

# Neural networks : low-memory training and regularization techniques

GARCIA M.<sup>†</sup> , JAN S.<sup>‡</sup> , MASMOUDI M.<sup>‡</sup>

<sup>†</sup>CERFACS, 42 Avenue Gaspard Coriolis, 31057 Toulouse Cedex 01. France.

<sup>‡</sup>MIP Laboratory, Paul Sabatier University, 31062 Toulouse cedex 9. France  
milagros.garcia@cerfacs.fr ; {jan,masmoudi}@mip.ups-tlse.fr

CERFACS Technical Report TR/PA/07/25

## Abstract

With the intention of using easy-to-compute and differentiable surrogate models for the optimization of computationally expensive objective functions, we have studied the artificial neural network methodology. We describe in this paper a low-memory Levenberg-Marquardt algorithm for the supervised training of artificial neural networks. It is based on the use of the forward and reverse modes of the algorithmic differentiation. We explain also how we have combined three different strategies of regularization to build neural networks that generalize well.

**Keywords:** neural network, algorithmic differentiation, training, regularization.

## Motivation and notations

In the real life there exists a lot of optimization problems in which the objective function is very expensive to compute (for example, computational fluid dynamics or crash testing simulation). A way to treat problems of this type is to build an easy-to-compute model of the objective function and to optimize it. In order to allow the use of efficient optimization processes, we were also interested in differentiable models.

Among all the metamodeling techniques (response surfaces, artificial neural networks, polynomial regression, kriging, multivariate adaptive regression splines, radial basis functions, ...) which are listed in Jin et al. (2001), we have chosen the neural network methodology. It provides us together with the two required qualities : ease of computation and differentiability.

After a brief recall about the artificial neural network techniques in section 1, we describe in section 2 how we have used the algorithmic differentiation tools to develop a low-memory Levenberg-Marquardt training algorithm. Then we explain in section 3 the different regularization techniques we have implemented to obtain neural networks that do not overfit the given data. Finally, we give some promising numerical results in section 4, including some comparisons with the `trainbr` function of the Neural Network Toolbox of MATLAB.

We assume now that we want to approximate a continuous function  $f$  defined from  $\mathbb{R}^{n_I}$  ( $n_I$  input or control variables) into  $\mathbb{R}^{n_O}$ . That means that we try to approximate together  $n_O$  continuous real-valued functions.

# 1 A brief recall about neural networks

## 1.1 Universal approximators

Since it has already been proven in (Cybenko, 1988; Cybenko, 1989; Hornik et al., 1989) that three-layers neural networks are universal approximators for the continuous functions, we have chosen these interesting tools to build our easy-to-compute and differentiable approximation models.

Let us now describe how a three-layers neural network works. Then we will be able to highlight its main qualities and to describe what is called the supervised training phase.

The first layer of the neural network is called the input layer. It contains  $n_I + 1$  cells corresponding to the  $n_I$  control variables involved in the phenomenon to be approximated and a cell which is called the bias. The third layer is the output layer and it contains  $n_O$  cells. Finally, the intermediate layer is also called the hidden layer. The optimal number of hidden cells we have to choose to build a good approximation is not easy to find. Unfortunately, the size of the hidden layer may increase a lot with the complexity of the phenomenon to be approximated. Let us denote by  $n_H$  this number of hidden cells.

Each cell  $c_j$  of one layer is connected to each cell  $c_i$  of the following layer and each of these links is associated to a weight  $w_{ij}$ .

Let us denote by  $x_i^l$  the state of the cell  $c_i$  of the layer  $l$ . In our work, the state of the cell  $j$  of the second layer is given by

$$x_j^2 = f \left( \left( \sum_{k=1}^{n_I} w_{jk}^1 x_k^1 \right) + w_{j,n_I+1}^1 \right) \quad (1)$$

where  $f$  is approximately (cf. section 3) the activation function defined for all  $z$  in  $\mathbb{R}$  by

$$f(z) = \frac{1}{1 + e^{-z/10}}. \quad (2)$$

The state of the cell  $i$  of the output layer is given by

$$x_i^3 = \sum_{j=1}^{n_H} w_{ij}^2 x_j^2. \quad (3)$$

The interpretation of the preceding formulæ is the following. First (formula (1)), basis functions adapted to the problem are built between the input layer and the hidden one. Then (formula (3)), a linear approximation is performed in this basis.

