**Revista Colombiana de Estadística**
July 2018, Volume 41, Issue 2, pp. 173 to 189
DOI: http://dx.doi.org/10.15446/rce.v41n2.55250

# Artificial Neuronal Networks: A Bayesian Approach Using Parallel Computing

**Redes neuronales regularizadas: un enfoque bayesiano usando cómputo paralelo**

Eduardo Guzmán[1,a], Mario Vázquez[2,b], David del Valle[1,c], Paulino Pérez-Rodríguez[1,d]

[1]Estadística, Socio Economía Estadística e Informática, Colegio de Postgraduados, Texcoco, México

[2]Matemáticas y Cómputo, Departamento de Irrigación, Universidad Autónoma Chapingo, Texcoco, México

**Abstract**

An Artificial Neural Network (ANN) is a learning paradigm and automatic processing inspired in the biological behavior of neurons and the brain structure. The brain is a complex system; its basic processing unit are the neurons, which are distributed massively in the brain sharing multiple connections between them. The ANNs try to emulate some characteristics of humans, and can be thought as intelligent systems that perform some tasks in a different way that actual computer does. The ANNs can be used to perform complex activities, for example: pattern recognition and classification, weather prediction, genetic values prediction, etc. The algorithms used to train the ANN, are in general complex, so therefore there is a need to have alternatives which lead to a significant reduction of times employed to train an ANN. In this work, we present an algorithm based in the strategy "divide and conquer" which allows to train an ANN with a single hidden layer. Part of the sub problems of the general algorithm used for training are solved by using parallel computing techniques, which allows to improve the performance of the resulting application. The proposed algorithm was implemented using the C++ programming language, and the libraries Open MPI and ScaLAPACK. We present some application examples and we asses the application performance. The results shown that it is possible to reduce significantly the time necessary to execute the program that implements the algorithm to train the ANN.

***Key words***: Empirical Bayes; Nonlinear models; Parallel processing.

[a]MSc. E-mail: eduguz.h@gmail.com
[b]PhD. E-mail: mavp52@hotmail.com
[c]PhD. E-mail: dhvallep@gmail.com
[d]PhD. E-mail: perpdgo@gmail.com

**Resumen**

Una Red Neuronal Artificial (RNA) es un paradigma de aprendizaje y procesamiento automático inspirado en el comportamiento biológico de las neuronas y en la estructura del cerebro. El cerebro es un sistema altamente complejo; su unidad básica de procesamiento son las neuronas, las cuales se encuentra distribuidas de forma masiva compartiendo múltiples conexiones entre ellas. Las RNAs intentan emular ciertas características propias de los humanos, pueden ser vistas como un sistema inteligente que lleva a cabo tareas de manera distinta a como lo hacen las computadoras actuales. Las RNAs pueden emplearse para realizar actividades complejas, por ejemplo: reconocimiento y clasificación de patrones, predicción del clima, predicción de valores genéticos, etc. Los algoritmos utilizados para entrenar las redes, son en general complejos, por lo cual surge la necesidad de contar con alternativas que permitan reducir de manera significativa el tiempo necesario para entrenar una red. En este trabajo se presenta una propuesta de algoritmos basados en la estrategia "divide y conquista" que permiten entrenar las RNAs de una sola capa oculta. Parte de los sub problemas del algoritmo general de entrenamiento se resuelven utilizando técnicas de cómputo paralelo, lo que permite mejorar el desempeño de la aplicación resultante. El algoritmo propuesto fue implementado utilizando el lenguaje de programación C++, así como las librerías Open MPI y ScaLAPACK. Se presentan algunos ejemplos de aplicación y se evalúa el desempeño del programa resultante. Los resultados obtenidos muestran que es posible reducir de manera significativa los tiempos necesarios para ejecutar el programa que implementa el algoritmo para el ajuste de la RNA.

***Palabras clave*:** Bayes empírico; modelos no lineales; procesamiento en paralelo.

## 1. Introduction

An Artificial Neural Network (ANN) is a learning paradigm and automatic processing inspired in the biological behavior of neurons and the brain structure. The brain is a complex system; its basic processing unit are the neurons, which are distributed massively in the brain sharing multiple connections between them. From the statistical point of view the ANN are non linear regression models that are useful for prediction. ANN have been used to perform complex tasks in many fields, for example classification and pattern recognition, weather prediction, prediction of genetic values (Pérez-Rodríguez, Gianola, Weigel, Rosa & Crossa 2013), human health (Lebrón-Aldea, Dhurandhar, Pérez-Rodríguez, Klimentidis, Tiwari & Vazquez 2015), electrical engineering (Lampinen, Vehtari & Leinonen 1999), etc. In some applications, for example genetics, it is usual to have a large number of predictors, when included in a neural network the number of parameters to estimate increases exponentially, which could lead to a problem known as *overfitting*. Fortunately there exists some fitting algorithms that can avoid these problems. These algorithms are complex and costly in computational terms and have been implemented in some software packages, for example, the nn toolbox in Matlab, the brnn R-package (Pérez-Rodríguez et al. 2013), the flexible bayesian modeling (Neal 1996), among others.

Modern computers include several CPU cores that can be used to speed up the computations. Some the software packages mentioned above are able to use multiple cores for the computations (e.g. brnn R-package and trainbr function in the nn toolbox in Matlab). Nowadays we also have available computer clusters that can be used to solve complex problems by using several workstations that work cooperatively in the same problem. In this work we present an algorithm based in the strategy "divide and conquer" which allows to train an ANN with a single hidden layer. Part of the sub problems of the general algorithm used for training are solved by using parallel computing techniques, which allows to improve the performance of the resulting application. The proposed algorithm was implemented using the C++ programming language, and the libraries Open MPI (Gabriel, Fagg, Bosilca, Angskun, Dongarra, Squyres, Sahay, Kambadur, Barrett, Lumsdaine, Castain, Daniel, Graham & Woodall 2004) and ScaLAPACK (Blackford, Choi, Cleary, D'Azevedo, Demmel, Dhillon, Dongarra, Hammarling, Henry, Petitet, Stanley, Walker & Whaley 2012) which can be downloaded freely from the internet.

The work is organized as follows. In Section 2 we briefly review linear and non linear models and its relationship with Neural Networks. In Section 3 we review the algorithms that are used to fit a Neural Network. Section 4 presents the implementation of the proposed algorithm. Section 5 shows two application examples. Section 6 presents a benchmark of computational times for the proposed algorithm. Finally in Section 7 we present the concluding remarks.

## 2. Linear models and ANN

Multiple Linear Regression (MLR) model is widely used in statistics and many other fields. The model can be written as:

$$y_i = \beta_0 + \beta_1 x_{i1} + \cdots + \beta_p x_{ip} + e_i, \;\; i = 1, \ldots, n, \;\; j = 1, \ldots, p, \tag{1}$$

where $y_i$ is the response variable for the $i$-th individual, $x_{ij}$ is the value of the $j$-th covariate for individual $i$, $\boldsymbol{\beta} = (\beta_0, \ldots, \beta_p)'$ are regression coefficients to be estimated from the data and $e_i$ are random residuals, usually $e_i \sim N(0, \sigma_e^2)$.

