

Parallel Implementation of Back-Propagation Neural Network Software on SMP Computers*

Victor G. Tsaregorodtsev

Institute of Computational Modeling SB RAS, Krasnoyarsk, Russian Federation
tsar@neuropro.ru

Abstract. Experiments of neural network training procedure parallelization are conducted. Several styles of parallelization are described and compared, estimations of neural network size and training set size that allow speedup on two-processors SMP machine are obtained.

1 Introduction

Artificial neural networks are a flexible instrument for solving a lot of problems including non-linear regression, supervised learning and pattern recognition, unsupervised learning, associative memory, optimization tasks etc. Here we study only a back-propagation neural networks introduced in 1986 and named so because of the main part of its training algorithm doing "back propagation of errors" to compute gradient vector along adjustable (trainable) variables.

Experiments of running neural networks on parallel computers start at the end of 1980-ies [1,2,3], but mostly focus on the specific architectures – transputers, connection machines, massively parallel computers. This epoch of investigations ends in the middle of 1990-ies with a remarkable works of [4,5,6,7,8] (see also references therein). Moreover, results of [8] were confirmed recently [9,10,11]: so-called online training is theoretically faster than batch-training, but unparallelizable (batch training that accumulates penalties and updates over the patterns of the training set can be parallelized, but in general converges slowly).

Here we study parallelization for SMP (symmetric multiprocessors) computers. This research becomes necessary due to wide usage of multiprocessor servers, HyperThreading technology that was introduced by Intel Corp. recently, and plans of Intel Corp. to step to multicore processors, each core of which will support HyperThreading (i.e. can run two threads simultaneously under some restrictions). So in a nearest future we will be able to run up to 4 parallel threads on a single dual-core processor, and SMP computations will be of usual use.

Such a perspectives of hardware evolution makes SMP-programming more valuable than clustering techniques because of widely usage of a common computers (with multikernel processors therein) in a nearest future. Also, results obtained for SMP-software give some landmarks for a cluster platforms too.

In the paper we briefly describe neural network structure and training algorithm and possible parallelization schemes, then describe data bases used in

* This work was supported by the Krasnoyarsk regional scientific fund, grant 15G277.

experiments and provide experimental results. Then we discuss some additional questions and perspectives.

2 Artificial Neural Networks

2.1 Neural Network Structure

Here we use only a most convenient network structure – feedforward network with a single hidden layer of neurons. For a input vector \mathbf{x} of n components and vector of desired outputs \mathbf{y} of m components we can describe neural network as follows. Firstly, we compute the outputs of a hidden layer of N neurons as

$z_i = f\left(\sum_{k=1}^n x_k w_{ki} + w_{0i}\right)$ for each that neuron i using nonlinear function f , usually

of sigmoidal form a-la $f(\theta) = \frac{\theta}{c + |\theta|}$, $c > 0$. Then we compute each j -th output

signal $\hat{y}_j = \sum_{l=1}^N z_l u_{lj} + u_{0j}$. Variables w_{ab} , u_{cd} are trainable network coefficients

that should be adjusted during training. Training should minimize differences between desired and obtained signals (y_j and \hat{y}_j respectively) using some error measure, e.g. of a mean square error form.

Different optimization methods can be used during training – random search, genetic algorithms, gradient optimization techniques. Here we use the last one because "back propagation of errors" (precisely, back propagation of partial derivatives of the error measure) algorithm allows fast computation of gradient vector of error measure function along values of trainable network variables. When we obtain gradient vector, we can use gradient descent equation to improve quality of network's response by making step along the antigradient direction.

2.2 Training Scheme and Possibilities of Parallelization

Here we use batch training – accumulation of gradients for all the patterns collected in training set to compute overall gradient: for error measure $H_i(\mathbf{x}_i, \mathbf{y}_i) = \|\mathbf{y}_i - \hat{\mathbf{y}}_i(\mathbf{x}_i)\|$ for a training pattern $\{\mathbf{x}_i, \mathbf{y}_i\}$ and overall error $H = \sum H_i$ we can compute ∇H as $\nabla H = \nabla \sum H_i = \sum \nabla H_i$, therefore computation of different H_i 's and their gradients can be done in parallel over a different parts of the training set. I.e. for two parallel threads and K training patterns we can divide training set onto sets with pattern indexes $\{1, \dots, K/2\}$ and $\{K/2 + 1, \dots, K\}$, simultaneously compute H^1 and ∇H^1 by the first and H^2 and ∇H^2 by the second thread and then obtain final values of $H = H^1 + H^2$ and $\nabla H = \nabla H^1 + \nabla H^2$.

Oppositely, online-training corrects network after each pattern processing (along ∇H_i , not ∇H) – this is unparallelizable due to frequent modifications of the network and loss of suitability for any other parallelly computed gradient (which becomes obsolete after model correction along the concurrent one).

