 1, \ldots, Nt - 1. \tag{22}$$

It is necessary to eliminate θ^* from expression (22), and to do that, two additional hypotheses are necessary:

- Suppose the problem is quadratic, i.e., $J(\theta) = \frac{1}{2} \theta^T \mathbf{Q} \theta - \mathbf{b}^T \theta$

  Then in the optimum solution θ^*, is valid the expression:

$$\nabla J(\theta^*) = 0 \Rightarrow \mathbf{Q}\theta^* - \mathbf{b} = 0 \Rightarrow \mathbf{Q}\theta^* = \mathbf{b}. \tag{23}$$

- Suppose $\mathbf{A} = \mathbf{Q}$.

  Though, equation (22) results in:

$$\alpha_j^* = \frac{\mathbf{d}_j^T \mathbf{b}}{\mathbf{d}_j^T \mathbf{A} \mathbf{d}_j}, \qquad j = 1, \ldots, Nt-1, \tag{24}$$

  and the optimal solution θ* is given by:

$$\theta^* = \sum_{j=0}^{Nt-1} \frac{\mathbf{d}_j^T \mathbf{b}}{\mathbf{d}_j^T \mathbf{A} \mathbf{d}_j} \mathbf{d}_j. \tag{25}$$

Assuming iterative solution with θ* expressed as in equation (25), the coefficients α_j^*, $j = 1, \ldots, Nt-1$, are given by:

$$\theta^* = \theta_0 + \alpha_0^* \mathbf{d}_0 + \alpha_1^* \mathbf{d}_1 + \ldots + \alpha_{Nt-1}^* \mathbf{d}_{Nt-1}, \tag{26}$$

$$\alpha_j^* = \frac{\mathbf{d}_j^T \mathbf{Q} (\theta^* - \theta_0)}{\mathbf{d}_j^T \mathbf{Q} \mathbf{d}_j}, \qquad j = 1, \ldots, Nt-1. \tag{27}$$

In iteration *j*, and taking into account equation (26), we obtain:

$$\alpha_j^* = -\frac{\mathbf{d}_j^T \nabla J(\theta_j)}{\mathbf{d}_j^T \mathbf{Q} \mathbf{d}_j}, \qquad j = 1, \ldots, Nt-1, \tag{28}$$

and the adjustment rule of the conjugate directions method is given by:

$$\theta_{i+1} = \theta_i - \frac{\mathbf{d}_i^T \nabla J(\theta_i)}{\mathbf{d}_i^T \mathbf{Q} \mathbf{d}_i} \mathbf{d}_i. \tag{29}$$

6.5.2 Conjugate gradient method

Before applying the adjustment rule given by equation (29), it is necessary to obtain the **Q**-conjugate directions $\mathbf{d}_i \in \Re^{Nt}$, $i=0,\ldots,Nt-1$. One way of determining these directions is taking them as follows [BAZARAA *et. al.*, 1993]:

$$\begin{cases} \mathbf{d}_0 = -\nabla J(\theta_0) \\ \mathbf{d}_{i+1} = -\nabla J(\theta_{i+1}) + \beta_i \mathbf{d}_i \qquad i \geq 0 \end{cases}$$
$$\text{com} \quad \beta_i = \frac{\nabla J(\theta_{i+1})^T \mathbf{Q} \mathbf{d}_i}{\mathbf{d}_i^T \mathbf{Q} \mathbf{d}_i} \tag{30}$$

6.6 Non-quadratic problems – Polak-Ribière method (PR)

The derivation of the previous equations was made supposing quadratic problems, what is not always true. To adapt the previous equations to non-quadratic problems, the matrix **Q** must be

approximated by the hessian matrix calculated in the point θ_i. One of these approximations is given by the Polak-Ribière method.

In PR method we use a line search procedure to determine the step size α, and approximate the parameter β by the following expression:

$$\beta_i = \frac{\mathbf{g}_{i+1}^T(\mathbf{g}_{i+1} - \mathbf{g}_i)}{\mathbf{g}_i^T \mathbf{g}_i} \tag{31}$$

