Congr. Int. en Ing. Electrónica. Mem. ELECTRO, Vol. 39, pp. 269-274, Oct 2017, Chihuahua, Chih. México
http://electro.itchihuahua.edu.mx/memorias_electro/MemoriaElectro2017.zip
ISSN 1405-2172

DISTRIBUTED BACKPROPAGATION NEURAL NETWORK ALGORITHM IMPLEMENTED IN MATLAB'S PARALLEL COMPUTER TOOLBOX

David Sáenz-Zamarrón[1], Nancy I. Arana-de-las-Casas, José F. Alatorre-Ávila, Enrique García-Grajeda, Cesar Sáenz-Zamarrón and Miriam A. Mendoza-Mendoza

[1]Instituto Tecnológico de Cd. Cuauhtémoc
Departamento de Sistemas y Computación
Av. Tecnológico No. 137, Cuauhtémoc, Chih. México. C.P. 31500
52 (625) 581 1707
davsaenz@gmail.com

RESUMEN

Para las redes neurales los algoritmos de aprendizaje involucran procesamiento intensivo en CPU y se han desarrollado implementaciones paralelas para reducir el tiempo de aprendizaje. Este artículo propone un esquema paralelo para el algoritmo de retro propagación, este consiste de una arquitectura del sistema distribuida, la cual desarrolla el entrenamiento paralelo mediante la partición de patrones. Esta implementación está basada en la estructura *parfor* de Matlab, la cual explota una clase de paralelismo inherente a las redes neurales alimentadas hacia adelante entrenadas con retro propagación, llamada paralelismo de tareas. Bajo este desarrollo, los cambios en los pesos son calculados concurrentemente, intercambiados entre *Workers* y ajustados apropiadamente hasta que el proceso de aprendizaje se completa. Se muestra el diseño de un soporte distribuido para aprendizaje paralelo de redes neurales usando el método de partición de patrones, resultados teóricos y experimentales de aceleración son analizados.

Palabras clave:Retro propagación paralela, partición de patrones, parfor de Matlab.

ABSTRACT

In neural networks, learning algorithms engage CPU intensive processing and parallel implementations which can lead to reduce learning time. In this paper, a parallel scheme for a back propagation algorithm is proposed; it consists of Matlab´s distributed system architecture, which develops a parallel training with a partition pattern scheme. This implementation is based on the Matlab´s *parfor* structure, which exploits a kind of parallelism inherent in back propagation feed forward, layered neural networks, namely task parallelism. Under this approach, weight changes are computed concurrently, exchanged between *Workers* and adjusted accordingly until the whole parallel learning process is completed. This article shows the design of a distributed support for parallel learning of neural networks using a pattern partitioning approach. Results on speedup theoretically and experimentally are shown.

Keywords: Parallelized back propagation, pattern partitioning, Matlab's parfor.

1. INTRODUCTION

Feed forward neural networks (FFNN) are extensively used to solve complex problems in pattern classification, system modeling and identification, among other things. One of the characteristics of the FFNN is its learning (or training) ability. Learning is the adjustment process of the neural network to external stimuli. By training, the neural network can give correct answers not only for learned examples, but also for the inputs similar to the learned examples, showing its strong associative and rational ability, which are suitable for solving large, nonlinear problems with complex classification, some others require function approximation. These properties are attained during a slow learning process. The standard method for training FFNN is the back propagation (BP) algorithm which is based on the gradient descent optimization technique. Despite the general success of this algorithm it may converge to a local minimum of the mean squared-error objective function and requires a large number of learning iterations to adjust the weights of the FFNN.

Many approaches to speed up the training process have been devised by means of parallelism [1]-[7]. To parallelize BP either the network or the training pattern space is partitioned [4], and run in parallel into a multi-core processors computer or across a computer multi-node cluster connected via a communication network [8].

Parallel programming using supercomputers is not available for everyone, and the use of separated tools as C++/FORTRAN and MPI libraries is not easy, without mentioning that creating parallel code in these languages takes a long time [9]. To lessen this problem, Matlab offers an integrated development tool for parallel computing, allowing solving complex problems efficiently [10].