Each batch-training epoch consists of overall gradient computation, unnecessary phase of step size and/or descent direction selection (e.g. using conjugate

gradient method) and network modification. Iterations last until the desired value of H is obtained or some stopping criteria is met, e.g. local minima found.

For SMP-parallelization, i.e. fast memory access without any slow network links, we propose the following three data separation schemes:

1. Thread requests for a next unprocessed pattern, i.e. there is no hard or formal separation of the training set. But we should additionally synchronize access and modification of a pointer to a next pattern.
2. Hard separation – training set is divided onto a parts which number is equal to a number of processors without any redivisioning lately. Each processor (thread) here will work with a constant subset of patterns that can be cached.
3. Hard separation, but if the thread finishes his prescribed data processing, it handles some of currently unprocessed data assigned to the other thread.

The third scheme is required because we can stop i -th pattern training when the desired H_i is obtained. So we can skip ∇H_i computation, and when the numbers of already-trained patterns in a sets corresponded to different threads differs greatly, some threads may finish their work earlier and should wait others.

3 Data Bases Used in Experiments

We use 20 real-world data bases available from <http://kdd.ics.uci.edu/>. All of them are classification tasks. Table 1 summarize data base properties (number of classes, number of training examples and the number of bytes to store preprocessed data base and some additional information needed) and neural networks properties (number of neurons, input signals, number of adaptive variables in network). For three data bases we use neural networks of two different size in order to propagate results further along the network size scale.

4 Results of Experiments

Parallelization properties was implemented into author's own neural network software package running under MS Windows. We did not use any high-level parallelization package but create and synchronize threads using Window API. Neural network kernel previously was programmed careful enough – without object orientation that can hurt performance, with manual reprogramming of some routines using Assembler.

Experiments was conducted on a SMP workstation with two 1Ghz Pentium III processors. For the last two parallelization schemes (as numbered in Section 2.2) we also study implementations with hard assignment of every thread to a definite processor in order to maximize cache hitting for non-changing partitions of training set. Curiously, this versions help to increase speedup further (about 3÷5%) not for the small data bases but also for the great ones too, where only a little amount of data can be cached, but this results are not shown here.

Results of experiments are presented on Figures 1-3 where vertical axes count speedup obtained over the initial single-threaded version.

Table 1. Data base sizes and corresponded neural network sizes

Database name	Num. of patterns	Num. of input signals	Num. of classes	Num. of neurons	Num. of adaptive variables	Data size, in bytes
AnnThyroid	3772	21	3	10	253	437552
Car	1728	6	4	15	169	110592
HypoThyroid	3162	19	2	10	222	316200
Letter	20000	16	26	25	1101	5600000
Mushrooms	8124	111	2	10	1142	3802032
Musk	6598	166	2	10 / 15	1692 / 2537	4539424
Nursery	12960	8	5	20	285	1036800
OptDigits	3823	62	10	10	740	1284528
PageBlocks	5473	10	5	10	165	481624
PenDigits	7494	16	10	10	280	1139088
Satellite	4435	36	6	20	866	887000
Shuttle	43500	9	7	15	262	4350000
Spambase	4601	57	2	25	1502	1159452
Vowel	990	11	11	15	356	138600
Yeast	1484	8	10	40	770	178080
MF-Fac	2000	216	10	10 / 15	2280 / 3415	1904000
MF-Fou	2000	76	10	15	1315	784000
MF-Kar	2000	64	10	10	760	688000
MF-Pix	2000	240	10	10 / 15	2520 / 3775	2096000
MF-Zer	2000	47	10	15	880	552000

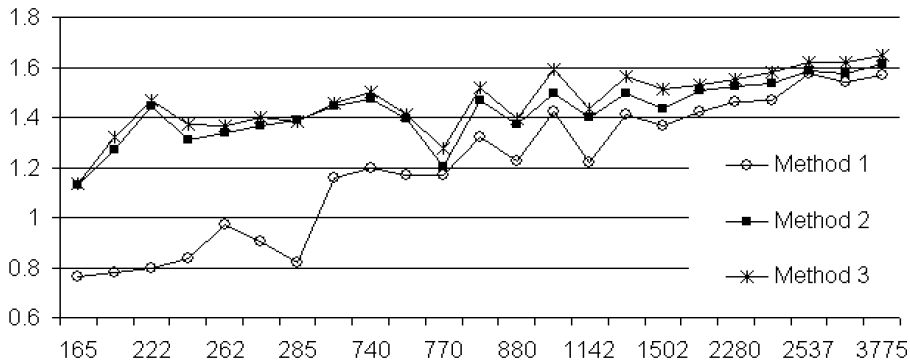


Fig. 1. Speedup obtained using two threads over single thread. Marks on a horizontal axis are from the simple ordering of the neural networks along their size

Fig.1 show speedup values when neural networks are simply ordered by their size. Effectivenesses correspond to parallelization schemes numbering, which is clear enough: for the scheme where threads simultaneously ask for the next unprocessed pattern great enough fraction of time is thrown away because of synchronization waits. When data set is divided between threads the third scheme with adaptive capturing of some examples unprocessed by the other thread runs faster than scheme with no-helping-to-each-other threads.

