2014 International Joint Conference on Neural Networks (IJCNN)
July 6-11, 2014, Beijing, China

GPU Implementation of the Feedforward Neural Network with Modified Levenberg-Marquardt Algorithm

Bacek Tomislav, Dubravko Majetic and Danko Brezak

***Abstract*— In this paper, an improved Levenberg-Marquardt-based feedforward neural network, with variable weight decay, is suggested. Furthermore, parallel implementation of the network on graphics processing unit is presented. Parallelization of the network is achieved on two different levels. First level of parallelism is data set level, where parallelization is possible due to inherently parallel structure of the feedforward neural networks. Second level of parallelism is Jacobian computation level. Third level of parallelism, i.e. parallelization of optimization search steps, is not implemented due to the variable weight decay, which makes third level of parallelism redundant. Suggested weight decay variation enables the compromise between higher accuracy with oscillations on one side and stable, but slower convergence on the other. To improve learning speed and efficiency, modification of random weight initialization is included. Testing of proposed algorithm is performed on two real domain benchmark problems. The results obtained and presented in this paper show effectiveness of proposed algorithm implementation.**

I. Introduction

ARTIFICIAL NEURAL NETWORKS (NN) are non-parametric mathematical models of their biological counterparts. Due to their capability to learn and to generalize on previously unseen data, NNs are used in a wide variety of applications. Although there are many different NNs reported in the literature, the most widely used are feedforward NNs, due to their simple structure and ability to perform nonlinear mapping of any given input to any desired output.

Feedforward NNs use the so-called error back propagation mechanism to adjust learning parameters until stopping criterion is met. Learning parameters can be changed in many different ways, depending on whether local or global information on error surface is used, as well as whether information on error surface curvature is used or not. Most widely used learning algorithm has long time been simple gradient descent (GD), whose poor convergence rates were significantly improved [1] by introducing different modifications, including momentum and adaptive learning coefficients [2][3][4]. Nonetheless, it has been shown that methods of second order, such as Gauss-Newton (GN) method, result in much faster convergence rate since those methods take into account information on error surface as well [5]. One method that gained particular interest in solving non-linear least square problems is Levenberg-Marquardt method (LM), which interpolates between GD and GN methods. In this way, LM takes good properties of both methods - fast convergence of GN and stable convergence of GD method.

Although the idea behind LM algorithm seems promising in theory, it has a serious drawback in practice. When used on a small-scale problems, this method proves to have very fast convergence rates, but when it comes to large-scale problems, LM becomes very inefficient 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 the reduction of memory demands and computational complexity can be found in [7][8][9]. Another approach was suggested in [6], where variable weight decay rate was introduced in order to decrease error oscillations of standard LM algorithm. Some researchers succesfully implemented online version of LM algorithm as well [10][11], where Hessian matrix is approximated recursivelly, depending on current objective function estimate using current input-output pair.

Due to advances in computer architecture and inherently parallel nature of feedforward NNs, parallelization of NNs is yet another approach suggested in the literature. In the case of batch learning mode, objective function is simply a sum of all its local values, which correspond to each input-output learning pair. Therefore, it is possible to parallelize evaluation of the objective function and its partial derivatives using simple data-parallel decomposition [12]. Similar approach was suggested in [13], where MPI on .NET platform was used. Suri et. al. [14], on the other hand, suggested parallel LM-based NNs using MPI with two levels of parallelism. One level of parallelism was data-parallelism, whereas another one was parallelization of Jacobian row block computation. Three levels of parallelization, using clusters, are suggested in [15], where authors showed implementation of parallelization on data sets, parallelization of the Jacobian computation and parallelization of search steps.

Parallel implementation of NNs can be also obtained by using Graphics Processing Unit (GPU). In recent years, GPUs have evolved from configurable graphics processors to programmable, massivelly parallel many-core multiprocessors used for many general applications (GPGPU - General Purpose GPU). To our knowledge, only few 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 [16][17].

In order to overcome drawbacks of LM algorithm and, at the same time, utilize huge potential of GPUs, which are nowadays easily accessible, parallel GPU implementation of the feedforward NN with LM algorithm is suggested

Bacek Tomislav, Dubravko Majetic and Danko Brezak are with the Department of Robotics and Production System Automation, University of Zagreb, Croatia (email: {tomislav.bacek, dubravko.majetic, danko.brezak}@fsb.hr).

978-1-4799-1484-5/14/$31.00 ©2014 IEEE

and evaluated. As mentioned above, such networks can be parallelized on three levels, but in this paper, only two levels of parallelization are implemented, namely parallelization across data set and parallelization of Jacobian computation. Third level of parallelization, i.e. parallelization accross the optimization search steps, is made redundant by introducing variable weight decay rate, which decreases the number of unsuccessful trials and makes the algorithm to converge not only faster, but also finding better local minimum. The results of case studies show effectiveness of proposed parallel implementation of NN with LM algorithm.