In pattern partitioning the whole neural net is replicated in different *Workers* (the Matlab's parallel processing unit) and the weight changes due to distinct training patterns are parallelized [2]. As a result, this parallel model creates and trains a separate network for each pattern in parallel. The interconnection weights are updated by an amount based on the total weight change summed over all networks. This technique exploits task parallelism as well as architectural parallelism of the back propagation neural network algorithm.

The paper describes the development of a parellel back propagation algorithm, referred to as the PBP, and its implementation on Matlab's *parfor* parallel structure, also discusses its classification correctness, the PBP algorithm´s speedup factor into the distributed Matlab implementation, and a Gigaflops performance benchmark.

2. FEED FORWARD NEURAL NETWORK

The network will be applied to solve a 26-class pattern classification problem, one for each letter of the alphabet. It involves classifying each input pattern into one of R=26 possible classes. The data set consists of P=26 patterns; each

pattern of N=35 values, which is defined by a 5x7 bitmap of a letter, for training. Each pattern has N=35 values which can either be 1 or 0. Figure 1 shows one input pattern.

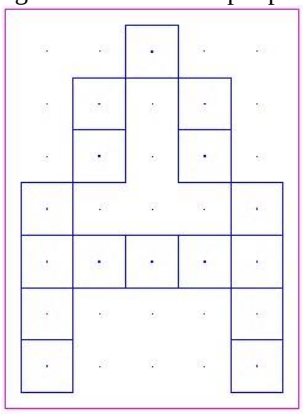

Fig. 1. One pattern of the total 26 alphabet, each consist of a letter in a 5x7 bitmap.

Let the node numbers of input and hidden layer be N and M, respectively. The node number of the output layer is defined as R. The pattern space contains a total number of P patterns. The whole synapses (interconnections) are given by (N+1) (M+1) + (M+1) (R+1) = 2268. Figure 2 shows a three layer FF network. The weight connected to the input with value one (1) included in all the layers of the FFNN represents the offset of the network. The implementation in this paper allows growing the number of hidden layers as wished, that´s why always the output of one layer includes a one (1), which will be used as offset for the next one.

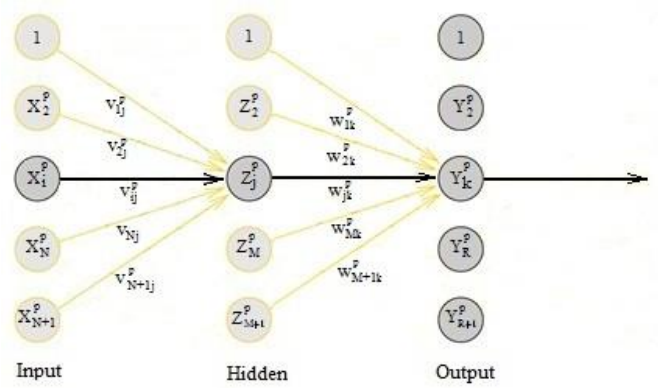

Fig. 2. Architecture and nomenclature of the Feed Forward Neural Network

X_i^p is the ith inputs for the pth pattern to the input NN layer, i varies from 1 to N+1, therefore the size of X is P (N+1).

$$X_i^p = \begin{bmatrix} 1 & \cdots & 0 & 0 & \cdots & 1 \\ \vdots & \ddots & \vdots & \vdots & \ddots & \vdots \\ 1 & \cdots & 1 & 0 & \cdots & 0 \end{bmatrix}_{P(N+1)} \quad (1)$$

Where v_{ij}^p represents the pth pattern to the first NN weight stage, i varies from 1 to N+1 and j does from 1 to M+1, therefore the size of v is P(N+1) (M+1) and initially is initialized with small random values.

$$v_{ij}^p = \begin{bmatrix} 8.8050 & \cdots & .00081 \\ \vdots & \ddots & \vdots \\ 0 & \cdots & .00013 \end{bmatrix}_{(N+1)(M+1)} \quad (2)$$

The offset weight 8.8050 is in position v_{11}^p, (equation 2), the other elements in the first columns are zero. The value of v_{11}^p is assigned in this way in order to obtain a value one as output in the next layer when the activation function in this layer is applied, see section 2.1.