Note that the relationship between the covariates (predictors) is *linear*. In many applications the relationship between predictors and response is *not linear*, and in that case a *non linear* regression model is a better alternative (Fox 2008). The model can be written as:

$$y_i = \beta_0 + f(\boldsymbol{\beta}^*, \boldsymbol{x}_i') + e_i, \tag{2}$$

where $\boldsymbol{\beta}^* = (\beta_1^*, \ldots, \beta_p^*)'$, $\boldsymbol{x}_i' = (x_{i1}, \ldots, x_{ip})$ and $f$ is a function that relates the predictors with the response variable, this function maps from the input space to the real line. Note that model (1) is a special case of (2) since $f(\boldsymbol{\beta}^*, \boldsymbol{x}_i') =$

$\beta_1 x_{i1} + \cdots + \beta_p x_{ip}$. The Kolmogorov's theorem (Kolmogorov 1957) states that any multivariate function can be represented as follows:

$$f(\boldsymbol{x}_i) = f(x_{i1}, \ldots, x_{ip}) = \sum_{q=1}^{2p+1} g_q \left( \sum_{r=1}^{p} h_{qr}(x_{ir}) \right), \tag{3}$$

where $h_{pq}(x)$ are fixed increasing functions on $I = [0, 1]$ and $g_q(\cdot)$ are properly chosen function of one variable (Girosi & Poggio 1989). Therefore, by using the Kolmogorov's theorem, model (2) can be written as:

$$y_i = \beta_0 + \overbrace{\sum_{k=1}^{s} w_k \underbrace{g_k \left( b_k + \sum_{j=1}^{p} x_{ij} \beta_j^{[k]} \right)}_{\text{output from hidden layer}}}^{\text{Combine output from hidden layer}} + e_i \tag{4}$$

where $g_k(x)$ is an *activation function*, for example $g_k(x) = \frac{\exp(2x)+1}{\exp(2x)-1}$, is known as the *tanh* activation function.

Model (4) is known in the literature as a Single Hidden Layer Neural Network (SHLNN). In these models, the prediction is performed in two steps: i) The inputs are transformed non-linearly in the hidden layer, ii) Outputs from hidden layer are combined, and the predictions can be obtained once the parameters are estimated. Figure 1 shows a SHLNN, which includes an *Input layer*, *Hidden layer* with $s$ neurons and the *Output layer*, see Pérez-Rodríguez, Gianola, González-Camacho, Crossa, Manes & Dreisigacker (2012) for further details. Note that no further transformation to the data is applied in the output layer. The MLR model is a particular case of (4). Model (1) is obtained by setting $s = 1$, $w_1 = 1$, $b_1 = 0$ and $g_1(\cdot)$ as the identity function in (4).

The parameters to be estimated in the SHLNN are $w_1, \ldots, w_s$ (neurons' weights), $b_1, \ldots, b_s$ (intercepts or biases), $\beta_1^{[1]}, \ldots, \beta_p^{[1]}; \ldots, \beta_1^{[s]}, \ldots, \beta_p^{[s]}$ (connection strengths for covariates or regression coefficients), $\beta_0$ (general intercept or general mean) and $\sigma_e^2$. Note that here we refer to $w_1, \ldots, w_s$ as neuron's weights and $\beta_1^{[1]}, \ldots, \beta_p^{[1]}; \ldots, \beta_1^{[s]}, \ldots, \beta_p^{[s]}$ can also be referred as weights for covariates. The number of parameter to be estimated for each neuron are $2 + p$, i.e. one neuron's weight ($w$), one intercept ($b$) and $p$ connection strengths for covariates, hence the total number of parameter to estimate in the full network without including the variance component $\sigma_e^2$ is $m = 1 + s \times (2 + p)$. As the number of covariates included in the model and the number of neurons increases the number of parameter to estimate also increases. This could lead to a problem often referred as *overfitting* that is avoided using *penalized estimation algorithms* or more generally *Bayesian regularization*, although another strategies are used commonly in neural networks such as dimensionality reduction, early stopping, pruning, etc. (Prechelt 2012). In the next section we briefly review the Empirical Bayes approach that is used to fit the SHLNN.

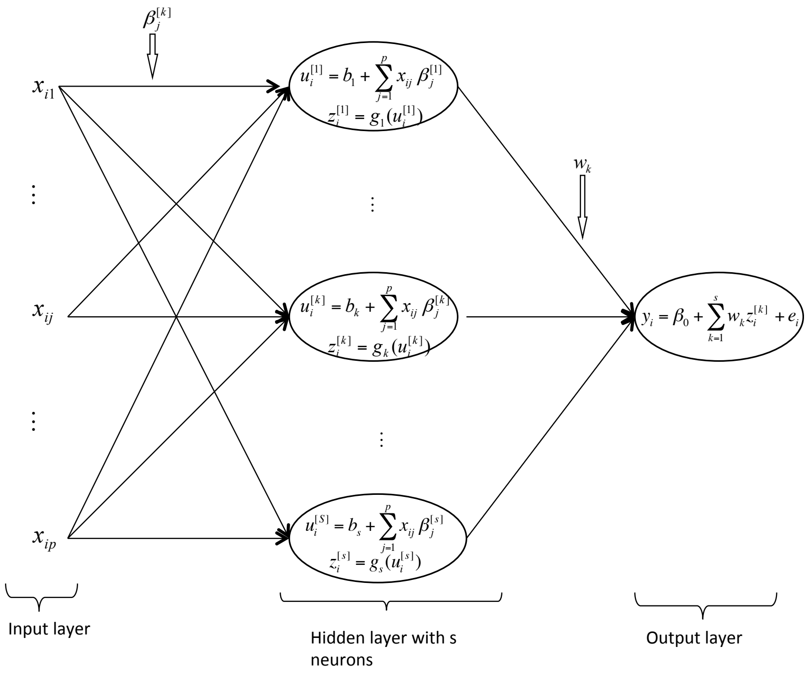

$\beta_j^{[k]}$
$x_{i1}$
$x_{ij}$
$x_{ip}$
$u_i^{[1]} = b_1 + \sum_{j=1}^{p} x_{ij}\beta_j^{[1]}$
$z_i^{[1]} = g_1(u_i^{[1]})$
$u_i^{[k]} = b_k + \sum_{j=1}^{p} x_{ij}\beta_j^{[k]}$
$z_i^{[k]} = g_k(u_i^{[k]})$
$u_i^{[S]} = b_s + \sum_{j=1}^{p} x_{ij}\beta_j^{[s]}$
$z_i^{[s]} = g_s(u_i^{[s]})$
$w_k$
$y_i = \beta_0 + \sum_{k=1}^{s} w_k z_i^{[k]} + e_i$
Input layer
Hidden layer with s neurons
Output layer

FIGURE 1: Single Hidden Layer Neural Network. Source: Pérez-Rodríguez et al. (2012).

