PARALLEL LEVENBERG-MARQUARDT-BASED NEURAL NETWORK WITH VARIABLE DECAY RATE

Tomislav Bacek, Dubravko Majetic, Danko Brezak

Mag. ing. mech. T. Bacek, University of Zagreb, FSB, I. Lucica 5, 10000 Zagreb
Prof. dr.sc. D. Majetic, University of Zagreb, FSB, I. Lucica 5, 10000 Zagreb
Doc. dr.sc. D. Brezak, University of Zagreb, FSB, I. Lucica 5, 10000 Zagreb

Keywords: neural networks, regression, parallel levenberg-marquardt algorithm

Abstract

In this paper, parallel Levenberg-Marquardt-based feed-forward neural network with variable weight decay, implemented on the Graphics Processing Unit, is suggested. Two levels of parallelism are implemented in the algorithm. One level of parallelism is achieved across the data set, due to inherently parallel structure of the feed-forward neural networks. Another level of parallelism is achieved in Jacobian computation. To avoid third level of parallelism, i.e. parallelization of optimization search steps, and to keep the algorithm simple, variable decay rate is used. Parameters of variable decay rate rule allow for compromise between oscillations and higher accuracy on one side and stable but slower convergence on the other side. To improve training speed and efficiency modification of random weight initialization is included. Testing of a parallel algorithm is performed on two real domain benchmark problems. Results, given in a form of a table with obtained speedups, show the effectiveness of proposed algorithm implementation.

1. INTRODUCTION

Artificial neural networks (NN) are used in a wide variety of applications due to their capability to learn and to generalize. Due to their simple structure and capability of nonlinear mapping of any input to any output, most widely used NNs are feed-forward NN.

Many different learning algorithms for feed-forward NNs have been reported in the literature so far. Most widely used learning algorithm has long time been gradient descent, whose poor convergence rates were significantly improved, as shown in [1], by introducing different modifications, including momentum and adaptive learning coefficient, [2], [3], [4]. Nonetheless, methods of second order, such as Gauss-Newton method, result in much faster convergence rate since they take into account information on error surface as well, [5]. Best convergence rates gives Levenberg-Marquardt (LM) method since it is a combination of simple gradient descent and Gauss-Newton method, thus taking best of both methods – stable convergence of former and fast convergence of latter method.

Although this pseudo-second order method has fast convergence rates on a small-scale problems, it proves to be very inefficient when it comes to a large-scale problems due to computational complexity, memory requirements and error oscillations, [5], [6]. In order to tackle these problems, different approaches have been suggested in the literature. Works on reduction of memory demands and computational complexity can be found in [7], [8] and [9]. Another approach is suggested in [6], where variable decay rate was introduced in order to decrease error oscillations of standard LM algorithm.

Due to the advances in computer architecture and inherently parallel nature of feed-forward NNs, parallelization of NNs was yet another approach suggested in the literature. If NN learns using batch mode, then it is possible to parallelize evaluation of objective function and its partial derivatives using simple data-parallel decomposition, [10]. Similar approach, but using MPI on .NET platform, was suggested in [11]. Suri et. al. [12] suggested parallel LM-based NNs using MPI, but in addition to simple data-parallelism, they also implemented computation of row block of the Jacobian in parallel. Three levels of parallelization using clusters are suggested in [13], where authors implemented parallelization on data sets, parallelization of the Jacobian computation and parallelization of the search steps.

Apart from using clusters to implement parallel NNs, another way is to use GPU (Graphics Processing Unit). In recent years, GPUs have rapidly evolved from configurable graphics processors to programmable, massivelly parallel many-core multiprocessors used for many general applications (hence the name GPGPU – General Purpose GPU). To our knowledge, not many implementations of parallel NNs on GPU have been proposed in the literature so far. Moreover, proposed implementations are based on one-level, data-parallelism only, [14], [15].