6.6.1 Matlab® source code

The source code for this method is:

```
function [w1, w2, y, sse] = pr(P,T,nh,minerr,maxep,dn,val)
%
% PR
% Main Program (function)
% MLP net with Backprop training
% Pllak-Ribière conjugate gradient method
% Off-line Updating
% Author: Leandro Nunes de Castro
% Unicamp, January 1998
%

%--------------------------------------------------
% Definition and initialisation of the parameters
%--------------------------------------------------
P0 = P;
[np,ni] = size(P0);
[no] = size(T,2);
ep = 0; alfa = 0.001;

w1 = 2.* val.*rand(ni+1,nh) - val;
w2 = 2.* val.*rand(nh+1,no) - val;

%------------------
% Network Training
%------------------
sse = 10; sseant = sse;
P = [ones(np,1) P0]; veterr = []; vetalfa = [];
Nt = (ni+1)*nh + (nh+1)*no; vgrad = zeros((ni+1)*nh + (nh+1)*no,1);
beta = 0; d = vgrad; fini = flops; t0 = clock; val = 0;
while (ep < maxep & sse > minerr)
   sseant = sse;   sse = 0;
   gdw1=[]; gdw2=[];

   %---------------
   % Forward pass
   %---------------
   z0 = tanh(P*w1);
   z = [ones(np,1) z0];
   y = z*w2;                           % Linear output

   %---------------------------------
   % Correction and error calculus
   %---------------------------------
   dk = (T-y); gdw2 = z'*dk;           % Linear output
```

```
   w20 = reshape(w2(2:nh+1,:),nh,no);
   dj = (dk*w20').*(1-z0.^2);
   gdw1 = P'*dj;
   verr = (T-y); verr = reshape(verr,np*no,1);
   sse = verr'*verr;

   %---------------------------------
   % Gradient and search direction
   %---------------------------------
   vgrada = vgrad;
   vgrad = [reshape(gdw1,(ni+1)*nh,1); reshape(gdw2,(nh+1)*no,1)];
   ngrad = norm(vgrad); vgrad = vgrad/ngrad;
   d = vgrad + beta*d;
   if ep >= 1,
         beta = (vgrad'*(vgrad-vgrada))/(vgrada'*vgrada);
   end;
   if rem(ep+1,Nt) == 0,
            d = vgrad; disp('Restart');
      end;
   gdw1 = reshape(d(1:(ni+1)*nh),ni+1,nh);
   gdw2 = reshape(d((ni+1)*nh+1:(ni+1)*nh+(nh+1)*no),nh+1,no);
   alfa = goldsec(w1,w2,gdw1,gdw2,T,P,dn);

   %--------------------------
   % Updating
   %--------------------------
   w1 = w1 + alfa*gdw1;
   w2 = w2 + alfa*gdw2;

   ep = ep + 1;
   disp(sprintf('SSE: %f   Iteration: %u     ||GRAD||: %f      LR:
%f',sse,ep,ngrad,alfa));
   veter = [veter sse]; vetalfa = [vetalfa alfa];
end;                             % end of stopping criteria
fend = flops;  tflops = fend-fini;
disp(sprintf('Flops total: %d    Time: %d',tflops,etime(clock,t0)));

% Ploting results
figure(1); clf; plot(T,'r*'); hold on; plot(y,'g'); drawnow;
figure(2); semilogy(veter); title('FR'); xlabel('Epochs'); ylabel('SSE');
```

6.6.2 Example of application

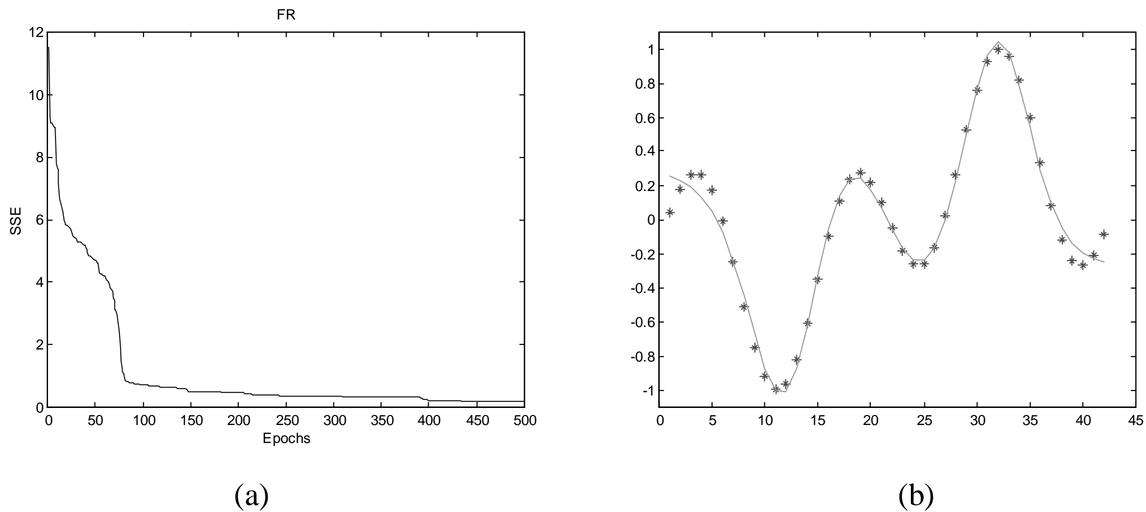
To run this algorithm for the problem presented in Section 4, we used the following command line:

```
>>  [w1, w2, y, sse] = pr(P,T,10,0.1,500,0.0001,0.5);
```

The result given by the net was:

```
>> SSE: 0.171918  Iteration: 500    ||GRAD||: 0.153779      LR: 0.006578
>> Flops total: 73122665      Time: 1.196120e+002
```

Figure 10(a) and (b) presents the error behaviour and the resultant approximation, given by the PR algorithm when applied to the $sin(x) \times cos(2x)$ problem, respectively.



(a) (b)

Figure 10: (a) Error behaviour. (b) Resultant approximation.

6.7 Non-quadratic problems – Fletcher & Reeves method (FR)

It is a conjugate direction method like Polak-Ribière, and the difference resides in the way the parameter β is determined.