## 3. Estimation Algorithms

MacKay (1994) developed Empirical Bayes approach framework for estimating parameters in a neural network. Let $\boldsymbol{\theta} = (w_1, \dots, w_s, b_1, \dots, b_s, \beta_1^{[1]}, \dots, \beta_p^{[1]}; \dots, \beta_1^{[s]}, \dots, \beta_p^{[s]}, \beta_0)'$. MacKay (1994) assigned the following multivariate normal distribution to the elements of $\boldsymbol{\theta}$:

$$p(\boldsymbol{\theta}|\sigma^2_{\boldsymbol{\theta}}) = MN(\mathbf{0}, \sigma^2_{\boldsymbol{\theta}}\boldsymbol{I}),$$

that is $\theta_j \sim N(0, \sigma^2_{\boldsymbol{\theta}}), j = 1, \dots, m$. Note that $\sigma^2_{\boldsymbol{\theta}}$ is common to all the elements of $\boldsymbol{\theta}$. The algorithm to obtain the posterior mode of $\boldsymbol{\theta}$, $\sigma^2_e$, $\sigma^2_{\boldsymbol{\theta}}$ is as follows:

1. Obtain conditional posterior modes of the elements in $\boldsymbol{\theta}$ assuming $\sigma^2_{\boldsymbol{\theta}}, \sigma^2_e$ known. These are obtained by maximizing:

$$p(\boldsymbol{\theta}|\boldsymbol{y}, \sigma^2_{\boldsymbol{\theta}}, \sigma^2_e) = \frac{p(\boldsymbol{y}|\boldsymbol{\theta}, \sigma^2_e)p(\boldsymbol{\theta}|\sigma^2_{\boldsymbol{\theta}})}{p(\boldsymbol{y}|\sigma^2_{\boldsymbol{\theta}}, \sigma^2_e)} = \frac{p(\boldsymbol{y}|\boldsymbol{\theta}, \sigma^2_e)p(\boldsymbol{\theta}|\sigma^2_\theta)}{\int_{\mathbb{R}^m} p(\boldsymbol{y}|\boldsymbol{\theta}, \sigma^2_e)p(\boldsymbol{\theta}|\sigma^2_{\boldsymbol{\theta}})d\boldsymbol{\theta}}$$

which is equivalent to minimizing the *"augmented sum of squares"*:

$$F(\boldsymbol{\theta}) = \frac{1}{2\sigma^2_e}\sum_{i=1}^{n} e_i^2 + \frac{1}{2\sigma^2_{\boldsymbol{\theta}}}\sum_{j=1}^{m} \theta_j^2 \;= \beta E_D + \alpha E_w, \tag{5}$$

where $\beta = \frac{1}{2\sigma_e^2}$ and $\alpha = \frac{1}{2\sigma_{\boldsymbol{\theta}}^2}$, $E_D = \sum_{i=1}^n e_i^2$, $E_w = \sum_{j=1}^m \theta_j^2$. The minimization of $F(\boldsymbol{\theta})$ can be performed using numerical routines, in the present work we adopted the Levenberg-Marquardt algorithm to solve this problem [Levenberg (1944), Marquardt (1963)].

2. Update $\sigma_{\boldsymbol{\theta}}^2, \sigma_e^2$ by maximizing marginal likelihood of the data $p(\boldsymbol{y}|\sigma_{\boldsymbol{\theta}}^2, \sigma_e^2)$. The marginal log-likelihood approximated as:

$$\log p(\boldsymbol{y}|\sigma_{\boldsymbol{\theta}}^2, \sigma_e^2) \approx k + \frac{n}{2}\log\beta + \frac{m}{2}\log\alpha - \frac{1}{2}\log|\boldsymbol{\Sigma}|_{\boldsymbol{\theta}=\boldsymbol{\theta}^{map}} - F(\boldsymbol{\theta})|_{\boldsymbol{\theta}=\boldsymbol{\theta}^{map}}$$

where $\boldsymbol{H} = \frac{\partial^2}{\partial\boldsymbol{\theta}\boldsymbol{\theta}'}F(\boldsymbol{\theta})$, *map* stands for *"maximum a posteriori"*. It can be shown that this function is maximized when:

$$\alpha = \frac{\gamma}{2E_w}, \quad \beta = \frac{n-\gamma}{2E_D}, \quad \gamma = m - 2\alpha tr(\boldsymbol{H})^{-1}. \tag{6}$$

where $\gamma$ is as estimate of the effective number of parameters, and it ranges from 1 to $m$, which corresponds to the number of elements in $\boldsymbol{\theta}$, see Foresee & Hagan (1997) and Gianola, Okut, Weigel & Rosa (2011) for further details, $\boldsymbol{H}$ is the Hessian matrix which is approximated with $\boldsymbol{H} = \nabla^2 F(\boldsymbol{\theta}) \approx 2\beta\boldsymbol{J}'\boldsymbol{J} + 2\alpha\boldsymbol{I}$, $\boldsymbol{J}$ is the Jacobian matrix for the errors in the training set and $\boldsymbol{I}$ is the identity matrix. Note that these solutions are obtained considering $\log p(\boldsymbol{y}|\sigma_{\boldsymbol{\theta}}^2, \sigma_e^2)$ as a function of $\sigma_{\boldsymbol{\theta}}^2$ and $\sigma_e^2$, deriving partially the function with respect to $\sigma_{\boldsymbol{\theta}}^2$ and $\sigma_e^2$, setting the partial derivatives equal to $\mathbf{0}$ and solving the resulting system of equations and rewriting the solution in terms of $\alpha$ and $\beta$.

The algorithm developed by MacKay has been described numerous times in the literature, see for example: Foresee & Hagan (1997), Lampinen et al. (1999), Lampinen & Vehtari (2001), Gianola et al. (2011), Okut, Gianola, Rosa & Weigel (2011), Pérez-Rodríguez et al. (2013), etc. The original implementation in the C programming language can still be downloaded from the author's web site (`http://www.inference.phy.cam.ac.uk/mackay`).

Figure 2 shows a simplified flow chart for the algorithm that fits a SHLNN, the flowchart was generated based on the Foresee & Hagan (1997) algorithm and the flowchart presented in Okut et al. (2011). Figure 3 presents the details of the computations to minimize the "*augmented sum of squares*" by using the Levenberg-Marquardt algorithm [Levenberg (1944), Marquardt (1963)] together fit the rest of steps necessary to fit a SHLNN. In order to initialize the vector $\boldsymbol{\theta}$ we adopted the Nguyen & Widrow (1990) algorithm.

Neal (1996) developed fully Bayesian approach for fitting neural network models, he developed Markov Chain Monte Carlo (MCMC) algorithms to that end. He also developed a computer package called *"flexible Bayesian modeling package"* that implements the MCMC based algorithms, the software can be downloaded from `www.cs.toronto.edu/~radford/fbm.software.html`. Lampinen & Vehtari (2001) implemented part of the routines developed by Neal (1996) using Matlab. Pérez-Rodríguez et al. (2013) extended the algorithms in order shrink differentially two sets of predictors.

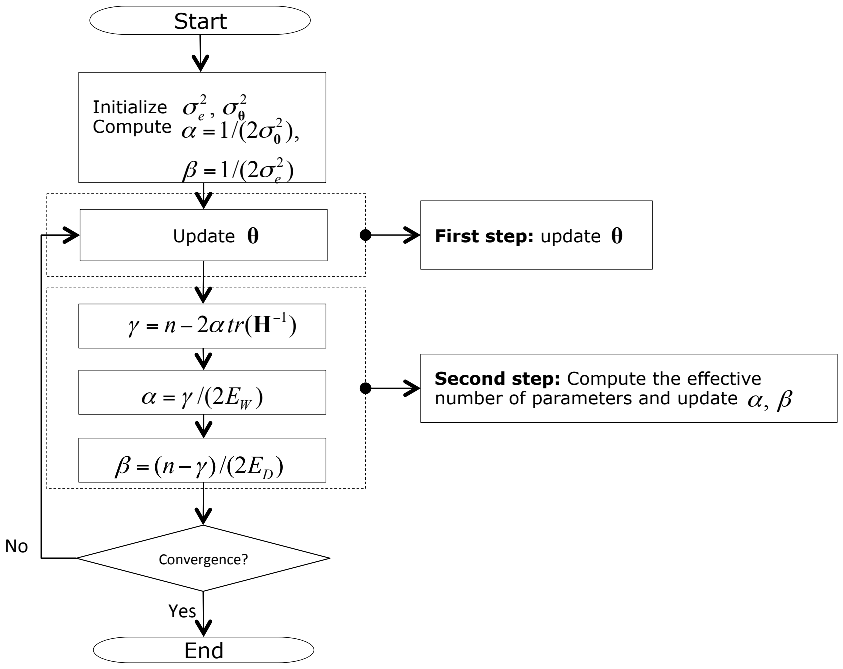

Start
Initialize $\sigma_e^2, \sigma_\theta^2$
Compute $\alpha = 1/(2\sigma_\theta^2)$, $\beta = 1/(2\sigma_e^2)$
Update $\theta$
**First step:** update $\theta$
$\gamma = n - 2\alpha tr(\mathbf{H}^{-1})$
$\alpha = \gamma/(2E_W)$
$\beta = (n-\gamma)/(2E_D)$
**Second step:** Compute the effective number of parameters and update $\alpha, \beta$
Convergence?
No
Yes
End

FIGURE 2: Flow chart for the algorithm that fits a Bayesian Regularized Neural Network with a single hidden layer.