What is interesting with neural networks is that the basis functions are adapted to each application. They are not defined a priori. This is a great advantage with respect to more classical approximation methods like polynomials, splines or wavelets.

The formulæ (1), (2) and (3) show that a model based on such a neural network is easy-to-compute. It is also easy-to-differentiate with respect to the input variables  $x_i$ ,  $i = 1, \dots, n_I$  or with respect to the weights  $w_{ij}^l$ ,  $l = 1, 2$ .

Let us introduce some notations:

- $X^1$  is the column vector formed with the  $n_I$  components  $x_k^1$  and  $\underline{X^1}$  is the column vector  $((X^1)^T, 1)^T$ ,
- $W^1$  is the  $n_H \times (n_I + 1)$  matrix formed with the  $w_{jk}^1$ ,
- and  $W^2$  is the  $n_O \times n_H$  matrix formed with the  $w_{ij}^2$ ,
- $W = (W^1, W^2) \in \mathbb{R}^{n_H \times (n_I + 1)} \times \mathbb{R}^{n_O \times n_H}$ ,
- $F$  is the function defined for all  $X^2 = (x_1^2, \dots, x_{n_H}^2)^T$  in  $\mathbb{R}^{n_H}$  by  $F(X^2) = (f(x_1^2), \dots, f(x_{n_H}^2))^T$ ,
- $X^3$  is the column vector formed with the  $n_O$  components  $x_i^3$ .

With these notations, the response of the neural network to the configuration  $X^1$  with the weights  $W$  is simply given by

$$X^3 = R(W, X^1) = W^2 F(W^1 \underline{X^1}). \quad (4)$$

## 1.2 The training of neural networks

We will describe here how a neural network such as that described in the preceding section may become a good approximation model of a physical phenomenon.

We assume that we have a set of  $n_P$  patterns

$$\Omega = \{(X_i, Y_i), i = 1, \dots, n_P\}.$$

In this set, each  $X_i$  is a vector with  $n_I$  components that corresponds to one configuration of the control variables of the phenomenon to be approximated. The vector of  $n_O$  components  $Y_i$  is the phenomenon response to the configuration  $X_i$ :  $Y_i = f(X_i)$ .

Let us now denote by  $G_\Omega$  the function defined for all weights  $W \in \mathbb{R}^{n_H \times (n_I + 1)} \times \mathbb{R}^{n_O \times n_H}$  by

$$G_\Omega(W) = \left( R(W, X_1) - Y_1, \dots, R(W, X_{n_P}) - Y_{n_P} \right) \in \mathbb{R}^{n_O \times n_P}.$$

In order to make the neural network become a good approximation model of the phenomenon we are interested in, we have to adjust the weights  $W$  in the following sense: we look for the weights  $\hat{W}$  that solves the following optimization problem:

$$\min_W h_\Omega(W) := \frac{1}{2n_O n_P} \|G_\Omega(W)\|_F^2. \quad (5)$$

This is called the training phase.

This optimization problem is classically solved using the Levenberg-Marquardt algorithm (described for example in (Kelley, 1999; Ranganathan, 2004)). This method is widely used, but it is well-known that it requires the storage of potentially large matrices. We will explain in the section 2 how we have got around this problem.

After the training phase, the error  $h_\Omega$  is very small, but for a new pattern  $(X_{n_P+1}, Y_{n_P+1})$  the neural network response  $R(\hat{W}, X_{n_P+1})$  may be quite far from  $Y_{n_P+1}$ . This phenomenon is called the overfitting. The network has memorized the training patterns, but it has not generalized to other situations. We will detail our strategy to improve the generalization of neural networks in the section 3.

## 2 A low memory training method

In order to write a self-contained paper, we will recall what is the Levenberg-Marquardt algorithm (LMA). Then we will make the link between the LMA and the trust region framework, and finally we will explain how we have used the algorithmic differentiation tools to obtain a significant memory reduction.

The Levenberg-Marquardt algorithm is particularly adapted to the nonlinear least-squares optimization problems. For sake of simplicity, we will consider the generic problem of this type:

$$\min h(x) := \frac{1}{2} \sum_{k=1}^m \left( r_k(x) \right)^2 \quad (6)$$

where  $x \in \mathbb{R}^n$  and each  $r_i$  is a function from  $\mathbb{R}^n$  to  $\mathbb{R}$ .