The remaining of the paper is organized as follows. Section II gives description of standard, sequential LM algorithm, whilst Section III discusses problems caused by fixed weight decay and presents suggested function for varying it. Section IV describes our parallel implementation of NN with LM algorithm on the GPU. In Section V simulation results are given, whereas Section VI contains concluding remarks and future work.

II. Levenberg-Marquardt algorithm

The goal of the learning process of a multilayer feedforward NN is minimization of the objective function through learning parameter optimization. Usually, learning is not stopped until the value of the objective function becomes smaller than some predefined value. In this paper, predefined number of learning iterations is chosen as a stopping criterion.

Given a set of input-output learning pairs, NN's learning problem is formulated as follows:

$$\theta = arg\min_{\theta} V_N(\theta), \tag{1}$$

where $V_N(\theta)$ is objective function to be minimized, given as

$$V_N(\theta) = \mathbf{e}^T(\theta)\cdot\mathbf{e}(\theta), \tag{2}$$

where $\mathbf{e}(\theta) = [(\mathbf{D}_1 - \mathbf{O}_1)^T \quad (\mathbf{D}_2 - \mathbf{O}_2)^T \ \ldots \ (\mathbf{D}_N - \mathbf{O}_N)^T]^T$ denotes error vector, whilst $\mathbf{D}_n$ and $\mathbf{O}_n$ denote desired and actual output vector, respectively, with $n = 1,2,\ldots,N$. Number of input-output learning pairs is denoted by N, while θ denotes learning parameter. This way, NN's learning process comes down to non-linear least squares problem.

Iterative optimization procedure, which is used to find θ (1), is given in general form as

$$\theta^{k+1} = \theta^k - \mu_k \mathbf{d}_k, \tag{3}$$

where μ is step lenght that guarantees a decrease of the criterion (2) in each iteration. Step direction $\mathbf{d}_k$ is given by

$$\mathbf{d}_k = \mathbf{R}_k^{-1}\nabla V_N(\theta^k), \tag{4}$$

where $\mathbf{R}_k$ is a matrix that modifies the search direction obtained by gradient descent, given as

$$\nabla V_N(\theta^k) = \mathbf{e}'^T(\theta)\cdot\mathbf{e}(\theta) = \mathbf{J}^T(\theta)\cdot\mathbf{e}(\theta), \tag{5}$$

where $\mathbf{J}(\theta)$ denotes Jacobian matrix, defined as

$$\mathbf{J}(\theta) = \left[\frac{\partial e_k}{\partial \theta_j}\right], \quad 1 \le k \le N, \quad 1 \le j \le p, \tag{6}$$

with p being a total number of learning parameters.

When NNs are trained with LM algorithm, which is possible due to a modification of backpropagation given in [5], parameter μ is set to 1, while matrix $\mathbf{R}_k$ is given as

$$\mathbf{R}_k = \mathbf{J}_k^T\cdot\mathbf{J}_k + \lambda_k diag(\mathbf{J}_k^T\cdot\mathbf{J}_k). \tag{7}$$

Term $\mathbf{J}_k^T\cdot\mathbf{J}_k$ is used to approximate Hessian matrix and as such, it is valid only near-linearity of the objective function (i.e. where residuals are small or can be approximated by linear funtion).

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

1) initialize all learning parameters and set λ to some small value, e.g. λ = 0.01
2) compute objective function
3) compute Jacobians
4) compute new learning parameters using (3)
5) recompute objective function using new learning parameters
   IF $V_N(\theta+\Delta\theta) < V_N(\theta)$ in Step 2
      $\theta = \theta + \Delta\theta$
      $\lambda = \lambda\cdot\beta$ ($\beta = 0.1$)
      go to Step 2
   ELSE
      $\lambda = \lambda/\beta$
      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 objective (error) function, ultimately finding some local minimum of that function. This is achieved by trying different sets of learning parameters generated by altering parameter λ. If increase in the objective function is observed, quadratic approximation of the error curve is unsatisfactory and λ needs to be increased by a factor $1/\beta$, thus bringing the algorithm closer to the GD direction in the adaptation of learning parameters. If, on the other hand, decrease in the objective function is observed, quadratic approximation is good and λ needs to be decreased by a weight decay β, thus bringing the algorithm closer to GN algorithm, which then becomes dominant in learning parameters adaptation. An iteration is finished after the first set of learning parameters, that leads to decrease in the objective function, is found. Learning parameters that led to decrease in the objective function are set to be the new set of parameters, which will be used as the initial set in the next iteration of the algorithm.