## 4. Implementation

It is well known that as the number of predictors and neurons increases, the number of parameters to estimate increases exponentially. The MacKay's algorithm works very well when the number of parameters to estimate is small, but as the number of parameters to estimate increases three bottle necks becomes evident (see Figure 3):

1. The computation of the Jacobian ($\boldsymbol{J}$) and the approximation of the Hessian matrix ($\boldsymbol{H}$).
2. The updating of $\boldsymbol{\theta}$ in the Levenberg-Marquardt algorithm.
3. The computation of the effective number of parameters $\gamma$ when updating $\sigma_e^2$, $\sigma_{\boldsymbol{\theta}}^2$.

The most computational expensive operations in the include the solving of linear system of equations and obtaining the inverse of a matrix. Up to now the existing software routines that implement the Empirical Bayes approach proposed by MacKay are not able to use computational resources efficiently. The original software developed by MacKay was designed in order to run in a single core. The Matlab routine *trainbr* in the *nn* toolbox and the one developed by Pérez-Rodríguez et al. (2013) and implemented in the *brnn*-R package are able to use several cores in modern multi-core workstations, but are unable to use multiple cores in several works stations available in modern computing clusters. In the case of MCMC based approaches, the problem is even worse, the parallelization of the tasks is not possible.

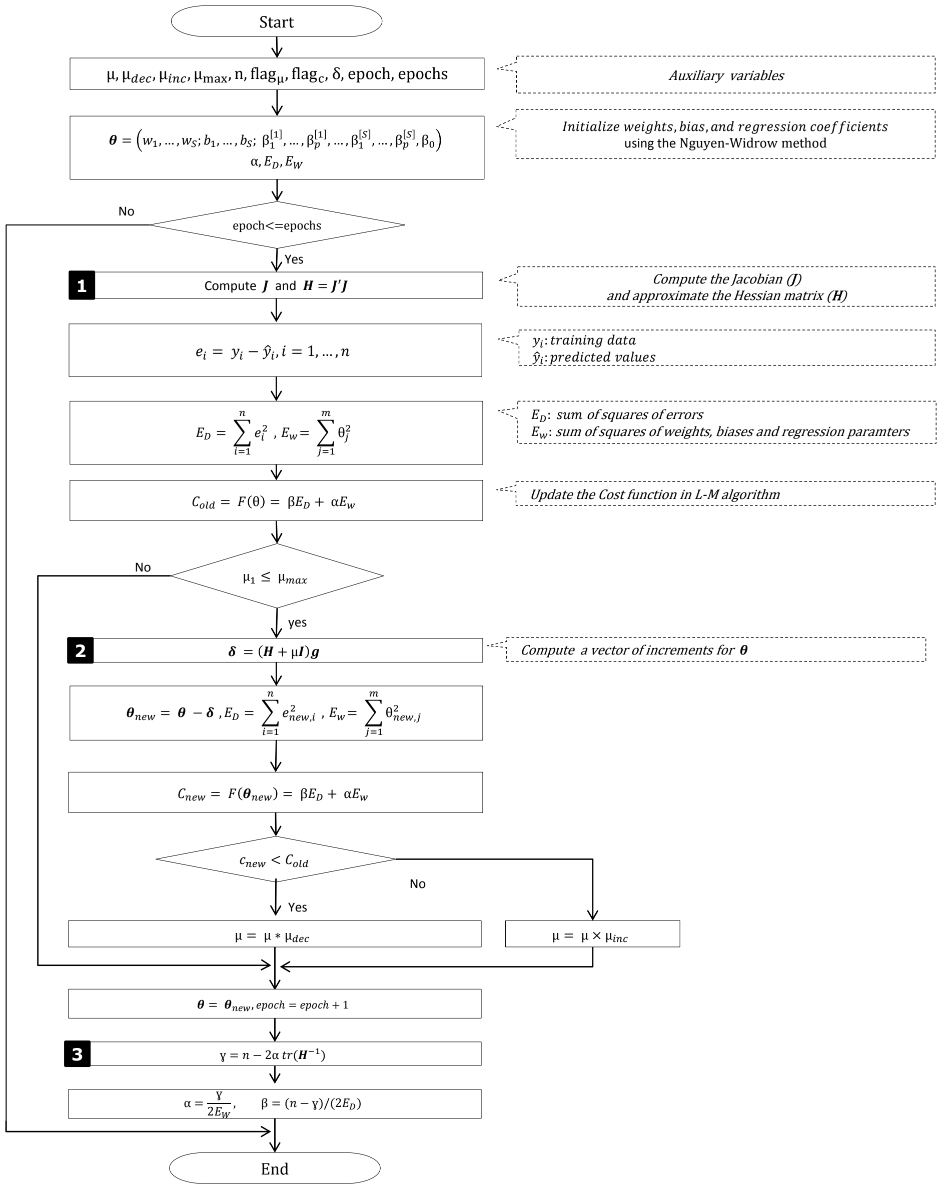

Start
$\mu, \mu_{dec}, \mu_{inc}, \mu_{\max}, n, \text{flag}_\mu, \text{flag}_c, \delta, \text{epoch}, \text{epochs}$
*Auxiliary variables*
$\boldsymbol{\theta} = \left(w_1, \dots, w_S; b_1, \dots, b_S;\ \beta_1^{[1]}, \dots, \beta_p^{[1]}, \dots, \beta_1^{[S]}, \dots, \beta_p^{[S]}, \beta_0\right)$
$\alpha, E_D, E_W$
*Initialize weights, bias, and regression coefficients* using the Nguyen-Widrow method
No
epoch<=epochs
Yes
1
Compute $\boldsymbol{J}$ and $\boldsymbol{H} = \boldsymbol{J}'\boldsymbol{J}$
*Compute the Jacobian (J) and approximate the Hessian matrix (H)*
$e_i = y_i - \hat{y}_i, i = 1, \dots, n$
$y_i$: *training data*
$\hat{y}_i$: *predicted values*
$E_D = \sum_{i=1}^{n} e_i^2$ , $E_w = \sum_{j=1}^{m} \theta_j^2$
$E_D$: *sum of squares of errors*
$E_w$: *sum of squares of weights, biases and regression paramters*
$C_{old} = F(\theta) = \beta E_D + \alpha E_w$
*Update the Cost function in L-M algorithm*
No
$\mu_1 \leq \mu_{max}$
yes
2
$\boldsymbol{\delta} = (\boldsymbol{H} + \mu \boldsymbol{I})\boldsymbol{g}$
*Compute a vector of increments for* $\boldsymbol{\theta}$
$\boldsymbol{\theta}_{new} = \boldsymbol{\theta} - \boldsymbol{\delta}$, $E_D = \sum_{i=1}^{n} e_{new,i}^2$ , $E_w = \sum_{j=1}^{m} \theta_{new,j}^2$
$C_{new} = F(\boldsymbol{\theta}_{new}) = \beta E_D + \alpha E_w$
$c_{new} < C_{old}$
No
Yes
$\mu = \mu * \mu_{dec}$
$\mu = \mu \times \mu_{inc}$
$\boldsymbol{\theta} = \boldsymbol{\theta}_{new}, epoch = epoch + 1$
3
$\gamma = n - 2\alpha\, tr(\boldsymbol{H}^{-1})$
$\alpha = \frac{\gamma}{2E_W}$, $\beta = (n - \gamma)/(2E_D)$
End