*Remark 2.1* problem (5) may be put into this formalism with the following changes:

- $m = n_P n_O$ ,
- the matrix  $W$  is transformed into a vector  $x$ ,
- and  $\forall i = 1, \dots, n_P$  and  $\forall j = 1, \dots, n_O$ ,  $r_{(i-1)n_O+j}(x) = \left( R(W, X_i) - Y_i \right)_j$ , where  $\left( Y_i \right)_j$  is the  $j$ -th component of  $Y_i$ .

### 2.1 A combination of Gauss-Newton and steepest descent iterations

The most intuitive method to solve an unconstrained optimization problem is to follow what is called the steepest descent direction, which is nothing but the opposite of the gradient direction. With this technique, we go from the current point  $x$  to the next one  $x_+$  using the update rule

$$x_+ = x - \mu \nabla h(x), \quad (7)$$

where  $\mu > 0$  represents the length of the step made in the steepest descent direction. Unfortunately, the rate of convergence of this simple method is often quite bad. This drawback may be overcome if the curvature of the problem is somehow taken into account.

The most popular method that makes use of the curvature is the Newton method whose rate of convergence is quadratic near the solution. This algorithm comes from the following approximation of  $\nabla h$  at the point  $x_+$ :

$$\nabla h(x) + \nabla^2 h(x)(x_+ - x).$$

The Newton direction  $d_N$  associated to problem (6) at the current point  $x$  is thus given by the solution of the linear system

$$\nabla^2 h(x) d_N = -\nabla h(x) \quad (8)$$

with

$$\begin{aligned} \nabla h(x) &= \sum_{k=1}^m r_k(x) \nabla r_k(x) = J(x)^T r(x), \\ \nabla^2 h(x) &= J(x)^T J(x) + \sum_{k=1}^m r_k(x) \nabla^2 r_k(x), \end{aligned}$$

where  $r(x) = \left( r_1(x), \dots, r_m(x) \right)^T$  is called the residual vector and  $J(x) = \left( \frac{\partial r_i}{\partial x_j} \right)_{\substack{1 \leq i \leq m \\ 1 \leq j \leq n}}$  is its Jacobian matrix.

When the residuals  $r_k$  are quite linear functions (in which case  $\nabla^2 r_k(x)$  is small) or when they are small, the Hessian  $\nabla^2 h(x)$  may be approximated at a negligible cost by its first term  $J(x)^T J(x)$ . The Gauss-Newton direction  $d_{GN}$  is precisely based on this approximation: it is the solution of the linear system

$$J(x)^T J(x) d_{GN} = -J(x)^T r(x). \quad (9)$$

Levenberg (1944) and Marquardt (1963) proposed to combine the steepest descent method and the Gauss-Newton one to take advantage of both. The Levenberg-Marquardt direction is thus given by the solution of the following linear system:

$$\left( J(x)^T J(x) + \alpha I \right) d_{LM} = -J(x)^T r(x). \quad (10)$$

Given an initial point  $x_0 \in \mathbb{R}^n$ , an initial positive real  $\alpha^0$  and  $k = 0$ , the Levenberg-Marquardt algorithm is thus

1. compute  $d_k$  solution of  $\left( J(x_k)^T J(x_k) + \alpha_k I \right) d_k = -J(x_k)^T r(x_k)$ ;
2. if  $h(x_k + d_k) < h(x_k)$ , then this implies that the quadratic model of  $h$  is quite good and  $\alpha$  is decreased to reduce the influence of the steepest descent; goto step 4;
3. if  $h(x_k + d_k) \geq h(x_k)$ , then the influence of the steepest descent is increased by increasing  $\alpha$  ; goto step 1;
4.  $x_{k+1} = x_k + d_k$ , increase  $k$  by one and goto step 1.

The Levenberg-Marquardt algorithm may also be seen as a trust region method.

## 2.2 A trust region algorithm

In the trust-region framework widely studied in Conn et al. (2000), an approximation model  $m_k$  of  $h$  is defined in the vicinity of the current point  $x_k$ . The neighbourhood of  $x_k$  where the model can be trusted is measured by a parameter  $\Delta$ , which is called the trust-region parameter. Then the subproblem to be solved to find the step is

$$\begin{aligned} \min m_k(d). \\ \|d\| \leq \Delta \end{aligned} \quad (11)$$

Let us consider here the following model:

$$m_k(d) = h(x_k) + \nabla h(x_k)^T d + \frac{1}{2} d^T J(x_k)^T J(x_k) d. \quad (12)$$