The stimulus to the hidden layer is calculated as

$$z_{in_j}^p = \sum_{i=1}^{N+1} X_i^p v_{ij}^p \quad (3)$$

We also analyze the time complexity for the evaluation of the hidden layer. Since the time taken to perform a floating point multiplication is the main indication of the time required by the feed forward procedure, we estimate the size of such operation to be the matrix multiplication of eq. 3, complexity is calculated in eq. 4.

$$O_1\big(P(N+1)(M+1)\big) \quad (4)$$

Activation function in the hidden layer is applied.

$$Z_j^p = f(z_{in_j}^p) \quad (5)$$

$f(z_{in_j}^p)$ is the activation function of the hidden layer.

Z_j^p is the jth value for the pth pattern of the hidden NN layer, j varies from 1 to M+1, therefore the size of Z is P (M+1).

$$Z_j^p = \begin{bmatrix} 1 & \cdots & 0 & 0 & \cdots & 1 \\ \vdots & \ddots & \vdots & \vdots & \ddots & \vdots \\ 1 & \cdots & 1 & 0 & \cdots & 0 \end{bmatrix}_{P(M+1)} \quad (6)$$

w_{jk}^p represents the pth pattern of the output NN weight layer, j varies from 1 to M+1 and k does from 1 to R+1, therefore the size of w is P (M+1) (R+1) and initially it is initialized with small random values.

$$w_{jk}^p = \begin{bmatrix} 8.8050 & \cdots & .00008 \\ \vdots & \ddots & \vdots \\ 0 & \cdots & .00057 \end{bmatrix}_{(M+1)(R+1)} \quad (7)$$

The offset weight w_{11}^p of the hidden layer is in eq. (7), the other values in the first column are zero.

The stimulus to the output layer is calculated as

$$y_{in_k}^p = \sum_{j=1}^{M+1} Z_j^p w_{jk}^p \quad (8)$$

Its complexity also is determined by the matrix multiplication of eq. 8.

$$O_2\big(P(R+1)(M+1)\big) \quad (9)$$

Activation function in the output layer

$$Y_k^p = f(y_{in_k}^p) \quad (10)$$

$f(y_{in_k}^p)$ is the activation function of the output layer.

Y_k^p is the actual output of the kth neuron for the pth training pattern, there is a R+1 column vector for each pattern P, for Y.

$$Y_k^p = \begin{bmatrix} 1 & \cdots & 0 & 0 & \cdots & 1 \\ \vdots & \ddots & \vdots & \vdots & \ddots & \vdots \\ 1 & \cdots & 1 & 0 & \cdots & 0 \end{bmatrix}_{P(R+1)} \quad (11)$$

T_k^p is the desired target of the kth neuron for the pth training pattern, there is a R+1 vector for all the patterns P, for T.

$$T_k^p = \begin{bmatrix} 1 & \cdots & 0 & 0 & \cdots & 1 \\ \vdots & \ddots & \vdots & \vdots & \ddots & \vdots \\ 1 & \cdots & 1 & 0 & \cdots & 0 \end{bmatrix}_{P(R+1)} \quad (12)$$

2.1 Activation Function

The activation function can be a linear or non-linear function. In this project the binary sigmoid function was used, as described in [3].

It must be analytical, continuous, differentiable, monotonically increasing, with an easy-to-calculate derivative. The function should be bounded, this is, approximate to a finite asymptotic minimum and maximum.

A binary sigmoid equation is defined as shown in eq. 13.

$$Y_k^p = f(y_{in_k}^p) = \frac{1}{1+e^{-y_{in_k}^p}} \tag{13}$$

Where $y_{in_k}^p$ is the slope of the binary sigmoid function.

In Section 3, it´s chosen the weights w_{jk}^p in such an optimal way that the outputs Y_k^p tends to targets T_k^p.

3. BACKPROPAGATION ALGORITHM AND TIME COMPLEXITY ANALYSIS

A forward propagation of stimuli and a backward propagation (BP) of error are the two parts which the BP algorithm for FFNN training procedure is divided for.

The back propagation algorithm attempts to find a minimum for the total squared error defined for one training pattern at the time. So, in BP, an input pattern is presented to the network. Based on that pattern, the network computes an output pattern. The output result is compared to a desired pattern and an error vector is computed. The error is back propagated through the network; based on the amount of inaccuracy passing through each connection, the weights are changed. After that, the next pattern is presented to the network and this procedure is repeated for the new one.