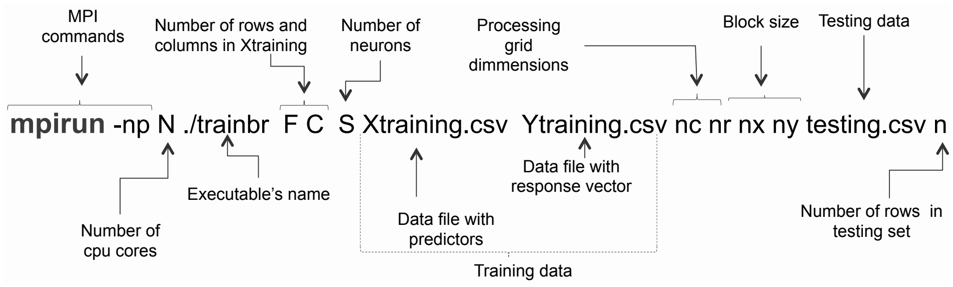
FIGURE 3: Levenberg-Marquardt algorithm.

In this work, we propose to alleviate the bottle necks in the EB algorithm by using parallel computing. We propose to use the software packages *ScaLAPACK* (`http://www.netlib.org/scalapack/`) and *OpenMPI* (`http://www.open-mpi.org`). ScaLAPACK stands for **Sca**lable **L**inear **A**lgebra **PACK**age (Blackford et al. 2012). The ScaLAPACK library is a software that was designed to perform efficiently many linear algebra operations. The library was designed to give high efficiency on Multiple Instructions Multiple Data (MIDM) distributed memory concurrent supercomputers (Yoginath, Bauer, Kora, Samatova, Kora, Fann & Geist 2009). The software can be used with clusters of work stations through network environments via Multiple Processing Interface (MPI). The ScaLAPACK library is based on block-partitioned algorithms that minimize the movement between different processes. ScaLAPACK uses a Parallel BLAS (Basic Linear Algebra Subroutines) for the computations and a set of routines (Basic Linear Algebra Communications Subprograms, BLACS) for the communication between the processes. The BLACS routines rely on one implementation of MPI, one of the most widely used is OpenMPI (Gabriel et al. 2004). In order to perform any computation with ScaLAPACK it is necessary to create a two-dimensional grid with $r$ rows and $c$ columns encompassing the process to be involved in a particular computation, this grid is referred as *process grid*. For example, it is possible to specify that we want to use 8 CPUs cores to perform a product matrix, then the four possible processing grids are $(2, 4)$, $(4, 2)$, $(8, 1)$, $(1, 8)$. Once that the *processing grid* has been defined the data is distributed among the involved process using an algorithm that is known as *block-cyclic data distribution*.

ScaLAPACK was written in Fortran and C/C++ programming languages. Routines can be call from programs written in both programming languages. In this work we decided to use C/C++ in order to implement the Empirical Bayes approach because this is the programming language that we have been using routinely. We use the following functions available in ScaLAPACK to perform the computations:

- *pdgemm*: Performs a matrix-matrix product.
- *pdgemv*: Performs a matrix-vector product.
- *pdgesv*: Solves a linear system of equations without inverting the matrix of coefficients.
- *pdpotrf*: Performs the Cholesky decomposition of a matrix.
- *pdpotri*: Inverts a positive definite and symmetric matrix using the Cholesky decomposition.

The resulting application is a console program that can run in standalone workstations or in computing clusters. The source code is available upon request from the authors. Figure 4, shows the command line and the arguments that are used to execute the program.



```
mpirun -np N ./trainbr F C S Xtraining.csv Ytraining.csv nc nr nx ny testing.csv n
```

MPI commands
Number of rows and columns in Xtraining
Number of neurons
Processing grid dimmensions
Block size
Testing data
Number of cpu cores
Executable's name
Data file with predictors
Data file with response vector
Training data
Number of rows in testing set

FIGURE 4: Command line example to execute the application that fits a neural network using parallel computing. The application uses ScaLAPACK to perform the matrix algebra operations, the communication between processes is performed using OpenMPI.

## 5. Application examples

In this section we present two applications examples for ANN. In the first example we predict building energy loads from a series of input variables. The response variable is electricity consumption and the input variables are temperature, humidity, solar flux and wind. The dataset was analyzed previously by MacKay (1994). In the second example we analyze grain yield for wheat. The data set was generated by International Center of Maize and Wheat Improvement (CIMMyT, `http://www.cimmyt.org`). The goal in this case is to predict the grain yield of wheat lines using 1279 molecular markers (input variables). The dataset was first analyzed by Crossa, de los Campos, Pérez-Rodríguez, Gianola, Burgueño, Araus, Makumbi, Singh, Dreisigacker, Yan, Arief, Banziger & Braun (2010) and has been reanalyzed by many authors since then. Finally we present a benchmark of the application.

### 5.1. Electricity

For this example we selected the electricity consumption data reported by MacKay (1994). The dataset was downloaded from the following web site `http://www.inference.phy.cam.ac.uk/mackay/Bayes_FAQ.html#Data`. The dataset consisted of hourly measurements from September 1 1989 to February 23 1990 of four input variables (temperature, humidity, solar flux, and wind) and three response variables: electricity, cooling water and heating water. Here we only analyze electricity consumption. The total number of records in the dataset is 4208. MacKay (1994) divided the data into two sets: Training and Testing. The testing set consisted of the last 1282 data points (January 1 1990 to February 23 1990). The problem is to predict electricity consumption using the input variables. Figure 5 shows the Whole Building Electric consumption (WBE), and also shows the training and testing sets. MacKay (1994) fitted neural networks with a single hidden layer of *tanh* units, he found that models with between 4 and 8 neurons were appropriate for the problem. MacKay (1994) omitted some records in the training set, because when fitting the models he obtained large residuals (see Figure 5). MacKay preprocessed the original data and he added variables that allow him to get different representations of time and holidays, moving averages of the environmental variables, etc. So at the end of the preprocessing of the original data he got a data set with 25 variables that he used as inputs. The pre-processed dataset can also be downloaded from the MacKay's web site.