Then the solution of (11) is given by the following theorem :

**Theorem 2.2**  $\hat{d}$  is a solution of (11) with  $m_k$  defined by (12) if and only if  $\|\hat{d}\| \leq \Delta$  and there exists a scalar  $\alpha \geq 0$  such that  $J(x_K)^T J(x_k) + \alpha I$  is positive semidefinite and

$$\left( J(x_K)^T J(x_k) + \alpha I \right) \hat{d} = -\nabla h(x_k), \quad (13)$$

$$\alpha(\Delta - \|\hat{d}\|) = 0. \quad (14)$$

It can be seen that (13) is exactly the same as (10) and that (14) basically states that  $\alpha$  may be zero when  $\|\hat{d}\| < \Delta$ . This means that  $\alpha$  may be decreased when  $\hat{d}$  is inside the trust-region, but it must be increased when  $\hat{d}$  goes outside the trust-region. We obtain thus here the same parameter update rule as in the preceding subsection. But in the trust region framework, there exist some criteria allowing a reliable update of the  $\Delta$  parameter. These criteria are based on a reduction coefficient  $\rho$  that measures the real progress made towards the solution. We have thus chosen to implement the trust-region form of the Levenberg-Marquardt algorithm.

As it can be seen on (10), the Levenberg-Marquardt algorithm usually requires the computation of the inverse of  $J(x)^T J(x) + \alpha I$  whose size may be quite large in some cases. We will detail now how we have overcome this drawback.

## 2.3 Memory reduction

We describe here an efficient way, at least in terms of storage, to solve linear systems of the form:

$$\left( J(x)^T J(x) + \alpha I \right) d = -J(x)^T r(x). \quad (15)$$

First, we solve this linear system using the conjugate gradient method which requires only matrix-vector products. Then the computation of the left-hand side of (15) is made in two steps.

1. We compute first  $z = J(x)d$ . This quantity may be rewritten

$$J(x)d = \lim_{\varepsilon \rightarrow 0} \frac{r(x + \varepsilon d) - r(x)}{\varepsilon} = \left. \frac{\partial r(x + \varepsilon d)}{\partial \varepsilon} \right|_{\varepsilon=0}.$$

In this form,  $J(x)d$  corresponds to the differentiation of a vector-valued function  $r$  with respect to a single parameter  $\varepsilon$ . This can be done very efficiently using the forward mode of the algorithmic differentiation.

2. Then we have to compute  $J(x)^T z$  that can be rewritten

$$J(x)^T z = \sum_{i=1}^m \nabla r_i(x) z_i = \nabla \left( \sum_{i=1}^m r_i(x) z_i \right) = \nabla (r(x)^T z).$$

In this form,  $J(x)^T z$  corresponds to the differentiation of a scalar function with respect to several parameters and the reverse mode of the algorithmic differentiation is particularly efficient in this case.

For readers non familiar with the algorithmic differentiation, we advise the following references: Griewank, 2000; Rall and Corliss, 1996; Gilbert et al., 1991.

Of course, the computation of the right-hand side of (15) is realized in the same manner as in the second step of the computation of the left-hand side.

We have used this method to train neural networks on a set  $\Omega$  of patterns. That means that we have used the Levenberg-Marquardt algorithm (in its trust-region form) to solve the problem (5). But in some cases, as we have already said, the neural network does not give a correct approximation for data that were not in the training set  $\Omega$ . We describe in the next section the various regularization methods we have used to obtain neural networks that generalize well.

### 3 Combination of several regularization techniques

We detail now our strategy to improve the generalization of neural networks.

The set of patterns (that are all normalized between 0 and 1)  $\Omega$  is divided into three parts  $\Omega_T$ ,  $\Omega_G$  and  $\Omega_V$ , where  $T$  stands for **T**raining,  $G$  for **G**eneralization and  $V$  for **V**alidation. The weights  $W$  are initialized with small random values between  $-0.1$  and  $0.1$ .

We proceed then in three stages.

#### 1. First training phase

We look for the size of the hidden layer  $n_H$  that allows us to make a good training of the neural network on the patterns from  $\Omega_T$ . This means that  $n_H$  is increased and

$$h_{\Omega_T}(\hat{W}_0) = \min_W h_{\Omega_T}(W) \quad (16)$$

solved until  $h_{\Omega_T}(\hat{W}_0)$  become smaller than a threshold precision `tol`.