In order to overcome drawbacks of LM algorithm and to use huge potential of GPUs, which are nowadays easily accessible, parallel GPU imple-

mentation of the feed-forward NNs with LM algorithm is suggested. Although these networks can be parallelized on three levels, in our implementtation parallelization is achieved on two levels only – across the data sets and in Jacobian computation. Third level of parallelization, i.e. parallelization in optimization search step, is avoided by introducing variable decay rate, which significantly decreases number of unsuccessful trials and makes third level of parallelization unnecessary. The results show effectiveness of proposed parallel implementation of NN.

The remaining of the paper is organized as follows. Section II gives description of a standard sequential LM algorithm, whilst Section III discusses problems caused by fixed decay rate. Section IV describes authors' parallel implementation of LM algorithm on the GPU. In Section V simulation results are given, whereas Section VI contains concluding remarks and future work.

2. LEVENBERG-MARQUARDT ALGORITHM

The goal of the learning process of a multilayer feed-forward NN is minimization of performance index through learning parameter optimization. Learning is not stopped until the value of the performance index becomes smaller than some predefined value. The most widely used performance index, used in this paper as well, is a sum of squared errors, as follows:

$$E(\vartheta) = \boldsymbol{e}^T(\vartheta)\boldsymbol{e}(\vartheta), \tag{1}$$

where $\boldsymbol{e} = [(\boldsymbol{d}_1 - \boldsymbol{O}_1)^T \ (\boldsymbol{d}_2 - \boldsymbol{O}_2)^T \ \dots \ (\boldsymbol{d}_N - \boldsymbol{O}_N)^T]$ denotes error vector, whilst $\boldsymbol{d}_n$ and $\boldsymbol{O}_n$ denote desired and actual output vector, respectively, with $n = 1,2,\dots,N$. Number of input-output learning pairs is denoted by N, while ϑ denotes learning parameter.

When NN are trained with LM method, which is a combination of Gauss-Newton second order method and standard gradient descent method, learning parameter change $\Delta\vartheta$ at k-th iteation is given as

$$\Delta\vartheta = -[\nabla^2 E + \mu \cdot diag(\nabla^2 E)]^{-1}\nabla E, \tag{2}$$

where $\nabla^2 E$ denotes Hessian and ∇E gradient matrix, respectively. Since Hessian matrix is often hard to find, an approximation

$$\nabla^2 E = \boldsymbol{J}^T(\vartheta) \cdot \boldsymbol{J}(\vartheta), \tag{3}$$

which is valid only near-linearity of error function (i.e., where residuals are small or can be approximated by linear function), is used. Gradient of error function is given as

$$\nabla E = \boldsymbol{J}^T(\vartheta) \cdot \boldsymbol{e}(\vartheta), \tag{4}$$

where $\boldsymbol{J}$ denotes Jacobian matrix, given as

$$\boldsymbol{J}(\vartheta) = \left[\frac{\partial e_k}{\partial \vartheta_j}\right], \quad 1 \le k \le N, \qquad 1 \le j \le p, \tag{5}$$

with p being a total number of learning parameters. Since NN optimizes three different learning parameters, i.e. input-hidden and hidden-output layer weights and slope of bipolar sigmoid activation function of hidden layer neurons, three different Jacobian matrices need to be calculated.

Pseudo-code of sequential LM algorithm is given as follows:

1. initialize all learning parameters and set μ to some small value, e.g. $\mu = 0.01$
2. compute the sum of squared errors E over all training patterns
3. compute Jacobian matrices
4. compute increments of learning parameters using Eq. (2)
5. recompute $E(\vartheta)$ using $\vartheta + \Delta\vartheta$, and then
   IF $E(\vartheta + \Delta\vartheta) < E(\vartheta)$ in Step 2
   $\vartheta = \vartheta + \Delta\vartheta$
   $\mu = \mu \cdot \beta \quad (\beta = 0.1)$
   go to Step 2
   ELSE
   $\mu = \mu/\beta$
   go to Step 4
   END IF

Starting from some initial set of learning parameters ϑ_0, LM algorithm iteratively proceeds down the slope of the error function, ultimately finding some local minimum of that function. This is done by trying different sets of parameters generated by altering parameter μ. If error function increases, quadratic approximation of the error curve is unsatisfactory and μ is increased by factor $1/\beta$, thus making gradient method dominant in the adaptation of learning parameters. If, on the other hand, error function decreases, quadratic approximation is good and μ is decreased by decay rate β, thus making Gauss-Newton method dominant in learning parameters adaptation. First set of parameters that leads to decrease in error function is set to be the new set of parameters which will be used as the initial set in the next iteration of the algorithm.