3.1 A neuron in the hidden layer

The output values of the output layer are compared with the target output values. The target output values are those that we attempt to teach to our network. The error between actual output values and target output values is calculated and propagated back towards the hidden layer. This is called the backward pass of the back propagation algorithm. The error is used to update the connection strengths between nodes, this is, weight matrices between input-hidden layers (v_{ij}^p) and hidden-output layers (w_{jk}^p) are updated. See Figure 3.

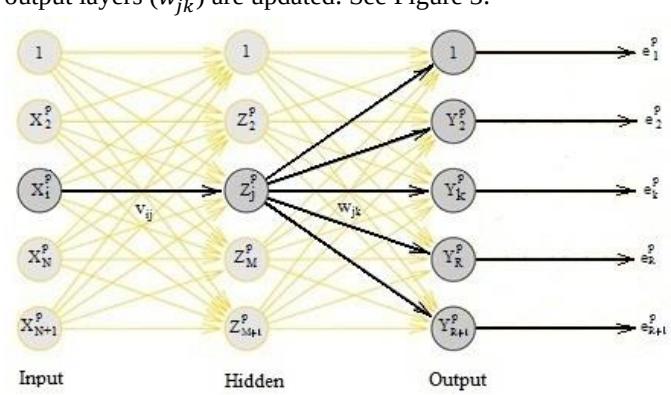

Fig. 3. Error determination

Then, the error function for this step of training is

$$e_k^p = T_k^p - Y_k^p \tag{14}$$

The overall square error function is

$$E^p = \frac{1}{2}\sum_k e_k^{p\,2} \tag{15}$$

The back propagation algorithm attempts to find a minimum for the total squared error defined for the whole training pattern. The weights are updated as stated in the following Section.

3.2 Gradient Descent

The gradient descent is a method (used in BP algorithm) that tries to minimize the error function by choosing the appropriated v_{ij} weights.

The weight update is obtained at time *t*, by the Delta rule.

$$v_{ij}(t) = v_{ij}(t-1) + \Delta v_{ij} \tag{16}$$

Where $\Delta v_{ij} = -\alpha \sum_{p=1}^{P} \frac{\partial E^p}{\partial v_{ij}}$ (17)

With α being the learning rate, it is a positive constant with range 0 < α < 1.

By the chain rule, eq. 17 can be written as eq. 18.

$$\Delta v_{ij} = -\alpha \sum_{p=1}^{P} \sum_k \frac{\partial E^p}{\partial e_k^p} \frac{\partial e_k^p}{\partial Y_k^p} \frac{\partial Y_k^p}{\partial y_{in_k}^p} \frac{\partial y_{in_k}^p}{\partial v_{ij}^p} \tag{18}$$

After doing some substitutions, the following is obtained.

$$\Delta v_{ij} = \alpha \sum_{p=1}^{P} \sum_k \left(T_k^p - Y_k^p\right) f'\left(y_{in_k}^p\right) w_{jk}^p f'\left(z_{in_j}^p\right) X_i^p \tag{19}$$

The definition of Sigma Out is in eq. (20).

$$\delta_k^p = \left(T_k^p - Y_k^p\right) f'\left(y_{in_k}^p\right) \tag{20}$$

$$\Delta v_{ij} = \alpha \sum_{p=1}^{P} \sum_k \delta_k^p w_{jk}^p f'\left(z_{in_j}^p\right) X_i^p \tag{21}$$

Defining Sigma Hidden in eq. 22.

$$\delta_j^p = \sum_k \delta_k^p w_{jk}^p f'\left(z_{in_j}^p\right) \tag{22}$$

Eq. (22) is another matrix multiplication with complexity.

$$O_3\big(P(R+1)(M+1)\big) \tag{23}$$

$$\Delta v_{ij} = \alpha \sum_{p=1}^{P} \delta_j^p X_i^p \tag{24}$$

The complexity of eq. (24) is

$$O_4\big(P(N+1)(M+1)\big) \tag{25}$$

Putting all together

$$\Delta v_{ij} = \alpha \sum_{p=1}^{P} \sum_k \left(T_k^p - Y_k^p\right) Y_k^p (1 - Y_k^p) \ldots$$
$$\ldots w_{jk}^p Z_j^p (1 - Z_j^p)\, X_i^p \tag{26}$$

Nodes in the hidden layer are organized in such a way that different nodes recognize different features of the total training space, this happens during the progression of the training. After training, when a new input pattern is supplied, units in the hidden layer will generate active outputs if such an input pattern preserves the features they (individually) learned. Conversely, if such an input pattern does not contain those known features then these units will be inclined to inhibit their outputs.