#### 2. Regularization phase

- (a) if the error  $h_{\Omega_G}(\hat{W}_0)$  is also smaller than `tol`, then we set  $\hat{\beta} = 0$  and we go to the step 3 ;
- (b) else we look for what we have called a Tikhonov regularization parameter  $\beta$  in order to enforce the weights involved in the definition of the basis functions (and only them) to stay small. By this way, we try to define smooth and stretched basis functions that will prevent the neural network to oscillate too much. This is realized in the following manner. For several increasing values of  $\beta > 0$ , we look for  $\hat{W}_\beta$  that solves the optimization problem

$$\min_W h_{\Omega_T}(W) + \beta \frac{1}{2(n_I + 1)n_H} \|W^1\|_F^2. \quad (17)$$

This problem is solved with the Levenberg-Marquardt algorithm described in section 2, using the same initial weights as for the first training phase.

We denote by  $\hat{\beta}$  the value of  $\beta$  for which  $h_{\Omega_G}(\hat{W}_\beta)$  is the smallest. Of course,  $\hat{\beta}$  may be zero if the smallest value for  $h_{\Omega_G}$  is obtained in step 1.

#### 3. Final training phase

We perform a new training phase on the set  $\Omega_T \cup \Omega_G$  using the regularization parameter obtained at the step 2. This means that we solve

$$\min_W h_{\Omega_T \cup \Omega_G}(W) + \hat{\beta} \frac{1}{2(n_I + 1)n_H} \|W^1\|_F^2 \quad (18)$$

with the weights  $\hat{W}_{\hat{\beta}}$  as initial weights.

*Remark 3.1* In the cases where  $\beta \neq 0$ , since the residuals near the solution are not small, the performance of the Levenberg-Marquardt algorithm on problem (17) and/or (18) is quite poor.

The numerical examples we give in the next section seem quite promising. We think they are the results of the use of several regularization techniques during the process described previously:

- the use of a Gauss-Newton derived method that guarantee that the change in the weights is minimal;
- the selection of a small number of hidden cells;
- the use of a Tikhonov type regularization technique in order to build smooth and stretched basis functions;
- the use of a slightly modified sigmoid function that enforces the weights between the input and the hidden layers to stay small.

## 4 Numerical experiments

The ideas developed in the preceding sections lead us to develop a software in Matlab that we have named GAP for Global Assimilation Process.

In this section, we will detail the results of several numerical experiments. We consider here phenomena with only one output ( $n_O = 1$ ), but there is no major difficulty in applying this tool to vector-valued functions. We have compared the comportment of GAP with the `trainbr` function of the Neural Network Toolbox of MATLAB in two types of situation. First, we have tried to build a good approximation of various basic real-valued functions. In a second step, we have tested the robustness of these methods to noise, using the Rastrigin function and two different sets of data.

Before the description of the numerical experiments realized, let us describe here briefly what is used in the `trainbr` function of the Neural Network Toolbox of Matlab. As our software GAP, `trainbr` is based on the Levenberg-Marquardt algorithm. There exists also a technique for reducing the memory, but it relies on the calculation of the approximate Hessian  $J^T J$  by summing a series of submatrices of  $J$ . In order to obtain neural networks that generalize well, `trainbr` uses the bayesian regularization described in (MacKay, 1992; Foresee and Hagan, 1997) which minimizes a linear combination of squared error and all the weights.

	Test number	Test name	$n_I$	nonlinearity	$n_T$	$n_G$	$n_V$
Without noise	1	Carredec2	2	low	12	3	100
	2	Trigo2	2	high	12	3	100
	3	Carredec6	6	low	50	10	100
With noise	4	Noisy sinus	1	high	25	12	4
	5	Rastrigin2	2	low	12	3	100
	6	Crash	7	high	85	10	5

Table 1: Description of the test cases

## 4.1 Description of the test cases

Let us now describe the numerical experiments we have realized in the table 1. In this table,  $n_T$  (respectively  $n_G$  and  $n_V$ ) represents the size of the set  $\Omega_T$  (respectively  $\Omega_G$  and  $\Omega_V$ ).