III. Variable weight decay rate

An iteration of LM algorithm, unlike iterations of other commonly used algorithms, such as GD, conjugate gradient, resillient bacpropagation, just to name few, always results in objective function decrease. This distinctive property, as much as it is desirable, can also lead to heavy computational load. This is due to the fact that the algorithm, before moving on to the next iteration, will do as many trials with new parameters as neccessary, until it finds a set of parameters

that lead to objective function decrease. The number of unsuccessful trials is directly related to weight decay β, since parameter λ, which determines step size and direction, is a function of weight decay, i.e. $\lambda = \lambda(\beta)$.

Initial suggestion by Marquardt [18] to use the same weight decay in both cases, i.e. regardless whether objective function is decreased or increased (with reciprocal value of weight decay used in latter case), was used by many researchers [14][15][19], but this strategy didn't give good results. Another strategy was proposed in [1], where different weight decay was used depending on whether objective function increase or decrease was observed. Although better results were obtained, number of unsuccessful trials was still significant, resulting in slow learning process in a case of a large learning problem.

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

Led by above mentioned observations, authors suggested log-linear function as a rule of varying weight decay β after an iteration. Their heuristic function was based on the reasoning that, at the beginning of a learning process, when objective function value is much greater than desired accuracy, repeated trials with decreasing λ by multiplying $\beta = 0.1$ should be performed. On the other hand, at subsequent stages of learning, as objective function approaches the neighborhood of minimum, λ should be decreased by multiplying $0.1 < \beta < 0.9$, because at this stage stable convergence is needed to avoid oscillations.

In this paper, we suggest different heuristic rule of varying weight decay, as follows:

$$\beta_{dec} = B_1 - B_2 \cdot e^{-\rho \Delta E_s}, \tag{8}$$

with ΔE_s given as

$$\Delta E_s = \frac{E - E_{min}}{E_0 - E_{min}}, \tag{9}$$

where E_0 and E are first calculated and reduced error, respectively, and E_{min} is the required training accuracy. Parameters B_1 and B_2 should be chosen in the interval $[0.1, 0.9]$, to keep LM algorithm converge. Choosing different values of B_1 and B_2 allows for compromise between oscillations and higher accuracy on one side and stable, but slower convergence on the other. In this paper, in order to achieve high efficiency learning, all the test were carried out with $B_1 = 0.9$, $B_2 = 0.8$ and $\rho = 100$. Fig. 1 shows suggested weight decay rule graphically.

The idea behind above suggested rule comes from the fact that LM algorithm can be seen as GN algorithm using a trust region approach. Since the GN algorithm relies on a quadratic Taylor expansion of $V_N(\theta)$ at θ, which is good

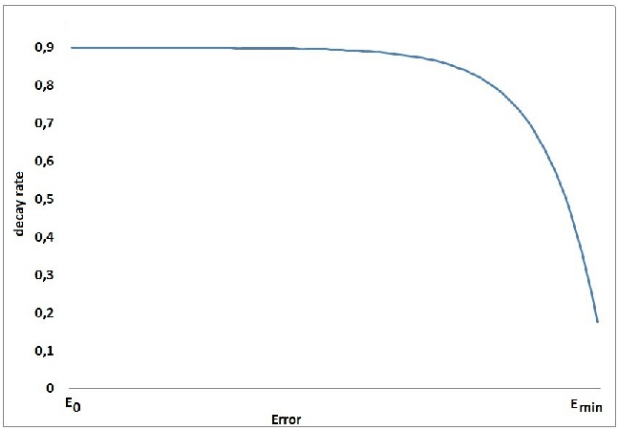

Fig. 1. Rule of weight decay variation

only in the neighborhood of θ, LM algorithm will converge well only if the size of that region, which is known as trust region, is changed in optimal way. A way to change trust region size is to alter parameter λ.

At the beginning of a learning process, while network is far from minimum, parameter λ should be changed slowly, since it is unlikely that objective function is quadratic at that stage. This can be achieved by multiplying λ by $\beta \leq 0.9$, thus slowly moving LM algorithm towards GN algorithm. As learning process proceeds and network approaches minimum, it is more likely that quadratic approximation of objective function will be satisfactory, so at that stage parameter λ, which determines to which extent will gradient descent direction be altered, should be changed faster. This is done by multiplying it by $0.1 \leq \beta < 0.9$, thus moving LM algorithm more and more towards GN algorithm. Such strategy will not significantly decrease network oscillations compared to the case of network with fixed weight decay, but it will improve network's performance at the earlier stages of learning. This proves to be sufficient for the proposed LM algorithm to outperform other modifications of sequential LM algorithm.