3. VARIABLE DECAY RATE

Although LM algorithm decreases error in every iteration, every iteration has a potential to become computationally very intensive if many unsuccessful trials need to be performed before new set of parameters is found. Number of unsuccessful trials is directly related to the decay rate β, since the step direction and the step size of each trial depend on parameter μ, which is a function of β. Initial suggestion by Marquardt [16] to use the same decay rate in both cases, i.e. regardless whether error function is decreased or increased (with reciprocal value of decay rate used in latter

case), was used by many researchers [13], [17], [18], but this strategy didn't give good results. Another strategy was proposed in [1], where different decay rates were used depending whether error function was increased or decreased. Although better results were obtained, number of unsuccessful trials was still significant, resulting in slow learning process in a case of a large network.

Another approach is considered in [6]. As authors showed in their paper, speed of convergence of LM algorithm slows down when approaching required accuracy due to many error oscillations. Authors also show that, by fixing decay rate to some value (usually 0.1), leads to oscillations in μ, which implies that many trials in decreasing μ by multiplying β would not lead to reduction in error but cause unexpected ascend of error and therefore, waste time.

To overcome this drawback, authors suggested log-linear function as a rule of varying decay rate β after each iteration, where β is given as a function of error. In this paper, we suggest different rule of varying decay rate, as follows

$$\beta_{dec} = B_1 - B_2 \cdot e^{-\rho(\Delta E_s)}, \quad (6)$$

with ΔE_s given as

$$\Delta E_s = \frac{E - E_{min}}{E_0 - E_{min}}, \quad (7)$$

where E is reduced sum of squared error, E_{min} is the required training accuracy and E_0 is first calculated error based on initial learning parameters. Parameters B_1 and B_2 should be chosen in interval $[0.1,0.9]$. Figure 1. shows decay rate rule graphically, when $B_1 = 0.9$ and $B_2 = 0.1$.

The idea behind above suggested rule is as follows. When network is far from desired accuracy and approaches minimum, decay rate is slowly decreased, thus slowly increasing step size and blending more towards Gauss-Newton method. This ensures stability and good convergence properties at the beginning stages of learning process. At the subsequent stages of learning process, when network is closer to the minimum and error is decreased, decay rate changes faster, thus blending more and more towards Gauss-Newton method. This ensures faster convergence to the minimum, but has a side effect of causing oscillations near minimum. Nonetheless, these oscillations are much less expressed than in the case of fixed decay rate, thus speeding up learning process.

It should also be noted that, in this paper, different decay rate is used in the case of error decrease and increase. When error is decreased, μ is multiplied by β_{dec}. On the other hand, when error is increased, μ is multiplied by fixed β_{inc} (usually 100), which ensures much faster convergence than when $1/\beta_{dec}$ is used.

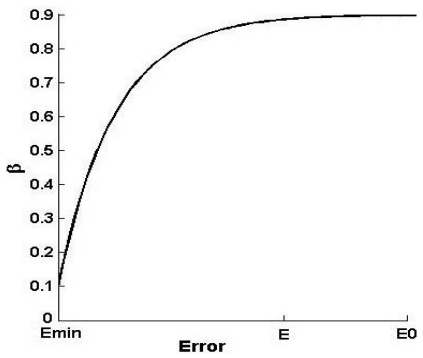

Figure 1. Rule of decay rate variation

4. PARALLELIZATION OF THE LEVENBERG-MARQUARDT ALGORITHM

As mentioned before, LM-based NN was parallelized on two levels – across data sets and in Jacobian computation. Since NNs with LM algorithm learn using batch mode, it is possible to decompose objective function in such a way that each GPU unit calculates objective function for one input-output learning pair. In other words, it is possible to separate learning patterns (input-output pairs) into disjoint sets and then perform all necessary operations on each learning pattern in parallel. This type of parallelism is known as SIMD (Single Instruction Multiple Data). After obtaining outputs for each learning pattern and calculating local objective functions in parallel, local errors are gathered and summed up, after which algorithm continues with execution of the subsequent operations.