In the first three tests, we want to build a good model of the first two functions described in the table 2. The fourth test case consists in the approximation of the sinus function given in table 2 using 41 noisy measurements. In the fifth case, we want to build a very coarse model 2 of the Rastrigin function which is given in table 2. This approach is a first step towards the use of neural networks as surrogate models for the optimization of noisy and/or expensive cost functions. Finally, in the test case number 6, we want to design a model of crash of a vehicule on a distortable barrier. We have the results for a total of 100 expensive crash-tests for which we have the measurements of seven various parameters such as the incidence speed and the seating of the vehicule. For the same experiments, we also have the strength imposed by the wall on the back of the vehicle front structure after the impact. This test case has been studied in the context of the MONASTIR consortium that consists of several societies like CADOE, MECALOG, MICHELIN, PSA, RENAULT, SNCF and the MIP laboratory from the Paul Sabatier university.

Name	Function	Design domain
Carredec	$\sum_{i=1}^{n_I} \left( x_i - 2 + 4 \frac{i-1}{n_I-1} \right)^2$	$-3 \leq x_i \leq 3$
Trigo	$n_I - \sum_{i=1}^{n_I} \sin(x_i)$	$-3 \leq x_i \leq 3$
Rastrigin	$\sum_{i=1}^{n_I} \left( x_i^2 - \frac{1}{2} \cos(18x_i) \right)$	$-2 \leq x_i \leq 2$
Sinus	$\sum_{i=1}^{n_I} \sin(2\pi x_i)$	$-1 \leq x_i \leq 1$

Table 2: Description of the functions used in some test cases

## 4.2 Accuracy measures

For comparison purposes, we use the following accuracy measures given in Mullur and Messac, 2005:

- the standard Root Mean Squared Error (RMSE) defined by

$$RMSE = \sqrt{\frac{1}{K} \sum_{k=1}^K (m(X_k) - Y_k)^2},$$

- the Normalized Root Mean Squared Error (NRMSE) defined by

$$NRMSE = 100 \sqrt{\frac{\sum_{k=1}^K (m(X_k) - Y_k)^2}{\sum_{k=1}^K Y_k^2}}.$$

where  $X_k$  belongs to a set of  $K$  data that we use to test the quality of the models,  $Y_k$  is the response to this configuration from the phenomenon we want to approximate and  $m$  is the model used (Neural Network Toolbox of MATLAB, GAP, ...). The NRMSE error measure represents the percentage error in the model. A value of zero indicates a perfect fit, while a high value indicates a poor fit.

In the test cases 1 to 3 and 5, we use  $K = 1000$  additional data that are randomly generated. In the cases 4 and 6, we have used the  $K = n_V$  patterns from the validation set  $\Omega_V$ .

We also use the R-value that we obtain by a linear regression between each element of the network response and the corresponding target for all the data contained in  $\Omega$ .

### 4.3 Description of the results

We propose a summary of our comparison in the table 3. Each entry in the table represents the mean value after 10 different trials, where different random initial weights are used in each trial. For the test cases 1, 2, 3 and 5, we used also different random given patterns in each trial. When a cell contains three stars, this means that the model built was flat for each of the ten trials.

Test case number	NN Toolbox of Matlab <code>trainbr</code> function				GAP				
	$n_H$	RMSE	NRMSE	R	$n_H$	RMSE	NRMSE	R	$\hat{\beta}$
1	5	0.19	1.54	1	3.3	0.15	0.84	1	0
2	20	***	***	***	5.6	0.086	3.84	0.997	2.37e-6
3	10	1.97	9.44	0.980	7.7	2.22	6.9	0.975	7.07e-5
4	7	0.095	16.89	0.994	15.2	0.14	25.8	0.987	1.24e-5
5	20	***	***	***	11.9	0.81	24.98	0.918	4.36e-5
6	20	14377.58	3.43	0.644	13.6	24833	5.92	0.909	2.47e-5

Table 3: Description of the numerical results

We can make the following comments about these results:

- Except for the fourth case, GAP needs less hidden neurons than the `trainbr` function of the NN Toolbox of Matlab;

- GAP has given satisfactory results (R-values are all greater than 0.9) in each of our numerical experiments, whereas Matlab has failed in the test cases 2 and 5 and was not good in the test case 6 (R-value less than 0.7);
- Although GAP has given a greater value for the NRMSE and a lower R-value for the noisy sinus case than MATLAB, the visual results (figure 1) are quite similar. The software GAP seems to be quite robust when it is applied to noisy data. This is quite clear if we compare the promising results of GAP to the failure of MATLAB for the CRASH example.