It should also be noted that, in this paper, different decay rate is used in the case of error decrease and increase. When error was decreased, λ was multiplied by suggested variable β_{dec} (8). On the other hand, when error was increased, λ was multiplied by fixed $\beta_{inc} = 10$, which ensures much faster convergence than when $1/\beta_{dec}$ is used.

IV. Parallelization of Levenberg-Marquardt algorithm

As mentioned before, LM-based NN was parallelized on two levels - across data sets and in Jacobian computation. Since NNs tested in this paper are trained in 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 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 Jacobian matrices. Three different Jacobians need to be calculated - one for hidden weights, one for output weights and one for sigmoidal activation function slope. Sizes of Jacobians are, respectively, $[N \times (N_{in} \cdot N_{hn})]$, $[N \times N_{hn}]$ and $[N \times (N_{hn} - 1)]$, where N_{in} and N_{hn} denote number of input and hidden layer neurons (including bias), respectively. Since, in this paper, NNs have only one output, each row in Jacobian matrix (10) corresponds to one input-output learning pair and is calculated in parallel and stored into appropriate matrix. After all learning patterns are processed, Jacobians are used to approximate Hessians (7).

$$\mathbf{J}(\theta) = \begin{bmatrix} \frac{\partial e_1}{\partial \theta_1} & \frac{\partial e_1}{\partial \theta_2} & \cdots & \frac{\partial e_1}{\partial \theta_p} \\ \frac{\partial e_2}{\partial \theta_1} & \frac{\partial e_2}{\partial \theta_2} & \cdots & \frac{\partial e_2}{\partial \theta_p} \\ \vdots & \vdots & & \vdots \\ \frac{\partial e_N}{\partial \theta_1} & \frac{\partial e_N}{\partial \theta_2} & \cdots & \frac{\partial e_N}{\partial \theta_p} \end{bmatrix}. \tag{10}$$

Calculation of Hessian is not done in parallel since i) it involves only matrix-matrix multiplication, an operation performed extremely fast on the GPU and ii) our algorithm is implemented in Matlab, which is known to give much better results when using vectorized code. 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 [15], is not implemented. Initial LM algorithm with fixed weight decay [18] often performs many unsuccessful optimization search steps, which significantly slow down convergence. If decay rate is given as an exponential function of an error, number of unsuccessful search steps can be controlled by changing parameters B_1 and B_2, ranging from no oscillations to a non-negligible number of oscillations, but still smaller than in the case of standard LM algorithm. Good property of suggested weight decay is that the oscillations only start when the network is already close to minimum and starts overfitting, so they don't influence learning capability. In such scenario, third level of parallelization becomes unnecessary since it makes algorithm GPU implementation more complex, without improving learning capability.

V. Case studies

All tests are carried out using 3-layered feedforward NN. Number of input and output layer neurons depends on the benchmark problem, while number of hidden layer neurons is arbitrary. Learning process was carried out using 2000 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 is used [20]:

$$\theta_0 = 0.7H^{\frac{1}{M}}(-1 + 2 \cdot rand()), \tag{11}$$

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

Effectivness of algorithms - ALG1 being LM algorithm with exponential weight decay and ALG2 being LM algorithm with lin-log weight decay - is compared at two different levels. First, NNs were tested using CPU only and results of their accuracy, efficiency and relative speed of convergence are given in tables. Second, a comparison of CPU and GPU implementations of NNs with both algorithms is given 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:

$$S = \frac{T_{CPU}}{T_{GPU}}, \tag{12}$$

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

In order to compare learning algorithm's quality, an error measure needs to be defined. In this paper, all error measures are reported using non-dimensional Normalized Root Mean Square (NRMS) error index [1]. In all the tables, $NRMS_{val}$ denotes validation set error, $NRMS_{ti}$ denotes i-th test error, with i=1,2,3, while NoT denotes number of trials needed to reach the smallest validation error.

A. Nonlinear chaotic system prediction

In their paper on nonlinear signal processing, Lapedes and Farber [21] suggested Glass-Mackey chaotic system as a NN benchmark problem, due to its simple definition but hard to predict behavior. Glass-Mackey system is given in discrete time as

$$x(n) = \frac{1}{1+b}\left[x(n-1) + \frac{a \cdot x(n-\tau)}{1 + x^{10}(n-\tau)}\right]. \tag{13}$$

In this paper, $a = 0.2$ and $b = 0.1$. Sampling time T_0 is set to one second, and time delay τ to 30 seconds.