The second level of parallelism is within the calculation of the Jacobian matrix. Since only regression problems are considered in this paper, NN has only one output, i.e each row in Jacobian matrix, given in Eq .(8), corresponds to one input-output pair. Each row of each Jacobian matrix is thus calculated in parallel, and stored into appropriate matrix. After all learning patterns are processed, obtained Jacobians are used to calculate Hessians, which are then used to determine learning parameter change according to Eq. (2).

Calculation of Hessian, although parallelizable, is not done in parallel since it involves only matrix-matrix multiplication, an operation performed extremely fast on the GPU unit. Furthermore, since our algorithm is implemented in Matlab, which is well known to, in general, give much better results when the code is given in vectorized form, Hessian computation is not parallelized. Two levels of parallelization suggested in this paper were accomplished using Accelereyes' Jacket, platform that enables single-threaded M-code to be transformed to GPU-enabled applications.

Third level of parallelization, i.e. parallelization of the search step, suggested in [13], is not implemented in this paper. Initial LM algorithm with fixed decay rate, as suggested in [16], often performs

many unsuccessful optimization search steps, which significantly slows down the convergence. In this paper, instead of parallelizing search steps and finding an optimal way to choose between good search steps if more than one is found, we suggest using variable decay rate. If decay rate is given as an exponential function of error, number of unsuccessful search steps is significantly decreased, thus making third level of parallelization unnecessary, and simplifies algorithm implementtation on the GPU unit. Changing parameters B_1 and B_2 in (6) in suggested interval, different shapes of variable decay rate are obtained, which directly influences number of unsuccessful search steps.

$$J(\vartheta) = \begin{bmatrix} \frac{\partial e_1}{\partial \vartheta_1} & \frac{\partial e_1}{\partial \vartheta_2} & \cdots & \frac{\partial e_1}{\partial \vartheta_p} \\ \frac{\partial e_2}{\partial \vartheta_1} & \frac{\partial e_2}{\partial \vartheta_1} & & \frac{\partial e_2}{\partial \vartheta_p} \\ \vdots & & \ddots & \vdots \\ \frac{\partial e_N}{\partial \vartheta_1} & \frac{\partial e_N}{\partial \vartheta_1} & \cdots & \frac{\partial e_N}{\partial \vartheta_p} \end{bmatrix}. \quad (8)$$

5. SIMULATION RESULTS

All tests are carried out using 3-layered feed-forward NN. Number of input and output layer neurons depends on the benchmark problem, while number of hidden layer neurons is arbitrary. In this paper, three different NN architectures are used. Learning process was carried out using 1000 steps during which network was validated after every 10 steps for there is no guarantee that the validation error will have strictly decreasing manner as learning proceeds. If validation error decreased compared to a previous one, learning parameters were saved. Otherwise, they were not considered.

In order to improve learning process, a modification of random learning parameters initialization, proposed in [19], is used:

$$\vartheta_0 = 0.7\, H^{\frac{1}{M}}(-1 + 2 \cdot rand), \quad (9)$$

where H and M denote the number of neurons in layers connected with parameter ϑ – former refers to succeeding and latter to preceding layer.

Algorithms are compared using speedup measure. Speedup shows computational advantage gained by using GPU over the amount of computation needed by the same algorithm on the CPU. Speedup S can be calculated as follows:

$$S = \frac{T_{CPU}}{T_{GPU}}, \quad (10)$$

where T_{CPU} denotes execution time on the CPU and T_{GPU} denotes execution time on the GPU.

In all the tables, *HLNs* denotes hidden layer neurons and $NRMS_{min}$ denotes minimum *NRMS* error, as given in [1], achieved on validation set. Furthermore, $NRMS_{min}$ *step* denotes iteration at which this error is achieved.