$$\beta_i = \frac{\|\mathbf{g}_{i+1}\|^2}{\|\mathbf{g}_i\|^2} \tag{32}$$

6.7.1 Matlab® source code

The source code for this method is:

```
function [w1, w2, y, sse] = fr(P,T,nh,minerr,maxep,dn,val)
%
% FR
% Main Program (function)
% MLP net with Backprop training
% Fletcher & Reeves conjugate gradient method
% Off-line Updating
% Author: Leandro Nunes de Castro
% Unicamp, January 1998
%

%--------------------------------------------------
% Definition and initialisation of the parameters
%--------------------------------------------------
P0 = P;
[np,ni] = size(P0);
[no] = size(T,2);
ep = 0; cm = .7; alfa = 0.001;

w1 = 2.* val.*rand(ni+1,nh) - val;
w2 = 2.* val.*rand(nh+1,no) - val;

%------------------
% Network Training
```

```
%------------------
sse = 10; sseant = sse;
P = [ones(np,1) P0]; veterr = []; vetalfa = [];
Nt = (ni+1)*nh + (nh+1)*no; vgrad = zeros((ni+1)*nh + (nh+1)*no,1);
beta = 0; d = vgrad; fini = flops; t0 = clock; val = 0;
while (ep < maxep & sse > minerr)
   sseant = sse;   sse = 0;
   gdw1=[]; gdw2=[];

   %---------------
   % Forward pass
   %---------------
   z0 = tanh(P*w1);
   z = [ones(np,1) z0];
   y = z*w2;                           % Linear output

   %--------------------------------
   % Correction and error calculus
   %--------------------------------
   dk = (T-y); gdw2 = z'*dk;           % Linear output
   w20 = reshape(w2(2:nh+1,:),nh,no);
   dj = (dk*w20').*(1-z0.^2);
   gdw1 = P'*dj;
   verr = (T-y); verr = reshape(verr,np*no,1);
   sse = verr'*verr;

   %--------------------------------
   % Gradient and search direction
   %--------------------------------
   vgrada = vgrad;
   vgrad = [reshape(gdw1,(ni+1)*nh,1); reshape(gdw2,(nh+1)*no,1)];
   ngrad = norm(vgrad); vgrad = vgrad/ngrad;
   d = vgrad + beta*d;
   if ep >= 1,
         beta = (vgrad'*vgrad)/(vgrada'*vgrada);
         beta = max(0,beta);
   end;
   if rem(ep+1,Nt) == 0,
            d = vgrad; disp('Restart');
      end;
   gdw1 = reshape(d(1:(ni+1)*nh),ni+1,nh);
   gdw2 = reshape(d((ni+1)*nh+1:(ni+1)*nh+(nh+1)*no),nh+1,no);
   alfa = goldsec(w1,w2,gdw1,gdw2,T,P,.0001);

   %-------------------------
   % Updating
   %-------------------------
   w1 = w1 + alfa*gdw1;
   w2 = w2 + alfa*gdw2;

   ep = ep + 1;
   disp(sprintf('SSE: %f   Iteration: %u     ||GRAD||: %f      LR:
%f',sse,ep,ngrad,alfa));
   veter = [veter sse]; vetalfa = [vetalfa alfa];
end;                                % end of stopping criteria
fend = flops;  tflops = fend-fini;
disp(sprintf('Flops total: %d    Time: %d',tflops,etime(clock,t0)));

% Ploting results
figure(1); clf; plot(T,'r*'); hold on; plot(y,'g'); drawnow;
figure(2); semilogy(veter); title('FR'); xlabel('Epochs'); ylabel('SSE');
```

6.7.2 Example of application

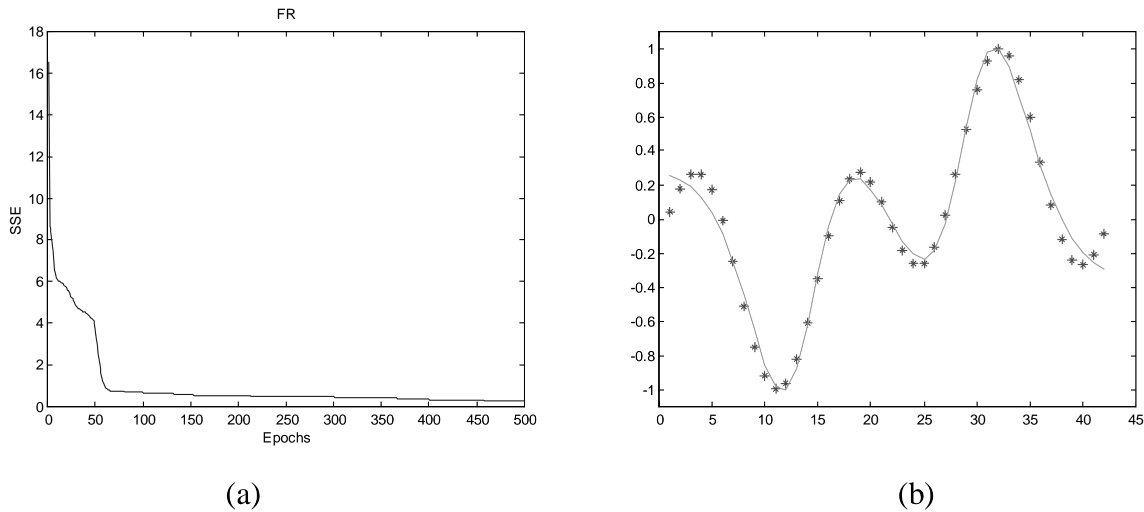
To run this algorithm for the problem presented in Section 4, we used the following command line:

```
>>  [w1, w2, y, sse] = fr(P,T,10,0.1,500,0.0001,0.5);
```

The result given by the net was:

```
>> SSE: 0.244957  Iteration: 500    ||GRAD||: 0.417732     LR: 0.000960
>> Flops total: 75384906      Time: 1.182600e+002
```

Figure 11(a) and (b) presents the error behaviour and the resultant approximation, given by the FR algorithm when applied to the *sin*(*x*)×*cos*(2*x*) problem, respectively.



(a) (b)

Figure 11: (a) Error behaviour. (b) Resultant approximation.

6.8 Scaled Conjugate Gradient method (SCGM)

The second order methods presented up to now use a line search procedure to determine the learning rate. The line search involves a great number of function (or its derivative) evaluations making the process extremely computational intensive. MOLLER [1993] introduces a new variation in the conjugate gradient algorithm (scaled conjugate gradient – SCG), that tries to avoid the line search at each iteration using a Levenberg-Marquardt approach with the goal of scaling the step size α.