The goal of NN with LM algorithm is to predict 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 mapping function $f(\cdot)$ as follows:

$$x(n+P) = f(x(n), x(n-\Delta), \ldots, x(n-m\Delta)). \tag{14}$$

In this paper, $P = \Delta = 6$ and $m = 4$, which results in the following mapping function:

$$x(n+6) = f(x(n), x(n-6), \ldots, x(n-24)). \tag{15}$$

Fig. 2 shows Glass-Mackey time series of 1000 time steps for chosen τ (time is in units of τ). First 500 time

steps of time series were used in learning process, while 500 remaining time steps were used for validation of an algorithm.

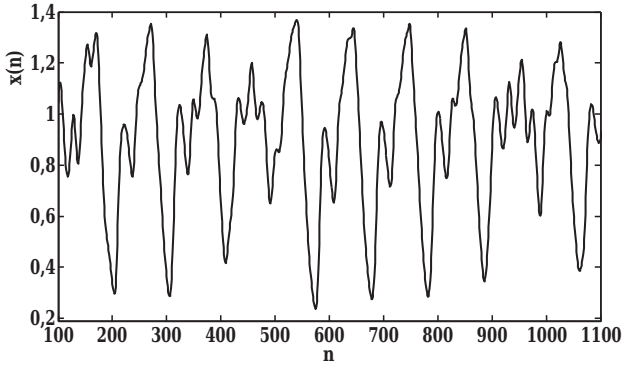

Fig. 2. Glass-Mackey time series

Task of a NN is to predict a single value of a system based on five known signal values, so the network has $6-13-1$ structure (bias neuron is added to both input and hidden layer). Number of hidden layer neurons, which is chosen to be 13, is arbitrary, i.e. is chosen based on author's experience.

Table I shows validation and test accuracies of prediction problem, when networks were trained fixed number of epochs on the CPU. Both algorithms were benchmarked 5 times, each time with different set of initial weights, generated by using (11). First row for each benchmark shows results obtained by network with ALG1 (NN ALG1), whereas second row shows results obtained by network with ALG2 (NN ALG2). It can be seen that, when it comes to accuracy, NN ALG1 performed similarly or better than NN ALG2 on all validation tests and additional 3 tests, which were performed on previously unseen data. When it comes to speed of convergence, NN ALG1 usually needs much more trials to reach minimum validation error than NN ALG2.

TABLE I
PREDICTION OF GLASS-MACKEY TIME SERIES - ACCURACY

	$NRMS_{val}$	NoT	$NRMS_{t1}$	$NRMS_{t2}$	$NRMS_{t3}$
Exper1	0.0790	2689	0.1145	0.1064	0.2964
	0.0820	624	0.1096	0.0951	0.2865
Exper2	0.0760	804	0.1091	0.1334	0.2896
	0.0771	704	0.1348	0.1652	0.2946
Exper3	0.0767	1975	0.1208	0.1553	0.2896
	0.0783	1083	0.2238	0.3134	0.3221
Exper4	0.0752	978	0.1142	0.1039	0.2877
	0.0802	606	0.1125	0.0987	0.2900
Exper5	0.0775	530	0.1173	0.1159	0.2882
	0.0793	426	0.1097	0.1109	0.2884

In order to directly compare both algorithms, the best accuracy, i.e. the lowest validation error achieved by a poorer performance algorithm, was used as a reference point of comparison. Table II shows number of trials and accuracy obtained by both algorithms, when trained to achieve reference validation error. It can be seen that, when both algorithms were trained to reach the same validation error, number of trials of ALG1 became similar or significantly smaller than the number of trials of ALG2, while keeping similar accuracy as in the case of fixed epochs training. Fig. 3 shows the best test result, obtained with NN ALG1.

TABLE II
PREDICTION OF GLASS-MACKEY TIME SERIES - ACCURACY (2)

	$NRMS_{val}$	NoT	$NRMS_{t1}$	$NRMS_{t2}$	$NRMS_{t3}$
Exper1	0.0820	770	0.1194	0.0988	0.2865
		624	0.1096	0.0951	0.2865
Exper2	0.0771	715	0.1111	0.1475	0.2886
		704	0.1348	0.1652	0.2946
Exper3	0.0783	753	0.1178	0.1504	0.2883
		1083	0.2238	0.3134	0.3221
Exper4	0.0802	319	0.1152	0.0883	0.2877
		606	0.1125	0.0987	0.2900
Exper5	0.0793	430	0.1123	0.1090	0.2881
		426	0.1097	0.1109	0.2884