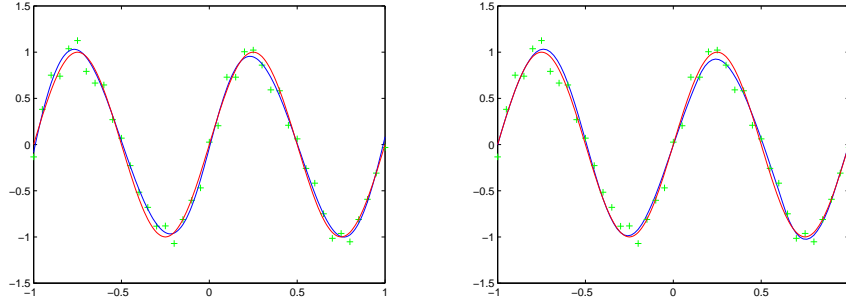


Figure 1: Comparison of Matlab (left) and GAP (right) for the test case 4. The green signs represent the targets (noisy sinus), the blue curve is the simulation of the neural network and the red curve is the sinus function (described in table 2) we wanted to approximate.

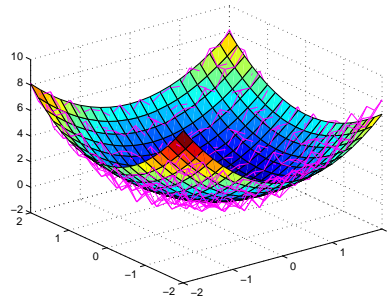


Figure 2: A coarse model (multicolor) of the Rastrigin function (in pink) obtained with GAP.

## 5 Conclusion

In this paper, we have presented a low-memory Levenberg-Marquardt algorithm to train three-layers neural networks. We have also explained the various regularization techniques we have used to obtain neural networks that generalize well: the use of a Gauss-Newton derived method, a minimal number of hidden neurons, and a Tikhonov type regularization on the weights between the input layer and the hidden one. We have also realized some numerical experiments that seem to be quite promising.

We have several ideas to improve and value this study. First, we have to find a better way to choose the Tikhonov regularization coefficient  $\beta$ . Then we have to perform more tests in higher dimensions and to quantify the real impact of the use of the modified sigmoid function. Finally, we want to use this type of neural network model to find at low cost optimal points of computationally expensive objective functions.

## References

Conn A. R., Gould N. I. M., Toint Ph. L. Trust-region methods. Society for Industrial and Applied Mathematics (SIAM): Philadelphia; 2000.

Cybenko G. Continuous valued neural networks with two hidden layers are sufficient. Technical report, Department of Computer Science, Tufts University, Medford, MA, 1988.

Cybenko G. Approximation by superpositions of sigmoidal function. Mathematics of Control, Signals and Systems, 1989; 2;303-314.

Foresee F. D. and Hagan M. T. Gauss-newton approximation to bayesian regularization. In Proceedings of the 1997 International Joint Conference on Neural Networks.

Gilbert J. C., Le Vey G., Masse J. La différentiation automatique de fonctions représentées par des programmes. Technical report, INRIA, 1991.

Griewank A. Evaluating derivatives. Society for Industrial and Applied Mathematics (SIAM): Philadelphia; 2000.

Hornik K., Stinchcombe M., White H. Multilayer feedforward networks are universal approximators. Neural Networks, 1989; 2; 359-366.

Jin R., Chen W., Simpson T. W. Comparative studies of metamodeling techniques under multiple modeling criteria. Structural and Multidisciplinary Optimization, 2001; 23(1); 1-13.

Kelley C. T. Iterative methods for optimization. Society for Industrial and Applied Mathematics (SIAM): Philadelphia; 1999.

Levenberg K. A method for the solution of certain problems in least squares. Quarterly of Applied Mathematics, 1944; 2; 164-168.

MacKay D. J. C. Bayesian interpolation. Neural Computation, 1992; 4(3); 415-447.

Marquardt D. An algorithm for least-squares estimation of nonlinear parameters. SIAM Journal on Applied Mathematics, 1963; 11; 431-441.

Mullur A. A. and Messac A. Extended radial basis functions: More flexible and effective metamodeling. *AIAA Journal*, 2005; 43(6); 1306-1315.

Rall L. B. and Corliss G. F. An introduction to automatic differentiation. In *Computational differentiation* (Santa Fe, NM, 1996), SIAM, Philadelphia, 1-18.

Ranganathan A. <http://www.cc.gatech.edu/~people/home/ananth/lmtut.pdf>. The Levenberg-Marquardt algorithm, 2004.