5.1. Nonlinear system prediction

In their paper on nonlinear signal processing, Lapedes and Farber [20] suggested Glass-Mackey chaotic system as a NN benchmark problem due to its simple definition but hard to predict behavior. This system is given in discrete time as

$$x(n) = \frac{1}{1+b}\left[x(n-1) + \frac{a \cdot x(n-\tau)}{1 + x^{10}(n-\tau)}\right]. \quad (11)$$

In this paper, $a = 0.2$ and $b = 0.1$. Sampling time is set to $T_0 = 1s$, and time delay $\tau = 30$.

The goal of the algorithm is to predict the behavior of chaotic system in P-th point in the future based on m past points and the current one. Standard method for this kind of prediction is to determine a mapping function $f(\bullet)$ as follows:

$$x(n+P) = f\big(x(n), x(n-\Delta), \ldots, x(n-m\Delta)\big). \quad (12)$$

In this paper, $P = \Delta = 6$ and $m = 4$, which leads to the following mapping function

$$x(n+6) = f\big(x(n), x(n-6), \ldots, x(n-24)\big). \quad (13)$$

Figure 2. shows time series of 1000 time steps for $\tau = 30$ (time in units of τ). First 500 time steps were used for learning, whereas remaining 500 were used for validation of an algorithm.

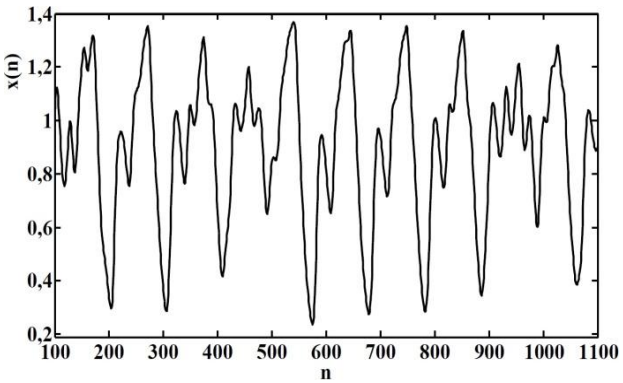

Figure 2. Glass-Mackey time series

Task of a NN is to predict a single value of a system based on five known signal values, so the network has $6 - J - 1$ structure (bias neuron is added to both input and hidden layer), where the number J of hidden layer neurons is arbitrary. In this paper, $J = 13, 23, 33$.

Learning and validation results of chaotic system prediction problem with fixed and variable decay rate are given in Table 1 and Table 2, respectively. It can be seen that in both cases networks give similar results, but networks with fixed decay rate perform, in total, much more unsuccessful search steps. Actual improvements introduced with variable decay rate can be seen in Table 3, which shows time and epochs needed for both networks to reach the same *NRMS=0.079* error.

Table 1. Chaotic system learning and validation results (fixed β)

No. of HLNs	13	23	33
$NRMS_{min}$ step	520	510	330
$NRMS_{min}$	0.0784	0.0743	0.0757
No. of trials	4003	2501	4004

Table 2. Chaotic system learning and validation results (variable β)

No. of HLNs	13	23	33
$NRMS_{min}$ step	500	510	560
$NRMS_{min}$	0.0790	0.0757	0.0765
No. of trials	1470	1706	1392

Table 3. Processor time and trials needed to achieve NRMS=0.079 on prediction problem

No. of HLNs	NN with fixed β		NN with variable β	
	Time[s]	Trial	Time[s]	Trial
13	13.98	1998	8.68	727
23	5.27	475	6.73	499
33	8.89	599	7.59	478

Speedups obtained with two levels of parallelism and variable decay rate are given in Table 4. Speedups are not significant in this case due to the fact that given problem is not sufficiently large in size to utilize all the computational power of the GPU. Negative correlation between the size of the network and speedups is due to the computationally intensive inversion of Hessian which, in this paper, is not parallelized, i.e. it is performed in a standard way, using LU factorization.

Table 4. Chaotic system prediction speedups

No. of HLNs	13	23	33
Speedup S	4.05	3.56	3.48

5.2. Filtration of estimated tool wear curves

Machine tool wear estimation is of a high importance in machining processes since every fifth machine downtime is caused by an unexpected tool wear. To fulfill high demands on reliability and robustness, a new tool wear regulation model is proposed in [21]. Data used therein for testing proposed model, which was obtained experimentally, will be used in this paper as well.