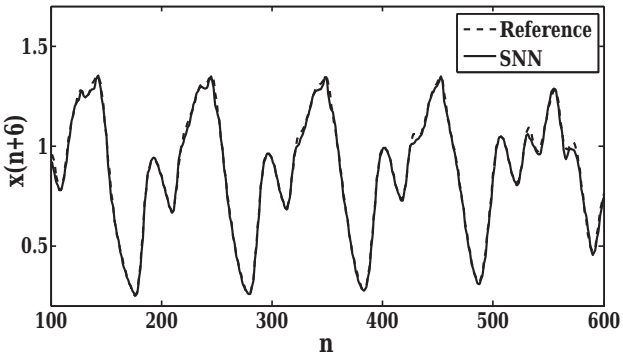

Fig. 3. Prediction of GM time series using NN ALG1

Table III shows efficiency of both algorithms, calculated as ratio of desired number of iterations and actually performed trials (efficiency of LM algorithm with fixed weight decay is around 50%). Although NN ALG2 shows higher efficiency than NN ALG1, other results show that this doesn't necessarily lead to better performance of the network, since most of the oscillations occur after the network had already reached validation minimum and started overfitting training data.

TABLE III
PREDICTION OF GLASS-MACKEY TIME SERIES - EFFICIENCY

	Exper1	Exper2	Exper3	Exper4	Exper5
ALG1 eff.(%)	68	72	69	68	74
ALG2 eff.(%)	75	77	77	76	77

Table IV shows speedups, as given in (12), achieved by networks with both algorithms when running them on the GPU, instead of on the CPU. It can be seen that networks with both parallel implementations of algorithms show similar speedups when compared to their sequential counterpart, although in all cases, NN ALG2 proves to be slighty more GPU-prone then NN ALG1.

TABLE IV

GLASS-MACKEY SYSTEM SPEEDUPS

	Exper1	Exper2	Exper3	Exper4	Exper5
ALG1	3.04	3.15	2.83	2.98	3.03
ALG2	3.17	3.38	3.31	3.13	3.46

B. 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 [22]. 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 Fig. 4. 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.

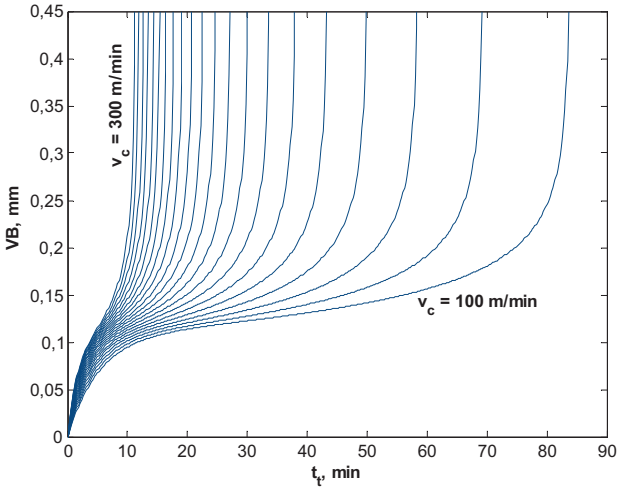

Fig. 4. Flank wear curves

Fig. 5. shows simulated (VB_{id}), estimated ($\tilde{VB}^E$) and filtrated ($\tilde{VB}$) curve. Filtrated curve represents desired output of a NN, so the goal of a NN is to generate an output similar to that curve, based on four previous values obtained from the estimator. Learning process was carried out using 2352 input-output pairs, whereas size of tests ranged from 278 to 835 input-output pairs.

Validation and test accuracies of filtration problem, obtained when network was trained fixed number of epochs on the CPU, are shown in Table V. Both algorithms were benchmarked 5 times. When it comes to accuracy, NN ALG1 outperformed NN ALG2 on all but one validation test and most of the additional 6 tests, performed on previously unseen data. When it comes to the speed of convergence, unlike in the case of a prediction problem, NN ALG1 reaches its minimum validation error much faster then NN ALG2. In Table V and Table VI, NN ALG1 is denoted by 1*, whereas

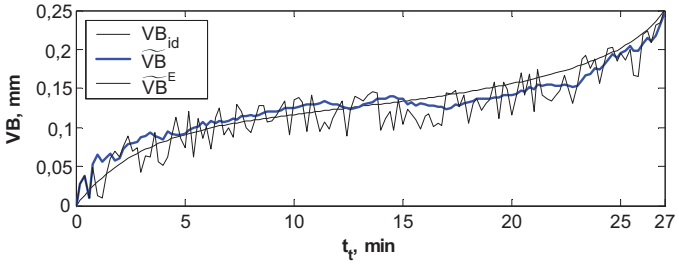

Fig. 5. Model and desired NN output

NN ALG2 is denoted by 2*.