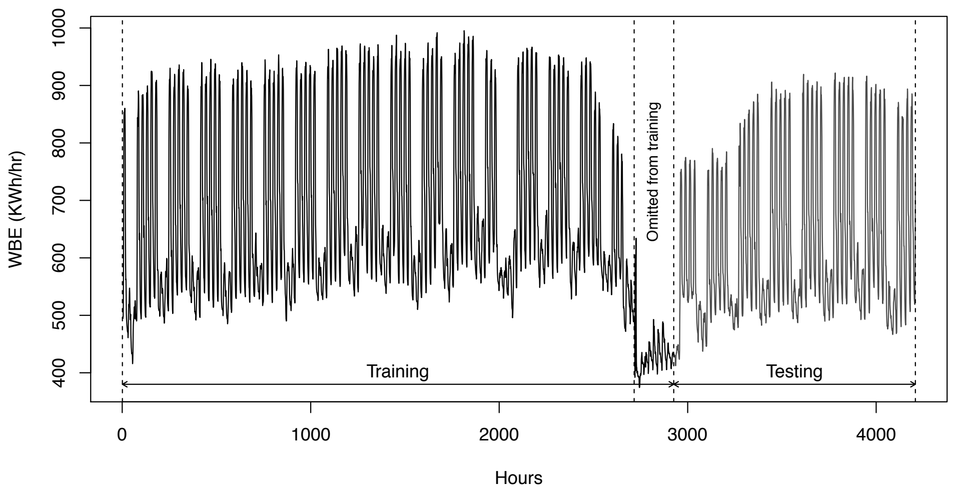


FIGURE 5: Whole Building Electric consumption (WBE) in KWh/hr from September 1 1989 to December 31 to February 23 1990.

We fitted an ANN using the pre-processed dataset created by MacKay that we described previously, the goal is just to show that the proposed algorithm works as expected. We fitted an ANN with $s = 8$ neurons and 4 CPU cores, the computing time necessary to complete the task were approximately 2 seconds. The number of input variables was $p = 25$, so with $s = 8$ neurons, the total number of weights, biasses and regression coefficients estimated was $s \times (p + 1 + 1) = 216$. Furthermore $\hat{\beta} = 184.0268$, $\hat{\alpha} = 1.5994$ and the effective number of parameters $\gamma = 207.26$. Note than the effective number of parameters is close to 216, so it is not necessary to include more than 8 neurons in the model.

Figure 6 shows the original data, the predictions obtained from the model and the residuals. It can be seen that the model predicts very well, even that is not an easy task. The RMSE was 41.34 and the Pearson's correlation between observed and predicted values was 0.9538. Note also that even that the model has a huge number of parameter it predicts very well in the testing set. It is well known that non regularized neural networks tends to overfit the training data and when the model is used to predict using a new dataset the predictions are very bad, but this is not the case in this application example because the ANN used avoids overfitting automatically.

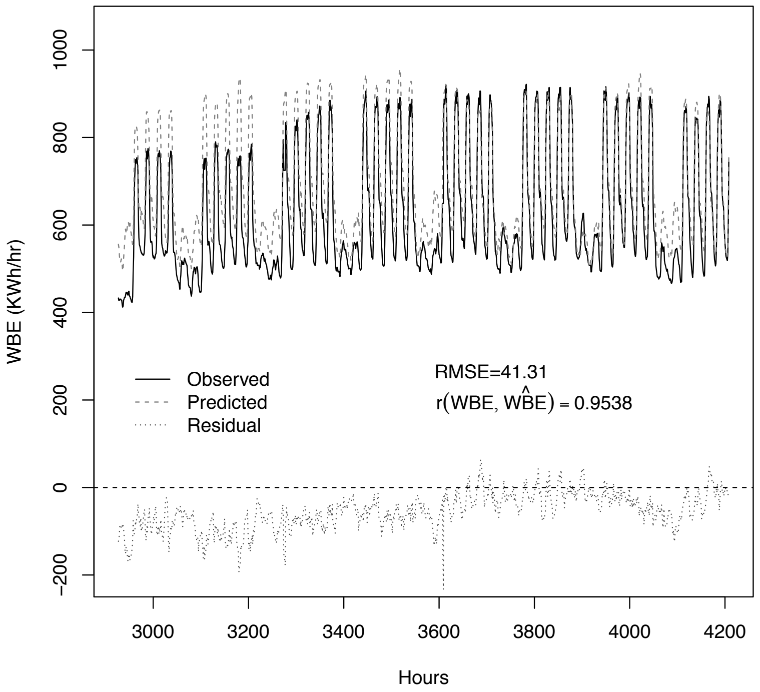


FIGURE 6: Predictions for Whole Building Electric consumption (WBE) in KWh/hr from January 1 1990 to February 23 1990. The Figure displays 3 series: Observed data, predicted data and residuals.

## 5.2. Wheat Dataset

The wheat data set is from CIMMYT's Global Wheat Program (`http://www.cimmyt.org`). Historically, this program has conducted numerous international trials across a wide variety of wheat-producing environments. The environments represented in these trials were grouped into four basic target sets of environments comprising four main agroclimatic regions previously defined and widely used by CIMMYT's Global Wheat Breeding Program. The phenotypic trait considered here was the average grain yield (GY) of the 599 wheat lines evaluated in each of these four mega-environments. Wheat lines were recently genotyped using 1447 Diversity Array Technology (DArT) generated by Triticarte Pty. Ltd. (Canberra, Australia; `http://www.triticarte.com.au`). The DArT markers may take on two values, denoted by their presence(1) or absence(0). The dataset was first analyzed by Crossa et al. (2010). The dataset can be downloaded from the following web site: `http://www.genetics.org/content/suppl/2010/09/02/genetics.110.118521.DC1`.

From the statistical point of view we have a prediction problem, the response variable ($y$) corresponds to grain yield and the predictors are the molecular markers ($x_1, \ldots, x_{1279}$). Crossa et al. (2010) predicted the grain yield using a MLR model, that is:

$$y_i = \beta_0 + \sum_{j=1}^{1279} x_{ij}\beta_j + e_i, i = 1, \dots, 599.$$

The model was fitted using the Bayesian framework by using different priors for $\beta_j$. Crossa et al. (2010) studied the predicted ability of the proposed model using cross-validation. The authors divided at random the dataset into 10 disjoint subsets, $S_1, \dots, S_{10}$. The observations are assigned to the sets which are then used in the evaluation process. For example, the full sample can be divided in *training* and *testing* sets, if set 1 is selected as testing, then observations in $S_2, \dots, S_{10}$ are used to train the model and obtain the predictions for individuals in the testing set, that is $\{\hat{y}_i, i \in S_1\}$ and compute the Pearson's correlation between observed and predicted values. Pearson's correlation is the standard statistic to measure linear relationship between two continuous variables, in this case the variables are observed phenotypic values ($\boldsymbol{y}$) and predicted phenotypical values ($\hat{\boldsymbol{y}}$). If the fitted model is predicting correctly then we expect that $\boldsymbol{y} \approx \hat{\boldsymbol{y}}$. In the case of prediction of complex traits (those that are controlled by large number of genes and with low heritabilities, e.g. grain yield) with high dimensional data like the one presented in this example correlations between observed and predicted value are expected to be low. The exercise can be repeated for the rest of the sets. Table 1 shows the results from the cross-validation experiment for Environment 1, together with the execution times. Note that the Pearson's correlation in the training set is always greater that the cross-validation in the testing set. The average of Pearson's correlation was 0.4559, a result similar to this was reported by Crossa et al. (2010).

Table 1: Pearson's correlations between observed and predicted values in training and testing set for the 10 fold Cross Validation experiment.