3.3 A neuron in the output layer

The output is contrasted against the desired target and an error value is computed as a function of the outcome error in each output unit. This makes up the forward phase.

The error for each unit of the output layer is.

$$E^p = \frac{1}{2} e_k^{p\,2} \tag{27}$$

$$e_k^p = T_k^p - Y_k^p \tag{28}$$

Using the Delta rule

$$w_{jk}(t) = w_{jk}(t-1) + \Delta w_{jk} \tag{29}$$

Weight change for the output layer

$$\Delta w_{jk} = -\alpha \sum_{p=1}^{P} \frac{\partial E^p}{\partial w_{jk}^p} \tag{30}$$

$$\Delta w_{jk} = -\alpha \sum_{p=1}^{P} \frac{\partial E^p}{\partial e_k^p} \frac{\partial e_k^p}{\partial Y_k^p} \frac{\partial Y_k^p}{\partial y_{in_k}^p} \frac{\partial y_{in_k}^p}{\partial w_{jk}^p} \tag{31}$$

$$\Delta w_{jk} = \alpha \sum_{p=1}^{P} (T_k^p - Y_k^p) f'\left(y_{in_k}^p\right) Z_j^p \tag{32}$$

$$\delta_k^p = (T_k^p - Y_k^p) f'\left(y_{in_k}^p\right) \tag{33}$$

$$\Delta w_{jk} = \alpha \sum_{p=1}^{P} \delta_k^p Z_j^p \tag{34}$$

Considering the matrix multiplication of eq. (34).

$$O_5\big(P(M+1)(R+1)\big) \tag{35}$$

Putting all together

$$\Delta w_{jk} = \alpha \sum_{p=1}^{P} (T_k^p - Y_k^p) Y_k^p (1 - Y_k^p) Z_j^p \tag{36}$$

The net will find an internal representation which enables it to generate appropriate responses when those patterns used during training are subsequently submitted to the network and secondly, the network will classify them according to the features they share (resemblance) with the training patterns.

The complexity of the back propagation algorithm is the addition from O_1 to O_5.

$$O\left(P(M+1)\big(2(N+1) + 3(R+1)\big)\right) \tag{37}$$

The learning process includes adjusting of weights (adaptation) in the network in order to minimize the error function. For this reason, the error obtained is propagated back from each node of the output layer to the corresponding (contributors) nodes of the intermediate layer. However each of these intermediate units receives only a portion of the total error according to its relative contribution to the current output. This process is repeated from one layer to the previous one until each node in the network receives an error signal describing its relative contribution to the total error. Based on this signal, weights are corrected in a proportion directly related to the error in the connected units. This makes up the backward phase.

4. PARALLEL BACKPROPAGATION BY PATTERN PARTITIONING

Pattern partitioning replicates the neural net structure (units, activation function and associated weights) is used by the parallel version of the BP algorithm at each *Worker* because there is no data dependency between the operations performed for different patterns; the training set is equally distributed among *Workers*. Each *Worker* performs the propagation and the adaptation phases for its local set of patterns simultaneously. Therefore, it is possible to simulate more than one network at a time and train each one with a different input pattern, in parallel. The networks are stored in a Matlab's Cell data type. These networks all have the same initial random weights and, some input patterns to learn. The weight changes are performed in parallel, after the completion of each local training pattern, they are added together when all the patterns in the training set were presented, the weights are updated, based on the total weight change computed. These weight changes are accumulated by using a special *parfor* accumulator parallel structure of the Matlab´s Parallel Computer Toolbox [8], which contains several high level programming structures that allow the applications to benefit from a distributed systems. The structures include the parallel for (*parfor*), thus, it simplifies the development of parallel code, eliminating the complexity of computing and data managing between Matlab session and the computational resources that are being used.

parfor is a structure that can automatically distribute independent tasks to multiple Matlab's *Workers* [11].