Simulated flank wear curves used in NN's learning process are shown in Figure 3. On the other hand, in real conditions estimation error is influenced by different disturbances that can occur during machining process. In order to capture real conditions, white noise is added to simulated model outputs.

Figure 4. shows simulated (VB_{id}) and estimated ($\widetilde{VB}^E$) curves, as well as filtrated ($\widetilde{VB}$) curve, which represents desired output of a NN. Therefore, the goal of a NN is to generate, based on four previous values obtained from the estimator, an output which will be similar to the filtrated curve shown in Figure 4.

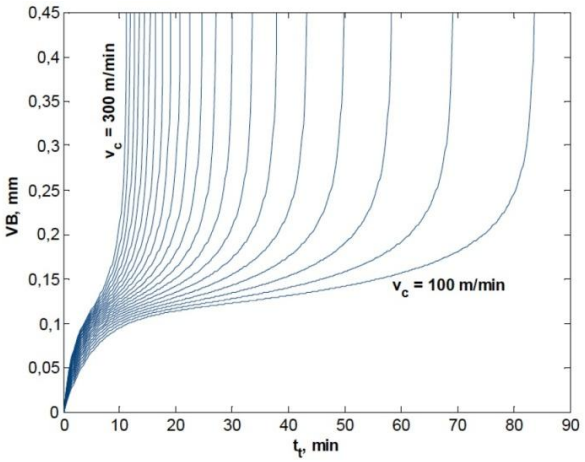

Figure 3. Flank wear curves

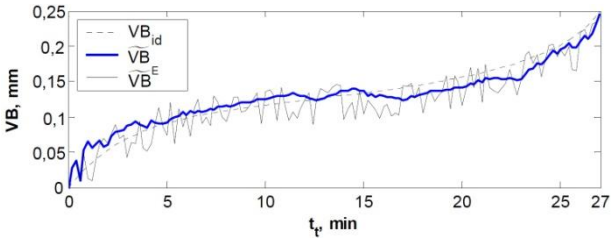

Figure 4. Model and NN output

The goal of the learning process is filtration of the estimated tool wear curves. Since NN has to estimate only one value based on a five known values, network has again $6 - J - 1$ structure.

Table 5 and Table 6 show learning and validation results on filtration of estimated tool wear curves in a case of fixed and variable decay rate, respectively. It can be seen that variable decay rate led to significant decrease in number of unsuccessful search steps.

Table 5. Learning and validation results on filtration of estimated tool wear curves (fixed β)

No. of HLNs	13	23	33
$NRMS_{min}$ step	320	340	150
$NRMS_{min}$	0.2700	0.2660	0.2696
No. of trials	4003	2501	2500

Table 7 shows actual improvements introduced with variable decay rate in the case of filtration of estimated tool wear curves, while Table 8 shows speedups in the same case. Since filtration problem is much bigger than prediction problem, speedups are significantly bigger, thus showing effectiveness of proposed algorithm. As said before, negative correlation between network size and speedups is due to the non-parallelized inversion of Hessian matrix, which becomes computationally intensive on the GPU as the size of a network increases.

Table 6. Learning and validation results on filtration of estimated tool wear curves (variable β)

No. of HLNs	13	23	33
$NRMS_{min}$ step	320	320	160
$NRMS_{min}$	0.2674	0.2646	0.2689
No. of trials	1864	1318	1459

Table 7. Processor time and trials needed to achieve NRMS=0.27 on filtration problem

No. of HLNs	NN with fixed β		NN with variable β	
	Time[s]	Trial	Time[s]	Trial
13	21.42	652	13.99	345
23	24.97	652	7.75	120
33	21.15	377	16.85	213

Table 8. Filtration of estimated tool wear curves speedups

No. of HLNs	13	23	33
Speedup S	16.36	14.06	12.88

6. CONCLUSION

GPU-based parallel implementation of LM-based NN with variable weight decay is proposed. Two levels of parallelism are implemented in the algorithm – parallelization across data sets and parallelization of Jacobian computation. In order to avoid parallelization of search steps, exponential variable decay rule is suggested. Simulation results show effectiveness of proposed algorithm, both in variable decay rate and parallel GPU implementation.