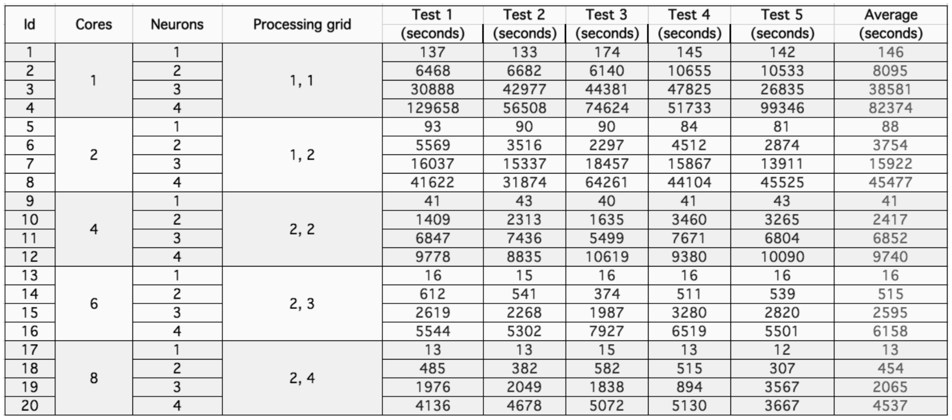
| Fold | $r_{trn}$ | $r_{test}$ | time (hrs) |
|---|---|---|---|
| 1 | 0.9781 | 0.5900 | 2.8 |
| 2 | 0.9748 | 0.6474 | 1.9 |
| 3 | 0.9752 | 0.4692 | 2.5 |
| 4 | 0.9711 | 0.3642 | 1.9 |
| 5 | 0.9842 | 0.3298 | 4.3 |
| 6 | 0.9742 | 0.5569 | 2.9 |
| 7 | 0.9876 | 0.4070 | 3.2 |
| 8 | 0.9724 | 0.4362 | 6.1 |
| 9 | 0.9768 | 0.2276 | 2.9 |
| 10 | 0.9782 | 0.5312 | 3.8 |
| **Avg.** | **0.9781** | **0.4559** | **3.23** |

## 6. Benchmark

We carried out a benchmark evaluation by fitting a Bayesian Regularized Neural Network using the wheat dataset described previously. We considered different scenarios involving 4 different number of neurons ($s = 1, 2, 3$ and 4) and $N = 1, 2, 4, 6, 8$ CPU cores. The CPU cores were arranged in 5 different processing grids: $(1, 1)$, $(1, 2)$, $(2, 2)$, $(2, 3)$, $(2, 4)$. The evaluation was carried out in a Linux workstation with an AMD Opteron processor (tm) with 32 core CPUs @ 2.6 GHz and 160 Gb of RAM memory. We obtained the source code for OpenMPI and ScaLAPACK libraries from `http://www.open-mpi.org` and `http://www.netlib.org/scalapack/` respectively and then we compiled and installed the libraries using the configuration scripts included with the source code of the libraries. The proposed algorithm uses the Nguyen & Widrow (1990) algorithm in order to initialize the weights, biases and regression coefficients in the vector $\boldsymbol{\theta}$, which can lead to different solutions and also it is possible that the running times vary from run to run, so we repeated each simulation experiment 5 times.

Table 2 shows the processing times necessary to fit the model. Figure 7 shows the processing times against the number of cores, from the Figure, it is clear that the time necessary to fit the models increases exponentially with the model complexity (number of neurons), on the other hand if the number of computing cores used to fit the model increases, the computing time decreases. Note that the decrease rate in computing time for a fixed number of neurons is not a linear function of the number of CPU cores used for the fitting, also note that for this particular problem the computing times are almost constant when using 4, 6 or 8 CPU cores, unfortunately we do not know how to determine the optimal CPU cores that must be used in the general case. Figure 8 shows the processing times vs the number of neurons, if the number of neurons increases the processing time also increases and the processing times decrease significantly as the number of CPU cores increases. Note also that there the relationship between processing times and number of neurons is not linear.

Table 2: Running times.



| Id | Cores | Neurons | Processing grid | Test 1 (seconds) | Test 2 (seconds) | Test 3 (seconds) | Test 4 (seconds) | Test 5 (seconds) | Average (seconds) |
|---|---|---|---|---|---|---|---|---|---|
| 1 | 1 | 1 | 1, 1 | 137 | 133 | 174 | 145 | 142 | 146 |
| 2 | | 2 | | 6468 | 6682 | 6140 | 10655 | 10533 | 8095 |
| 3 | | 3 | | 30888 | 42977 | 44381 | 47825 | 26835 | 38581 |
| 4 | | 4 | | 129658 | 56508 | 74624 | 51733 | 99346 | 82374 |
| 5 | 2 | 1 | 1, 2 | 93 | 90 | 90 | 84 | 81 | 88 |
| 6 | | 2 | | 5569 | 3516 | 2297 | 4512 | 2874 | 3754 |
| 7 | | 3 | | 16037 | 15337 | 18457 | 15867 | 13911 | 15922 |
| 8 | | 4 | | 41622 | 31874 | 64261 | 44104 | 45525 | 45477 |
| 9 | 4 | 1 | 2, 2 | 41 | 43 | 40 | 41 | 43 | 41 |
| 10 | | 2 | | 1409 | 2313 | 1635 | 3460 | 3265 | 2417 |
| 11 | | 3 | | 6847 | 7436 | 5499 | 7671 | 6804 | 6852 |
| 12 | | 4 | | 9778 | 8835 | 10619 | 9380 | 10090 | 9740 |
| 13 | 6 | 1 | 2, 3 | 16 | 15 | 16 | 16 | 16 | 16 |
| 14 | | 2 | | 612 | 541 | 374 | 511 | 539 | 515 |
| 15 | | 3 | | 2619 | 2268 | 1987 | 3280 | 2820 | 2595 |
| 16 | | 4 | | 5544 | 5302 | 7927 | 6519 | 5501 | 6158 |
| 17 | 8 | 1 | 2, 4 | 13 | 13 | 15 | 13 | 12 | 13 |
| 18 | | 2 | | 485 | 382 | 582 | 515 | 307 | 454 |
| 19 | | 3 | | 1976 | 2049 | 1838 | 894 | 3567 | 2065 |
| 20 | | 4 | | 4136 | 4678 | 5072 | 5130 | 3667 | 4537 |

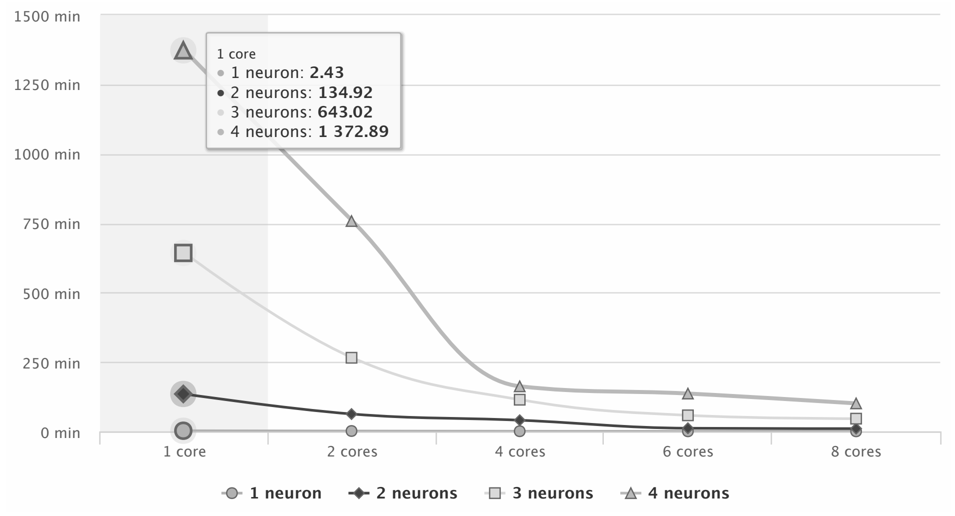


Figure 7: Running times vs number of cores.