The Matlab client schedules the *parfor*, it coordinates the *Workers* contained in the parallel pool in such a way that the iterations of loops can run in parallel in the pool. The necessary data which the *parfor* operates are sent from the client to the *Workers* and the results are sent back to the client in pieces that come together.

Learning is accelerated organizing different patterns into independent tasks (work units) and running multiple tasks concurrently. These kinds of procedures with parallel tasks are consistent with the pattern partitioning technique of FFNN used in this project, essentially the equations of Sections 2-3.

The time used to process the whole learning process by the distributed system ideally will be approximately t_s/W, where t_s is the time taken by the serial algorithm for the P training set and W is the total number of *Workers*.

5. PARALLEL BACKPROPAGATION ALGORITHM PERFORMANCE

In this section a brief description of the underlying system architecture and procedures supporting the parallel learning process is presented, running on the *Workers* distributed in a computer cluster. Each *Worker* is allocated for a replicated neural network.

The network is simulated by W=24 *Workers*, which is the size of the cores in the distributed system. These 24 *Workers*, emulate 500 epochs of the 35-35-26 input-hidden-output neurons of the network for each of the P=26 patterns. It is important to keep in mind that the degree of parallelism achieved depends on the number of *Workers* assigned to the network, and the number of patterns in the training set, as well as parallel resources of the machine. For example, the 26-class problem has P=26 patterns in its training set and the total number of *Workers* assigned for the network is W=24.

5.1 Parallel learning, hardware and software support

In any parallel machine, the degree of parallelism is limited for the physical parallel resources of the machine.

The features of the six computers connected on the cluster are:

Processor INTEL i7 4770 3.4 GHz, 4 cores and 8 threads
8 GB RAM SDRAM DDR3-1600 scalable to 32 GB
Hard Disk 500 GB SATA
MATLAB R2015a

The Admin center [12] was configured with six computers in the cluster, each with four cores, with a total of W=24

5.2 Testing the Neural Network

The network should not only recognize perfectly formed letters (training data set), but also noisy versions of the letters (testing data set). Figure 4 shows a noisy version of one of the letters. Values are limited to fall between 0 and 1.

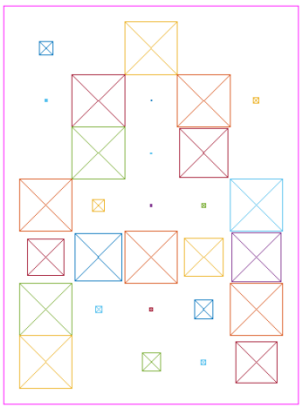

Fig. 4. Noisy version of one of the 26 patterns.

The weights obtained by the training samples can now be evaluated with the testing samples. This will be useful as a sense of how well the network will do when applied to data obtained from a noisy source.

A confusion matrix (Figure 5) is a measure of how well the neural network has fit across all the noisy samples; it shows the percentages of correct and incorrect classifications. Correct classifications are the green squares on the matrices diagonal and incorrect classifications form the pink squares.

Because the network has learned to classify properly, even with noisy input, the pink squares are cero and the diagonal has 1/27=3.7% each, indicating no misclassifications.

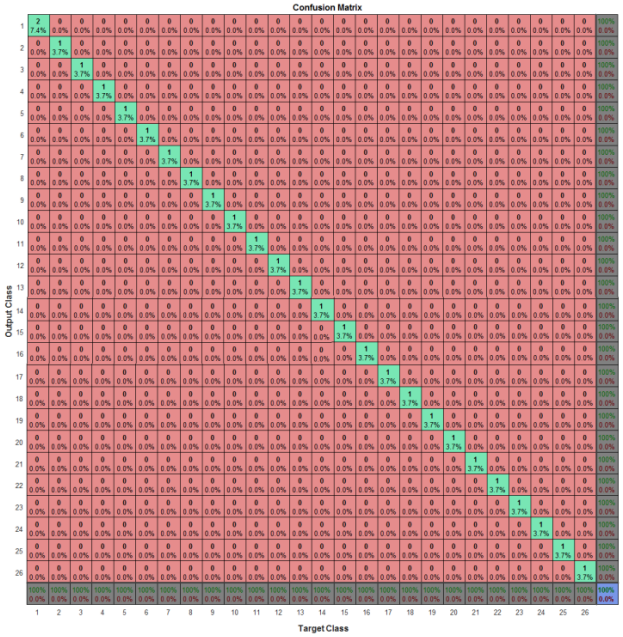

Fig. 5. Confusion matrix for the 26 alphabet pattern classification problem.