Future work will be oriented towards CUDA implementation of algorithm and parallelization of Hessian matrix inversion.

7. REFERENCES

[1] Brezak, D., Bacek, T., Majetic, D., Kasac, J., Novakovic, B., 2012, A Comparion of Feed-forward and Recurrent Neural Networks in Time Series Rorecasting, Conference Proceedings IEEE-CIFEr, pp. 119-124

[2] Zurada, J.M., 1992, Artificial Neural Systems, W.P. Company, USA

[3] Pearlmutter, B., 1991, Gradient descent: Second order momentum and saturating error, NIPS 2, pp. 887-894

[4] Tollenaere, T., 1990, SuperSAB:Fast adaptive backpropagation with good scalling properties, Neural Networks 3, pp. 561-573

[5] Hagan, M.T., Menhaj, M.B., 1993, Training Feedforward Networks with the Marquardt Algorithm, IEEE T Neural Networ 5, pp. 989-993

[6] Chen, T., Han, D., Au, F., Tham, L., 2003, Acceleration of Levenberg-Marquardt training of neural networks with variable decay rate, In Proc. of IJCNN '03., Vol. 3

[7] Zhou, G., Si, J., 1998, Advanced neural-network training algorithm with reduced complexity based on Jacobian deficiency, IEEE Trans. on Neural Net., vol. 9, no. 3, pp. 448-453

[8] Chan, L.W., Szeto, C.C., 1999, Training recurrent network with block-diagonal approximated Levenberg-Marquardt algorithm, In Proc. of IJCNN, vol. 3, pp. 1521-1526

[9] Wilamowski, B.M., Chen, Y., Malinowski, A., 1999, Efficient algorithm for training neural networks with one hidden layer, Proc. IJCNN, vol. 3, pp. 1725-1728

[10] Daniel, R., 1990, Parallel nonlinear optimization, Proc. Fifth Distributed Memory Computing Conf. (DMCC5), Charleston, SC

[11] Lotric, U., Dobnikar, A., 2005, pp. Parallel implementation of feed-forward neural network using MPI and C# on .NET platform, Proc. of International Conference on Adaptive and Natural Computing Algorithms, 534-537

[12] Suri, R., Deodhare, D., Nagabhushan, P., 2002, Parallel Levenberg-Marquardt-based Neural Network Training on Linux Clusters – A Case Study, India

[13] Cao, J., Novstrup, K., Goyal, A., Midkiff, S., Caruthers, J., 2009, A parallel Levenberg-Marquardt algorithm, ICS '09, New York, USA

[14] Prahbu, R.G., 2003, Gneuron:Parallel Neural Networks with GPU, International Conference on High Performance Computing, Hyderabad, India

[15] Kajan, S., Slacka, J., 2010, Computing of Neural Network on Graphics Card, International Conference Technical Computing, Bratislava

[16] Marquardt, D., 1963, An algorithm for least squares estimation of nonlinear parameters, J. Soc. Ind. Appl. Math, pp. 431-441

[17] Wilamowski, B. M. ,Iplikci, S., Kaynak, O., Efe, 2001, An algorithm for fast convergence in training neural networks, In Proc. of IJCNN '01., Vol. 3, pp. 1778-1782

[18] Suri, N. N., Deodhare, D., Nagabhushan, P., 2002, Parallel Levenberg-Marquardt-based neural network training on Linux clusters – A case study, ICVGIP 2002

[19] Nguyen, D., Widrow, B., 1990, Improving the Learning Speed of 2-Layer Neural Networks by Choosing Initial Values of the Adaptive Weights, In Proc. of the International Joint Conference on Neural Networks, San Diego, CA, USA, vol. 3, pp. 21-26

[20] Lapedes, A., Farber, R., 1987, Nonlinear Signal Processing Using Neural Networks: Prediction and System Modeling, Techical Report, Los Alamos, New Mexico

[21] Brezak, D., Majetic, D., Udiljak, T., Kasac, J., 2010, Flank Wear Regulation using Artificial Neural Networks, JMST 24(5), pp. 1041-10