If the problems we are dealing with are not quadratic, the matrix **Q** must be approximated by the hessian matrix calculated in the point θ_i, and equation (28) becomes:

$$\alpha_j^* = -\frac{\mathbf{d}_j^T \nabla J(\theta_j)}{\mathbf{d}_j^T \nabla^2 J(\theta_j) \mathbf{d}_j}. \tag{33}$$

The idea used by Moller is estimating the term $\mathbf{s}_j = \nabla^2 J(\theta_j)\mathbf{d}_j$ of the conjugate gradient method using an approximation of the form:

$$\mathbf{s}_j = \nabla^2 J(\theta_j)\mathbf{d}_j \approx \frac{\nabla J(\theta_j + \sigma_j \mathbf{d}_j) - \nabla J(\theta_j)}{\sigma_j}, \qquad 0 < \sigma_j << 1. \tag{34}$$

This approximation tends, in the limit, to the value $\nabla^2 J(\theta_j)\mathbf{d}_j$. Combining this strategy with the conjugate gradient and Levenberg-Marquardt approaches, one can obtain an algorithm directly applicable to the MLP net training. It can be accomplished in the following way:

$$\mathbf{s}_j = \frac{\nabla J(\theta_j + \sigma_j \mathbf{d}_j) - \nabla J(\theta_j)}{\sigma_j} + \lambda_j \mathbf{d}_j. \tag{35}$$

Let δ_j be the denominator of equation (33), then using expression (34), results:

$$\delta_j = \mathbf{d}_j^T \mathbf{s}_j \tag{36}$$

The adjustment parameter λ_j at each iteration and the sign of δ_j determines if the Hessian is definite-positive or not.

The quadratic approximation $J_{quad}(\theta)$, used by the algorithm, is not always a good approximation of $J(\theta)$, once λ_j scales the hessian matrix in an artificial way. A mechanism to increase and decrease λ_j is necessary to determining a good approximation, even when the matrix is definite positive. Define:

$$\begin{aligned}\Delta_j &= \frac{J(\theta_j) - J(\theta_j + \alpha_j \mathbf{d}_j)}{J(\theta_j) - J_{quad}(\alpha_j \mathbf{d}_j)} \\ &= \frac{2\delta_j \left[J(\theta_j) - J(\theta_j + \alpha_j \mathbf{d}_j)\right]}{\mu_j^2}\end{aligned}, \tag{37}$$

where $\mu_j = -\mathbf{d}_j^T \nabla J(\theta_j)$.

The term Δ_j represents a quality measure of the quadratic approximation $J_{quad}(\theta)$ in relation to $J(\theta_j + \alpha_j \mathbf{d}_j)$ in the sense that the closer from 1 Δ_j is, the better the approximation.

6.8.1 Exact calculation of the second order information

The high computational cost associated to the calculus and storage of the hessian matrix $\nabla^2 J(\theta)$ at each iteration can be drastically reduced applying the results obtained by PEARLMUTTER

[1994]. It gives the exact calculation of the second order information at the same time the associated computational cost is the same as the one required by the first order information calculus.

Using a differential operator it is possible to exactly calculate the product of the matrix $\nabla^2 J(\theta)$ by any desired vector, with no need of calculating or storing the matrix $\nabla^2 J(\theta)$. This result is of great value to the conjugate gradient methods, in particular to the Moller's scaled conjugate gradient, where the Hessian $\nabla^2 J(\theta)$ invariably appears multiplied by a vector.

Expanding the gradient vector $\nabla J(\theta)$ around a point $\theta \in \Re^{Nt}$ results:

$$\nabla J(\theta + \Delta\theta) = \nabla J(\theta) + \nabla^2 J(\theta)\Delta\theta + O\left(\|\Delta\theta\|^2\right), \tag{38}$$

where $\Delta\theta$ represents a small perturbation. Choosing $\Delta\theta = a\mathbf{v}$, with a being a positive constant close to zero and $\mathbf{v} \in \Re^{Nt}$ a unit vector, it is possible to calculate $\nabla^2 J(\theta)\mathbf{v}$ as follows:

$$\nabla^2 J(\theta)\mathbf{v} = \frac{1}{a}\left[\nabla J(\theta + a\mathbf{v}) - \nabla J(\theta) + O\left(a^2\right)\right] = \frac{\nabla J(\theta + a\mathbf{v}) - \nabla J(\theta)}{a} + O(a). \tag{39}$$

Taking the limit when $a \rightarrow 0$,

$$\nabla^2 J(\theta)\mathbf{v} = \lim_{a \rightarrow 0} \frac{\nabla J(\theta + a\mathbf{v}) - \nabla J(\theta)}{a} = \frac{\partial}{\partial a} \nabla J(\theta + a\mathbf{v})\Big|_{a=0}. \tag{40}$$

Furthermore, defining a differential operator

$$\Psi_v\{f(\theta)\} = \frac{\partial}{\partial a} \nabla J(\theta + a\mathbf{v})\Big|_{a=0}. \tag{41}$$

It can be applied to all the operations required to obtaining the gradient, producing

$$\Psi_v\{\nabla J(\theta)\} = \nabla^2 J(\theta)\mathbf{v} \quad \text{e} \quad \Psi_v\{\theta\} = \mathbf{v} \tag{42}$$

As a differential operator, $\Psi_v(\theta)$ it follows the usual differentiation rules. Applying these operator to the MLP error backpropagation equations, it is possible to obtaining the exact calculus of the second order information which is directly applicable to the conjugate gradient methods. The modified scaled conjugate gradient source code is presented below.