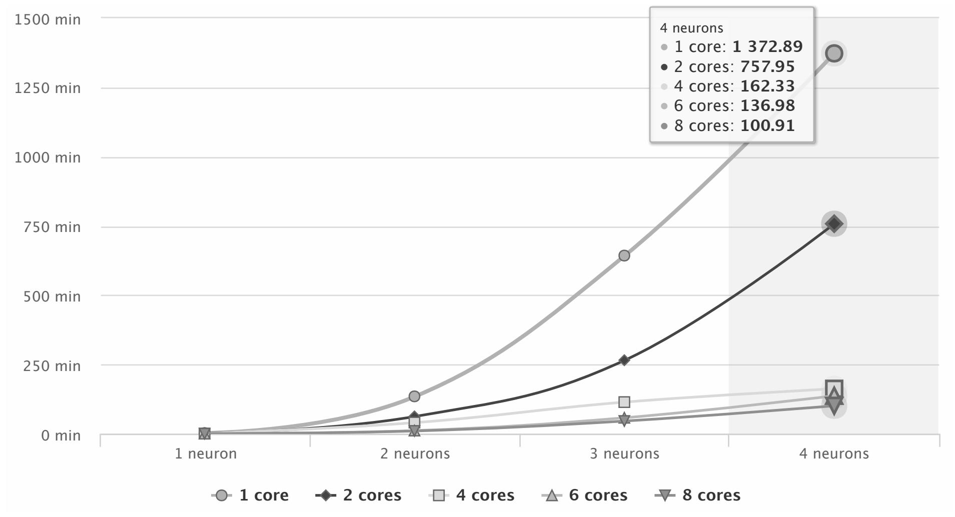


Figure 8: Running times vs number of neurons.

## 7. Concluding Remarks

The algorithms used to fit a Bayesian regularized neural networks are well know. It is also well known that as the number of neurons and the number of predictors increases the computing time necessary to fit the model also increases. We identified some bottle necks in the algorithm used to fit the model and we proposed to use parallel computing in order to alleviate this problem. The resulting application can be run in multicore computers, but it can also run in computing clusters where different multicore computers (computing nodes) can work cooperatively to solve a given problem.

[Received: January 2016 — Accepted: May 2018]

## References

Blackford, L. S., Choi, J., Cleary, A., D'Azevedo, E., Demmel, J., Dhillon, I., Dongarra, J., Hammarling, S., Henry, G., Petitet, A., Stanley, K., Walker, D. & Whaley, R. C. (2012), 'ScaLAPACK users' guide', Netlib's official site. 2015-03-14.
*http://netlib.org/scalapack/slug/index.html

Crossa, J., de los Campos, G., Pérez-Rodríguez, P., Gianola, D., Burgueño, J., Araus, J. L., Makumbi, D., Singh, R. P., Dreisigacker, S., Yan, J., Arief, V., Banziger, M. & Braun, H.-J. (2010), 'Prediction of genetic values of quantitative traints in plant breeding using pedigree and molecular markers', *Genetics* **186**(2), 714–724.

Foresee, F. D. & Hagan, M. T. (1997), 'Gauss Newton approximation to Bayesian learning', *Proc. IEEE International Conference Neural Networks* **3**, 1931–1933.

Fox, J. (2008), *Applied Regression Analysis and Generalized Linear Models*, SAGE Publications.
*http://books.google.ch/books?id=GKkn3LSSHFsC

Gabriel, E., Fagg, G. E., Bosilca, G., Angskun, T., Dongarra, J. J., Squyres, J. M., Sahay, V., Kambadur, P., Barrett, B., Lumsdaine, A., Castain, R. H., Daniel, D. J., Graham, R. L. & Woodall, T. S. (2004), Open MPI: Goals, concept, and design of a next generation MPI implementation, *in* 'Proceedings, 11th European PVM/MPI Users' Group Meeting', Budapest, Hungary, pp. 97–104.

Gianola, D., Okut, H., Weigel, K. A. & Rosa, G. J. (2011), 'Predicting complex quantitative traits with bayesian neural networks: a case study with jersey cows and wheat', *BMC Genetics* **87**, 7–8.

Girosi, F. & Poggio, T. (1989), 'Representation properties of networks: Kolmogorov's theorem is irrelevant', *Neural Computation* **1**(4), 465–469.
*http://dx.doi.org/10.1162/neco.1989.1.4.465

Kolmogorov, A. K. (1957), 'On the representation of continuous functions of several variables by superposition of continuous functions of one variable and addition', *Doklady Akademii Nauk SSSR* **114**, 369–373.

Lampinen, J. & Vehtari, A. (2001), 'Bayesian approach for neural networks review and case studies', *Neural networks* **14**(3), 259–262.

Lampinen, J., Vehtari, A. & Leinonen, K. (1999), 'Application of bayesian neural network in electrical impedance tomography', **6**, 3942–3947.

Lebrón-Aldea, D., Dhurandhar, E. J., Pérez-Rodríguez, P., Klimentidis, Y. C., Tiwari, H. K. & Vazquez, A. I. (2015), 'Integrated genomic and bmi analysis for type 2 diabetes risk assessment', *Frontiers in Genetics* **6**, 1–8.

Levenberg, K. (1944), 'A method for the solution of certain problems in least squares', *Quart. Applied Math.* **2**, 164–168.

MacKay, D. J. (1994), 'Bayesian non-linear modelling for the prediction competition', *ASHRAE transactions* **100**, 5–12.

Marquardt, D. W. (1963), 'An algorithm for least-squares estimation of nonlinear parameters', *SIAM Journal on Applied Mathematics* **11**(2), 431–441.

Neal, R. M. (1996), *Bayesian Learning for Neural Networks*, Springer-Verlag New York, Inc., Secaucus, NJ, USA.

Nguyen, D. & Widrow, B. (1990), Improving the learning speed of 2-layer neural networks by choosing initial values of the adaptive weights, *in* 'International Symposium on Neural Networks'.

Okut, H., Gianola, D., Rosa, G. J. & Weigel, K. (2011), 'Prediction of body mass index in mice using dense molecular markers and a regularized neural network', *Genetics research* **93**, 189–201.

Pérez-Rodríguez, P., Gianola, D., González-Camacho, J. M., Crossa, J., Manes, Y. & Dreisigacker, S. (2012), 'Comparison between linear and non-parametric regression models for genome-enabled prediction in wheat', *G3* **2**(12), 1595–1605.

Pérez-Rodríguez, P., Gianola, D., Weigel, K. A., Rosa, G. J. M. & Crossa, J. (2013), 'Technical note: An R package for fitting Bayesian regularized neural networks with applications in animal breeding', *Journal of Animal Science* **91**, 3522–3525.

Prechelt, L. (2012), Early stopping – but when?, *in* G. Montavon, G. B. Orr & K.-R. Müller, eds, 'Neural Networks: Tricks of the Trade: Second Edition', Springer Berlin Heidelberg, Berlin, Heidelberg, pp. 53–67. *https://doi.org/10.1007/978-3-642-35289-8_5

Yoginath, S., Bauer, D., Kora, G., Samatova, N., Kora, G., Fann, G. & Geist, A. (2009), 'RScaLAPACK: High-performance parallel statistical computing with R and ScaLAPACK'. 2016-01-12. *http://web.ornl.gov/ webworks/cppr/y2001/pres/124121.pdf