TABLE V

FILTRATION OF ESTIMATED TOOL WEAR CURVES - ACCURACY

		Exper1	Exper2	Exper3	Exper4	Exper5
1*	$NRMS_{val}$	0.2694	0.2725	0.2657	0.2642	0.2647
	NoT	578	166	616	421	370
	$NRMS_{t1}$	0.2700	0.2714	0.2724	0.2649	0.2758
	$NRMS_{t2}$	0.2621	0.2602	0.2749	0.2675	0.2471
	$NRMS_{t3}$	0.2282	0.2306	0.2236	0.2220	0.2194
	$NRMS_{t4}$	0.2377	0.2402	0.2332	0.2306	0.2301
	$NRMS_{t5}$	0.2214	0.2295	0.2141	0.2158	0.2119
	$NRMS_{t6}$	0.2338	0.2521	0.2293	0.2267	0.2277
2*	$NRMS_{val}$	0.2686	0.2728	0.2709	0.2744	0.2664
	NoT	2098	250	1170	1287	1135
	$NRMS_{t1}$	0.2746	0.2726	0.2730	0.2740	0.2774
	$NRMS_{t2}$	0.2707	0.2623	0.2665	0.2733	0.2521
	$NRMS_{t3}$	0.2263	0.2307	0.2313	0.2297	0.2216
	$NRMS_{t4}$	0.2357	0.2407	0.2380	0.2381	0.2311
	$NRMS_{t5}$	0.2188	0.2292	0.2246	0.2284	0.2154
	$NRMS_{t6}$	0.2351	0.2497	0.2366	0.2429	0.2313

Table VI shows number of trials and accuracy obtained by both algorithms, when trained to achieve reference validation error. It can be seen that, when both algorithms were trained to reach the same validation error, number of trials of ALG1 was further decreased and thus ALG1 became even more faster compared to ALG2, while keeping similar accuracy as in the case of fixed epochs training. Fig. 6 shows the best test result, obtained with NN ALG1.

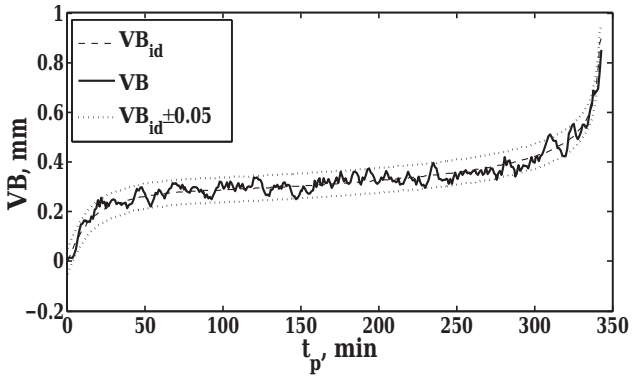

Fig. 6. Filtration of estimated tool wear curves using NN ALG1

Table VII shows efficiency of both algorithms (efficiency of LM algorithm with fixed weight decay is again around

TABLE VI

FILTRATION OF ESTIMATED TOOL WEAR CURVES - ACCURACY (2)

		Exper1	Exper2	Exper3	Exper4	Exper5
	$NRMS_{val}$	0.2694	0.2728	0.2709	0.2744	0.2664
1*	NoT	578	111	451	216	357
	$NRMS_{t1}$	0.2700	0.2729	0.2743	0.2708	0.2772
	$NRMS_{t2}$	0.2621	0.2639	0.2695	0.2643	0.2501
	$NRMS_{t3}$	0.2282	0.2330	0.2265	0.2303	0.2209
	$NRMS_{t4}$	0.2377	0.2426	0.2377	0.2384	0.2313
	$NRMS_{t5}$	0.2214	0.2324	0.2221	0.2291	0.2137
	$NRMS_{t6}$	0.2338	0.2570	0.2379	0.2434	0.2302
2*	NoT	1923	250	1170	1287	1135
	$NRMS_{t1}$	0.2712	0.2726	0.2730	0.2740	0.2774
	$NRMS_{t2}$	0.2658	0.2623	0.2665	0.2733	0.2521
	$NRMS_{t3}$	0.2263	0.2307	0.2313	0.2297	0.2216
	$NRMS_{t4}$	0.2358	0.2407	0.2380	0.2381	0.2311
	$NRMS_{t5}$	0.2196	0.2292	0.2246	0.2284	0.2154
	$NRMS_{t6}$	0.2363	0.2497	0.2366	0.2429	0.2313

50%). Similar to the case of a prediction problem, NN ALG2 shows higher efficiency than NN ALG1, but since this is not necessary condition for better performance, NN ALG2 doesn't outperform NN ALG1 on any criteria.