6.8.2 Matlab® source code

The source code of this algorithm is:

```
function [w1, w2, y, sse] = scgm(P,T,nh,minerr,maxep,val)
%
% SCGM
% Main Program (function)
% MLP net with Backprop training
% (Moller 1993) Scaled Conjugate Gradient with
% Exact calculus of second order information (Pearlmutter, 1994)
% Functions: GOLDSEC, PROCESS, CALCHV
% Off-line Updating
% Author: Leandro Nunes de Castro
```

```
% Unicamp, January 1998
%

%--------------------------------------------------
% Definition and initialisation of the parameters
%--------------------------------------------------
P0 = P;
[np,ni] = size(P0);
[no] = size(T,2);
ep = 0; cm = .7; alfa = 0.001;

w1 = 2.* val.*rand(ni+1,nh) - val;
w2 = 2.* val.*rand(nh+1,no) - val;

%------------------
% Initialisation
%------------------
lambda = 1e-6; lambdab = 0;
delta = 0; deltak = 0; mi = 0;
sse = 1000; sseant = sse; val = 0;
P = [ones(np,1) P0];
Nt = (ni+1)*nh + (nh+1)*no; vgrad = zeros((ni+1)*nh + (nh+1)*no,1);
beta = 0; d = vgrad;
[sse,vgrad,y] = process(w1,w2,P,T);
gdw1 = reshape(d(1:(ni+1)*nh),ni+1,nh);
gdw2 = reshape(d((ni+1)*nh+1:(ni+1)*nh+(nh+1)*no),nh+1,no);
ngrad = norm(vgrad); vgrad = vgrad/ngrad;
d = vgrad + beta*d;
s = calcHv(w1,w2,gdw1,gdw2,T,P,d);
fini = flops;  t0 = clock;
sucesso = 1;

%---------------------------------------------------
% Network training - SCGM
%---------------------------------------------------
while (ep < maxep & sse > minerr)
   ssea = sse; vgrada = vgrad;
   normd2 = d'*d;
   if sucesso == 1,
         s = calcHv(w1,w2,gdw1,gdw2,T,P,d);
         delta = d'*s;
   end;
   delta = delta + (lambda-lambdab)*normd2;
   if delta <= 0,                            % Hessian definite-positive
         lambdab = 2*(lambda-delta/normd2);
         delta = delta + lambda*normd2;
         lambda = lambdab;
   end;
   mi = d'*vgrad;
   alfa = mi/delta;
   w1t = w1 + alfa*gdw1; w2t = w2 + alfa*gdw2;
   [sse,vgrad,y] = process(w1t,w2t,P,T);
   if sse >= ssea
         alfa = goldsec(w1,w2,gdw1,gdw2,T,P,.0001);
         disp('Line Search');
         w1t = w1 + alfa*gdw1; w2t = w2 + alfa*gdw2;
         [sse,vgrad,y] = process(w1t,w2t,P,T);
   end;
   deltak = (2*delta*((ssea-sse)/(mi*mi)));
   w1 = w1t; w2 = w2t;
   if deltak >= 0,
```

```
          lambdab = 0;
          sucesso = 1;
          if rem(ep,Nt) == 0,
                d = vgrad; disp('Restart');
          else
                beta = (vgrad'*(vgrad-vgrada))/(vgrada'*vgrada);
                   beta = max(beta,0);
                d = vgrad + beta*d;
          end;
          if deltak >= 0.75,
                lambda = 0.25*lambda;
          elseif deltak < 0.25,
                lambda = lambda + (delta*(1-deltak)/normd2);
          end;
    else
          lambdab = lambda;
          sucesso = 0;
    end;
    if deltak < 0.25,
          lambda = lambda + (delta*(1-deltak)/normd2);
    end;

    gdw1 = reshape(d(1:(ni+1)*nh),ni+1,nh);
    gdw2 = reshape(d((ni+1)*nh+1:(ni+1)*nh+(nh+1)*no),nh+1,no);
    s = calcHv(w1,w2,gdw1,gdw2,T,P,d);
    ep = ep + 1;
    disp(sprintf('SSE: %f   Iteration: %u     ||GRAD||: %f      LR:
 %f',sse,ep,norm(vgrad),alfa));
    veter(ep) = sse; vetalfa(ep) = alfa;
 end;                              % end of stopping criteria
 fend = flops;  tflops = fend-fini;
 disp(sprintf('Flops total: %d    Time: %d',tflops,etime(clock,t0)));

 % Ploting results
 figure(1); clf; plot (T,'r+'); hold on; plot(y,'g'); drawnow;
 figure(2); semilogy(veter); title('SCGM'); xlabel('Epochs'); ylabel('SSE');
```