5.3 Gustafson's law

Gustafson´s law determines that with an increased number of *Workers*, bigger problems could be solved, getting better results, in the same period of time.

In Gustafson perspective [13], speedup is scaled (also called weak scaling) and represents the time required by the parallel computation divided into the time that would be required to solve the same problem with a single *Worker* and allows the problem size to be an increasing function of the number of *Workers*. This metric is useful because we usually scale up to a cluster to solve larger problems.

The speedup is based on the total time used for the computations, so it includes both the sequential and the parallel portions of the code.

The execution times were obtained with the Matlab tool called *profiler* [12].

$$t_s = 0.0763,\ t_p = 0.9237,\ W = 24$$

With t_s being the serial execution time, t_p is the parallel execution time, and W the number of *Workers*.

The serial portion of the computation in the parallel system is

$$f = \frac{t_s}{t_s + \frac{t_p}{W}} = .6647 \tag{38}$$

The maximum speedup is defined in the Gustafson's law as

$$\psi \leq f + W(1 - f) = 8.7117 \tag{39}$$

The Figure 6 shows the experimental speedup *vs*. the *Workers* involved in the PBP training algorithm and its implied linear (ideal) speedup. It also represents the optimal weights associated with the desired patterns obtained by the PBP training algorithm ran with 500 epochs and an increasing amount of *Workers* used. This performance includes the computer hardware and software, the network connection, the parallel structures and the algorithm. Therefore, the speedup curve represents all of these factors taken together. The experimental speedup curve approximately meets with the theoretical maximum scaled speedup defined by Gustafson. It means that all the previously mentioned factors work well together in this particular PBP benchmark.

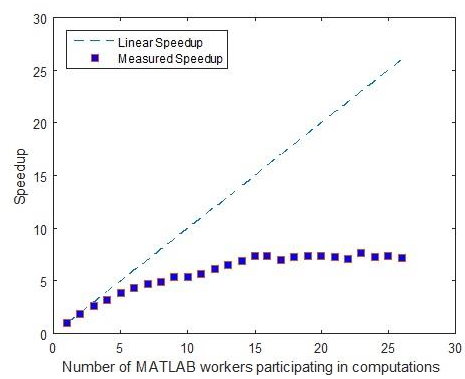

Fig. 6. Ideal and measured speedup

Another experiment to benchmark the overall performance of the whole distributed system was performed. Four networks (N_1 to N_4) were tested, with dimensions that present an increment in size, see Table 1.

The benchmarking consists of measuring the time taken for all the *Workers* to complete the training of patterns by task parallel computations (*parfor*), employing weak scaling, which consists of making the number of iterations proportional to the patterns and the number of *Workers*.

Table 1.Four NN with different sizes

Network	Input N	Hidden M	Output R	Patters P
N_1	2	4	4	4
N_2	3	8	8	8
N_3	4	16	16	16
N_4	5	32	32	32

It's used the number of floating point operations per second (Flops) as a measure of performance because that allows us to compare the performance of the PBP algorithm for different training patterns and different number of *Workers*.

The PBP has a complexity (number of Gflops) given by eq. (40), it is useful to calculate the Gigaflops computed by the parallel algorithm.

$$O\big(P(M+1)\big(2(N+1)+3(R+1)\big)/W\big)/1e^9 \quad (40)$$

Figure 7 shows the Gigaflops processed versus an increasing amount of training patterns (associated with *Workers*), as the number of *Workers* is increased, the problem size is increased proportionally. In the case of PBP, the performance of the algorithm depends greatly on the pattern's size. Figure 6 demonstrate the performance in Gigaflops for all the pattern sizes that were tested with and all the different numbers of *Workers*, which gives us the most detailed picture of the performance characteristics PBP on this particular cluster. The data shown in Figure 7 demonstrates that 32 *Workers* allow us to solve much larger PBP training than is possible with only 4 *Workers*. Additionally, it is seen that even when training with a PBP with pattern size 32 on 8 *Workers* (this means, more patterns per *Worker)* it would get a performance of approximately only .02 Gigaflops. Thus, even if the 8 *Workers* had sufficient resources to solve such problem, 32 *Workers* would greatly outperform them.