TABLE VII

FILTRATION OF ESTIMATED TOOL WEAR CURVES - EFFICIENCY

	Exper1	Exper2	Exper3	Exper4	Exper5
ALG1 eff.(%)	59	72	66	56	66
ALG2 eff.(%)	78	80	79	78	79

Table VIII shows speedups achieved by networks with both algorithms when running NNs on the GPU, instead of on the CPU. Again, both algorithms show similar speedups and NN ALG2 proves to be slighty more GPU-prone then NN ALG1. Speedups are here significanly bigger than in the case of a prediction problem, due to the size and complexity of a problem that was used for testing.

TABLE VIII

FILTRATION PROBLEM SPEEDUPS

	Exper1	Exper2	Exper3	Exper4	Exper5
ALG1	12.55	13.40	12.60	13.17	10.11
ALG2	12.97	13.72	12.96	12.79	10.53

VI. CONCLUSIONS

GPU-based, parallel implementation of the feedforward NN, with modified LM algorithm, is proposed. Algorithm is modified by introducing exponential weight decay and its effectiveness is compared to lin-log weight decay, suggested in the literature. Parallelization of the NN is achieved on two different levels - parallelization accross data set and parallelization of Jacobian computation. Third level of parallelization, suggested in the literature, is made redundant by introducing exponential weight decay.

Simulations are carried out on two real domain benchmark problems and results obtained show the effectiveness of proposed algorithm modification. Our modification works especially well, both on the CPU and the GPU, when learning problems become bigger, which is an important property since LM algorithm's main drawback are large-scale problems.

Future work will be oriented towards CUDA implementation of parallel NN with LM algorithm, as well as parallelization of Hessian matrix inversion, which proves to be main bottleneck of proposed implementation. Also, bigger in size benchmark problems will be used for further testing.

REFERENCES

[1] D. Brezak, T. Bacek, D. Majetic, J. Kasac and B. Novakovic, "A Comparison of Feedforward and Recurrent Neural Networks in Time Series Forecasting," *Conf. Proc. IEEEE-CIFEr*, pp. 119-124, 2012.

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

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

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

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

[6] T. Chen, D. Han, F. Au and L. Tham, "Acceleration of Levenberg-Marquardt training of neural networks with variable decay rate," *IEEE Proc. of IJCNN '03*, vol. 3, pp. 1873-1878, 2003.

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

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

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

[10] L.S.H. Ngia and J. Sjoberg, "Efficient Training of Neural Nets for Nonlinear Adaptive Filtering Using a Recursive Levenberg-Marquardt Algorithm," *IEEE Transactions on Signal Processing*, vol. 48, no. 7, pp. 1915-1927, July 2000.

[11] V.S. Asirvadam, S.F. McLoone and G.W. Irwin, "Parallel and Separable Recursive Levenberg-Marquardt Training Algorithm," *Proc. 12th IEEE Workshop on Neural Netw. for Signal Process.*, pp. 129-138, 2002.

[12] R. Daniel, "Parallel nonlinear optimization," *Proc. of Fifth Distributed Memory Computing Conf. (DMCC5)*, Charleston, SC, April 1990.

[13] U. Lotric and A. Dobnikar, "Parallel implementation of feedforward neural network using MPI and C# on .NET platform," *Adaptive and Neural Computing Algorithms*, pp. 534-537, Jul. 2005.

[14] N.N.R.R. Suri, D. Deodhare and P. Nagabhushan, "Parallel Levenberg-Marquardt-based Neural Network Training on Linux Clusters - A Case Study," *Proc. 3rd Indian Conf. on Computer Vision, Graphics and Image Processing*, India, 2002.

[15] J. Cao, K.A. Novstrup, A. Goyal, S.P. Midkiff and J.M. Caruthers, "A parallel Levenberg-Marquardt algorithm," *Proc. 23rd Inter. Conf. on Supercomputing* , pp. 450-459, USA, 2009.

[16] R. D. Prabhu, "GNeuron: Parallel Neural Networks with GPU," *Int. Conf. on HiPC*, December 2006.

[17] S. Kajan and J. Slacka, "Computing Of Neural Network On Graphics Card," *Int. Conf. Techical Computing*, Bratislava, 2010.

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

[19] B.M. Wilamowski, "An algorithm for fast convergence in training neural networks," *Proc. IJCNN*, vol. 3, pp. 1778-1782, 2001.

[20] D. Nguyen and B. Widrow, "Improving the Learning Speed of 2-Layer Neural Networks by Choosing Initial Values of the Adaptive Weights," *Proc IJCNN*, vol. 3, pp. 21-26, 1990.

[21] A. Lapedes and R. Farber, "Nonlinear signal processing using neural networks : prediction and system modelling," *Los Alamos National Laboratory*, New Mexico, 1987.

[22] D. Brezak, D. Majetic, T. Udiljak and J. Kasac, "Flank Wear Regulation using Artificial Neural Networks," *JMST*, vol. 24(5), pp. 1041-1052, 2010.