6.8.3 Example of application

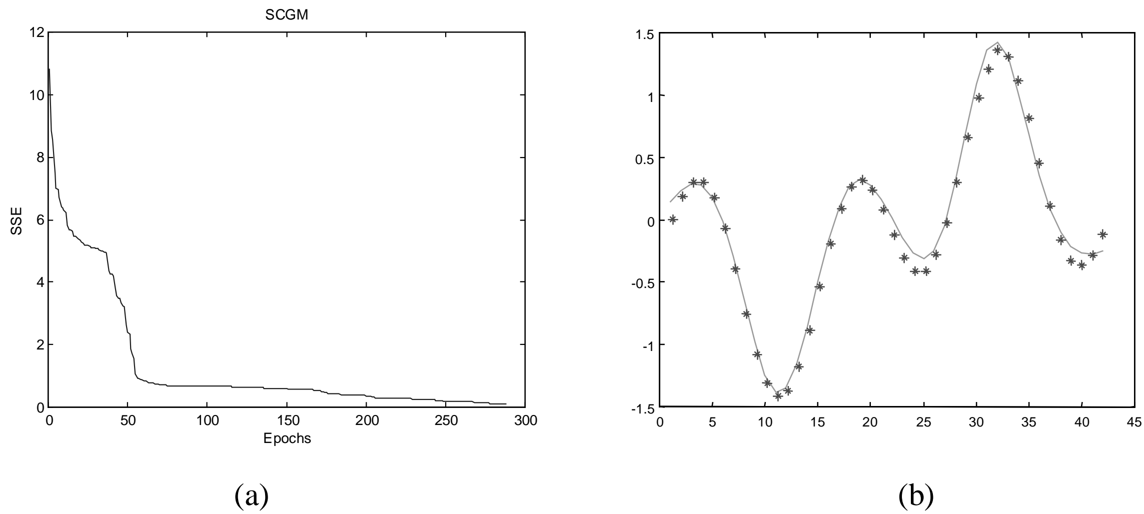
To run this algorithm for the problem presented in Section 4, we used the following command line:

```
>>  [w1, w2, y, sse] = scgm(P,T,10,0.1,500,0.5);
```

The result given by the net was:

```
>> SSE: 0.095196  Iteration: 288    ||GRAD||: 0.713084      LR: 0.010710
>> Flops total: 49210980      Time: 5.737300e+001
```

Figure 12(a) and (b) presents the error behaviour and the resultant approximation, given by the SCGM algorithm when applied to the $sin(x)\times cos(2x)$ problem, respectively.



(a) (b)

Figure 12: (a) Error behaviour. (b) Resultant approximation.

7. Learning rates

The MLP network training can be realised, basically, in two ways: using *batch* or *off-line updating*, or using *local* or *on-line updating*. In the batch updating, the parameters vector is only updated after all training samples are presented to the net. In the local updating procedure, the parameters vector updating is performed immediately after the presentation of each sample vector. Both procedures can have one or multiple values for the step size. A single value of the step size is equivalent to multiplying the adjustment direction by a scalar, though not changing the adjustment direction. Multiple values for the step size is equivalent to multiplying the adjustment direction by a matrix, i.e., it is equivalent to modifying the adjustment direction without scaling it.

As determining the step size value for each component of the parameters vector requires the use of the second order information, we will be restricted to the global methods of determining the step size. We will use the second order information, when it is the case, to directly determine the search direction.

7.1 Inexact line-search

To guarantee a minimising adjustment, unidimensional line-search techniques must be employed, demanding additional computational effort at each iteration.

7.1.1 Matlab® source code

The source code for the inexact line-search algorithm is:

```
function alfa = unidim(gdw1,gdw2,w1,w2,alfa,cm,sse,sseant,T,P);

%
% UNIDIM
% Secondary function
% Inexact line search
% Off-line Updating
% Author: Leandro Nunes de Castro
% Unicamp, January 1998
%

% Line search
[np,ni] = size(P); ni = ni - 1;
[nh,no] = size(w2); nh = nh - 1;
if sse < sseant
   w1a = w1; w2a = w2;
   w1 = w1 + alfa*gdw1;
   w2 = w2 + alfa*gdw2;
   w1 = w1 + cm *(w1-w1a);
   w2 = w2 + cm *(w2-w2a);
   if alfa < .5,                              % Maximum value limit
         alfa = 1.2*alfa;
   end;
else
   aux = 1;
   while (sse >= sseant & aux < 5)
         aux = aux + 1;
         ssep=sse; sse=0;
         alfa = .618*alfa;
         w1a = w1; w2a = w2;
         w1p = w1 + alfa*gdw1;
         w2p = w2 + alfa*gdw2;
         w1p = w1p + cm *(w1p-w1a);
         w2p = w2p + cm *(w2p-w2a);

         %---------------
         % Forward pass
         %---------------
         z = tanh(P*w1p);
         z = [ones(np,1) z];
         y = z*w2p;                           % Linear output
         verr = (T-y); verr = reshape(verr,np*no,1);
         sse = verr'*verr;
   end;
end;
```

7.2 Exact line search – golden section method (GOLDSEC)

The golden section method is a unidimensional line-search procedure that aims at minimising a strictly quasi-convex function in a closed and limited interval. This method performs two function evaluations in the first iteration and then only one in the following iterations [Bazaraa *et. al.*, 1993]. The idea of this strategy is to find an optimum value of α_i over an interval $(0,\overline{\alpha}]$, called *uncertainty interval*. To do so, a continuous reduction of the uncertainty interval is performed at each iteration, until a sufficiently small uncertainty interval is reached. The α_i optimum value is taken as the central point of the resulting interval (or one of its extremes).

This method uses the following methodology:

- given a point θ_i, determine a step $\bar{\alpha}$ that generates a new point $\bar{\theta} = \theta_i + \bar{\alpha}\,\mathbf{d}_i$.

Determining the step α_i involves solving the sub-problem $\min_{\alpha_i \in (0,\bar{\alpha}]} J(\theta_i + \alpha_i\mathbf{d}_i)$, that is a unidimensional search problem. Consider the function J: $\Re^{Nt} \rightarrow \Re$; for $\mathbf{d} \in \Re^{Nt}$ fix, the function g: $\Re \rightarrow \Re$, such as $g(\alpha) = J(\theta_i + \alpha_i\mathbf{d}_i)$ depends solely on the scalar $\alpha_i \in (0,\bar{\alpha}]$.