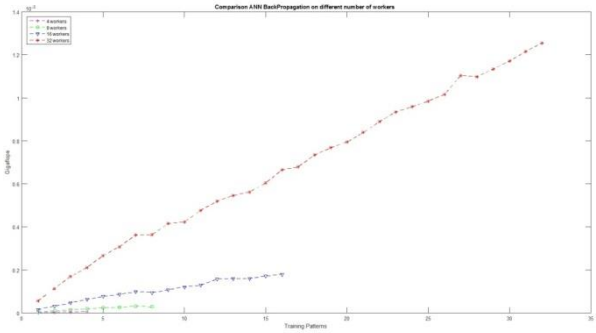

Fig. 7. Measured Gflops vs training of different patterns

6. CONCLUSIONS

Implementing neural network algorithms in parallel infrastructure is important to training bigger pattern sets in significantly shorter time. While the degree of architectural parallelism is limited to the size of the largest layer of the network, the degree of task parallelism is only limited by the number of the *Workers* and the number of training patterns, which are both often far larger than the number of the neurons in the largest layer of the network. The back propagation algorithm with pattern partitioning implemented with the *parfor* in Matlab's Parallel Computing Toolbox, presented mainly task parallelism, in the way it was parallelized in this paper, also it generated the expected theoretical and experimental performance; these elements give originality to this work in the field of study. The main contribution of this work was to organize the whole factors of the parallel infrastructure in order to work all properly. Parallel pattern partitioning implementations of neural networks training will allow us to investigate larger problems which are expected to be solved in a shorter period of time.

Acknowledgement
The authors are thankful to the TECNOLOGICO NACIONAL DE MEXICO by the support to this research, under the grant **5700.16-P**.

7. REFERENCES

[1] Yanping Bai, Haixia Zhang, Yilong Hao. 2009. The performance of the backpropagation algorithm with varying slope of the activation function. Chaos, Solutions and Fractals 40, 69–77. ScenceDirect

[2] Faramarz V., Okan K. E. A parallel implementation of backpropagation neural network on Maspar MP-1. The Purdue University MasPar MP-1 research.

[3] Amit G., Gurleen K.W. and Simranjeet K. 2012. Implementation of back propagation algorithm using Matlab. International Journal of Information Technology and Knowledge Management. July-December 2012, Volume 5, No. 2, pp. 429-431.

[4] Alfonso M., Kavka C., Printista M. 2000. Implementación paralela de algoritmo backpropagation en un cluster de computadoras. CACIC 2000.

[5] Tsaregorodtsev V.G. 2005. Parallel implementation of back-propagation neural network software on SMP computers. Institute ofcomputational modeling. Springer-Verlag Berlin Heidelberg. pp. 186-192.

[6] Suresh S., Omkar S.N., Mani V. 2005. Parallel implementation of back-propagation algorithm in networks of workstations. IEEE Transactions on parallel and distributed systems. Vol. 16, No. 1, pp. 24-34.

[7] Pethick M., Liddle M., Werstein P., Huang Z. 2000. Parallelization of a backpropagation neural network on a cluster computer. University of Otago, New Zealand.

[8] Mathworks. 2016. Parallel Computing Toolbox. [Online].

[9] Ohio Supercomputer Center. 2016. Available: https://www.osc.edu/node/306

[10] Abdi Z., Alyasseri A. 2014. Survey of Parallel Computing with MATLAB. Available on: https://arxiv.org/abs/1407.6878.

[11] Sharma G., and Martin J. 2008. "MATLAB®: A Language for Parallel Computing", Springer, INTERNATIONAL JOURNAL OF PARALLEL PROGRAMMING, Volume 37, Number 1 (2009), 3-36.

[12] Layton J. 2016. Matlab-Like Tools for HPC. Available on: http://www.admin-magazine.com/HPC/Articles/Matlab-Like-Tools-for-HPC.

[13] Michael J. Quinn, 2003. Parallel Programming in C with MPI and OpenMP, Ed. McGraw-Hill.