7.2.1 Matlab® source code

The source code for this method is:

```
function[step] = goldsec(w1,w2,gdw1,gdw2,T,P,dN);
%
% GOLDSEC
% Secondary function
% Unidimensional line search
% Author: Leandro Nunes de Castro
% Unicamp, January 1998
%

%-----------------------------------
% Global definitions
%-----------------------------------
np = size(P,1);
ra = (sqrt(5)-1)/2;
d = [0 1];                           % Initial interval
lb = d(1) + (1 - ra)*(d(2) - d(1));
mi = d(1) + ra*(d(2) - d(1));
mf = []; md = [];

%-----------------------------------
% Function evaluations
%-----------------------------------
w1a = w1; w2a = w2;
w1p = w1 + lb*gdw1;
w2p = w2 + lb*gdw2;
z = tanh(P*w1p);
z = [ones(np,1) z];
y = z*w2p;                           % Linear output
verr = (T-y); verr = reshape(verr,np*no,1);
f(1) = verr'*verr;
w1p = w1 + mi*gdw1;
w2p = w2 + mi*gdw2;
z = tanh(P*w1p);
z = [ones(np,1) z];
y = z*w2p;                           % Linear output
verr = (T-y); verr = reshape(verr,np*no,1);
f(2) = verr'*verr;

%-----------------------------------
% Processing
%-----------------------------------
while abs(d(2) - d(1))/2 > dN,
   if f(1) > f(2),
         d(1) = lb; lb = mi;
         mi = d(1) + ra*(d(2) - d(1));
```

```
        w1p = w1 + mi*gdw1;
        w2p = w2 + mi*gdw2;
        z = tanh(P*w1p);
        z = [ones(np,1) z];
        y = z*w2p;                    % Linear output
        verr = (T-y); verr = reshape(verr,np*no,1);
        f(2) = verr'*verr;
  else,
        d(2) = mi; mi = lb;
        lb = d(1) + (1 - ra)*(d(2) - d(1));
        w1p = w1 + lb*gdw1;
        w2p = w2 + lb*gdw2;
        z = tanh(P*w1p);
        z = [ones(np,1) z];
        y = z*w2p;                    % Linear output
        verr = (T-y); verr = reshape(verr,np*no,1);
        f(1) = verr'*verr;
  end;
  mf = [f(1) f(2); mf];
  md = [d(2) d(1); md];
end;
[y,vind] = min(mf);
if y(1) < y(2),
  ind = vind(1); step = md(ind,1)/2;
else
  ind = vind(2); step = md(ind,2)/2;
end;
```

8. Secondary functions

The secondary functions are the derivative of the activation function (DFAT), the function that runs the net (TESTNN) and the functions CALCHV, that makes the exact calculation of the product Hessian times a vector v and PROCESS, that determines the sum squared error (SSE), the gradient vector and the net output (**y**). The latter two are used in the modified-scaled conjugate gradient (SCGM) algorithm.

8.1 Running the net – (TESTNN)

This function executes the forward pass for the trained neural net.

```
function [sse,y] = testnn(w1,w2,P0,T);
%
% Function TESTNN
% Function that runs the trained network
% Execute the Forward pass and calculates the error
% Author: Leandro Nunes de Castro
% Unicamp, January 1998
%

[np,ni] = size(P0);
[nh,no] = size(w2);
disp(sprintf('Network architecture: [%d,%d,%d]',ni,nh-1,no));
disp(sprintf('Number of training samples: %d',np));
P = [ones(np,1) P0];
z = tanh(P*w1);
z = [ones(np,1) z];
```

```
y = z*w2;                          % Linear output
verr = (T-y); verr = reshape(verr,np*no,1);
sse = verr'*verr;

for i = 1: no,
   figure(i); clf; plot(T(:,i),'r*'); hold on; plot(y(:,i),'g'); drawnow;
   title('* Red: desired   -Green: net output');
   xlabel('Sample'); ylabel('Output');
end;

disp(sprintf('SSE: %f',sse));
```

8.1.1 Example of application

To run this algorithm for the problem presented in Section 4, we used the following command line:

```
>> [sse,y] = testnn(w1,w2,P,T);
```

The result given by the net was:

```
>> Network architecture: [1,10,1]
>> Number of training samples: 42
>> SSE: 0.095196
```

Figure 13 presents the resultant approximation given by the SCGM algorithm when applied to the $sin(x)\times cos(2x)$ problem. The function TESTNN determines the net output, the sum squared error and plots the net outputs versus the desired outputs.

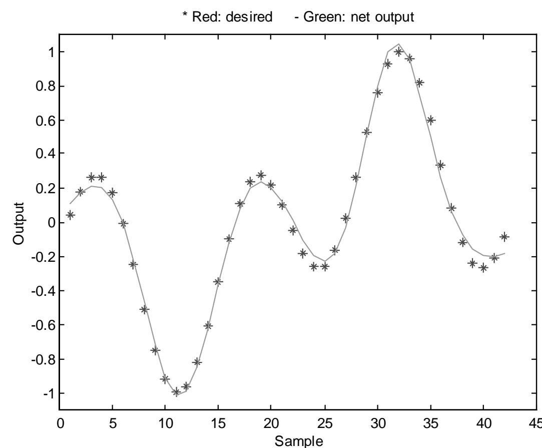


Figure 13: Resultant approximation.

8.2 Calculating the product H.v – (CALCHV)

```
function[Hv] = calcHv(w1,w2,gdw1,gdw2,T,P,d);
%
% Exact calculation of the second order information
% H.v product
% Used by the Moller scaled conjugate gradient (SCGM)
% Author: Leandro Nunes de Castro
% Unicamp, January 1998
%
```

```
%----------------------------------
% Global Definitions
%----------------------------------
[np,ni] = size(P); ni = ni – 1;
[nh,no] = size(w2); nh = nh - 1;

%---------------
% Second order
%---------------
Rx1 = zeros(np,ni+1);
z0 = tanh(P*w1);
df = dfat(P*w1);
Rz = (P*gdw1 + Rx1*w1).*df;
z = [ones(np,1) z0];
Rx2 = [zeros(np,1) Rz];
y = z*w2;                              % Linear output
Ry = z*gdw2 + Rx2*w2;                  % Linear output
erro = T-y; erro2 = erro;
Rerro2 = Ry;
Rw2 = Rx2'*erro2 + z'*Rerro2;
w20 = reshape(w2(2:nh+1,:),nh,no);
gdw20 = reshape(gdw2(2:nh+1,:),nh,no);
erro1 = (erro2*w20').*df;
Rerro1 = (Rerro2*w20' + erro2*gdw20').*df +...
(erro2*w20'.*(-2.*z0.*Rz));
Rw1 = Rx1'*erro1 + P'*Rerro1;

Hv = [reshape(Rw1,(ni+1)*nh,1); reshape(Rw2,(nh+1)*no,1)];
```

8.3 Calculating the SSE, gradient vector and net output – (PROCESS)

```
function [sse,vgrad,y] = process(w1,w2,P,T)
%
% SSE and gradient vector calculus
% Used by the Moller scaled conjugate gradient (SCGM)
% Author: Leandro Nunes de Castro
% Unicamp, January 1998
%

[np,ni] = size(P); ni = ni-1;
[nh,no] = size(w2); nh = nh-1;
z0 = tanh(P*w1);
z = [ones(np,1) z0];
y = z*w2;                              % Linear output
dk = (T-y); gdw2 = z’*dk;              % Linear output
w20 = reshape(w2(2:nh+1,:),nh,no);
dj = (dk*w20’).*(1-z0.^2);
gdw1 = P’*dj;

verr = (T-y); verr = reshape(verr,np*no,1);
sse = verr’*verr;
vgrad = [reshape(gdw1,(ni+1)*nh,1); reshape(gdw2,(nh+1)*no,1)];
```

9. References

[1] **Battiti, R.,** "First- and Second-Order Methods for Learning: Between Steepest Descent and Newton's Method", *Neural Computation*, vol. 4, pp. 141-166, 1992.

[2] **Battiti, R.,** "Learning with First, Second, and no Derivatives: A Case Study in High Energy Phisycs", *Neurocomputing*, NEUCOM 270, vol. 6, pp. 181-206, 1994, URL: ftp://ftp.cis.ohio-state.edu/pub/neuroprose/battiti.neuro-hep.ps.Z.

[3] **Bazaraa, M., Sherali, H. D. & Shetty, C. M.**, "Nonlinear Programming – Theory and Algorithms", 2° edição, John Wiley & Sons Inc., pp. 265-282, 1993.

[4] **Bromberg, M. & Chang, T. S.,** "One Dimensional Global Optimization Using Linear Lower Bounds," In C. A. Floudas & P. M. Pardalos (Eds.), Recent advances in global optimization, pp. 200-220, Princeton University Press, 1992.

[5] **Groot, C. de & Würtz, D.**, "Plain Backpropagation and Advanced Optimization Algorithms: A Comparative Study", *Neurocomputing*, NEUCOM 291, vol. 6, pp.153-161, 1994.

[6] **Haykin, S**. "Neural Networks – A Comprehensive Foundation", 1994.

[7] **Luenberger, D. G.**, "Optimization by Vector Space Methods", New York: John Wiley & Sons, 1969.

[8] **Luenberger, D. G.**, "Linear and Nonlinear Programming", 2° edição, 1989.

[9] **McKeown, J., J., Stella, F. & Hall, G.**, "Some Numerical Aspects of the Training Problem for Feed-Forward Neural Nets", *Neural Networks*, vol. 10, n° 8, pp.1455-1463, 1997.

[10] **Moller, M., F.,** "A Scaled Conjugate Gradient Algorithm for Fast Supervised Learning", *Neural Networks*, vol. 6, pp. 525-533, 1993.

[11] **Pearlmutter, B., A.,** "Fast Exact Calculation by the Hessian", *Neural Computation*, vol. 6, pp. 147-160, 1994, URL: ftp://ftp.cis.ohio-state.edu/pub/neuroprose/pearlmutter.hessian.ps.Z.

[12] **Rumelhart, D., E., McClelland, J. L., and the PDP Research Group**. "Parallel Distributed Processing: Exploration in the Microstructure of Cognition, vol. 1. MIT Press, Cambridge, Massachussetts, 1986.

[13] **Shepherd, A., J.**, "Second-Order Methods for Neural Networks – Fast and reliable Methods for Multi-Layer Perceptrons", Springer, 1997.

[14] **Van Der Smagt, P. P**, "Minimization Methods for Training Feedforward Neural networks," Neural Networks, vol 1, n° 7, 1994, URL: http://www.op.dlr.de/~smagt/papers/SmaTB92.ps.gz

[15] **Von Zuben, F. J**, "Modelos Paramétricos e Não-Paramétricos de Redes neurais Artificiais e Aplicações", Tese de Doutorado, Faculdade de Engenharia Elétrica, Unicamp, 1996.