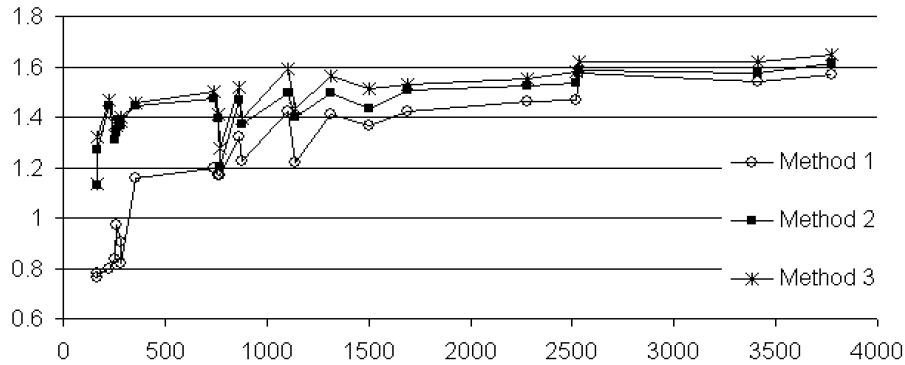


Fig. 2. The same results as in Fig.1, but horizontal axis counts real network size

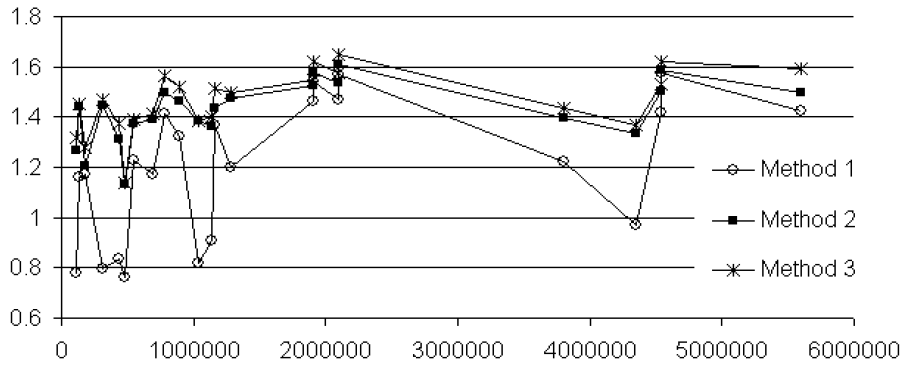


Fig. 3. Dependencies between data size, in bytes, and achieved speedup

Fig.2 shows the same results as Fig.1 but horizontal axis correspond to a real number of adaptive parameters in the network, so we can see more clearly that concurrent (first one) scheme gives no speedup for nets with less than 300 coefficients. Both Fig.1 and Fig.2 indicate speedup improvement with the network size growth, so it's possible to conclude that parallelization can be efficient (gives, at least, speedup of 1.5 for two-processors computer) only for the back-propagation networks with 1000 synapses at least. Network of that size is suitable for the great number of real world problems, and it's difficult to find a problem where a network with 10000 or more weights is required – but only for such a big networks we can achieve speedup near the theoretical limit of 2.

We should note that computations during neural network training are mainly of multiplication/accumulation instructions (see Section 2) that can be efficiently encoded by the compiler using SSE instructions (or programmer by itself can use vector-matrix computation package with SSE optimization therein) – this is the reason for the great influence of threads synchronization routines because each training epoch for small and medium-sized nets last only fraction of a second.

There may exist some dependence between speedup ratio and data base size (e.g., cache size influence), but less certain. Data base size affect on effectiveness less than network size, all the speedup drops on Fig.3 correspond to small nets.

5 Conclusion

We study the effectiveness of neural network software parallelization on two-processors SMP computer, explore three methods of data separation and some additional tricks. Obtained results are promising: for the most efficient scheme we obtain average speedup about 1.5 over a single-threaded program, and speedup varies from 1.2 to more than 1.6 over a wide pairs of network and data sizes.

We plan to step to heterogeneous parallel scemes which is necessary for HyperThreading feature of Intel Pentium IV Prescott processor where two threads can run simultaneously only while using different blocks of CPU, e.g. floating point and general purpose blocks. Also we'll study techniques and influential things more carefully, for other network structures and methods too [12].

References

1. Beyon T. A parallel implementation of the back-propagation algorithm on a network of transputers / Proc. First IEEE Int. Neural Network Conf. 1987.
2. Murali, P., Wechsler, H., Manohar, M. Fault-tolerance and learning performance of the back-propagation algorithm using massively parallel implementation / Proc. of Frontiers'90, The Third Symposium on the Frontiers of Massively Parallel Processing. College Park, MD, USA. 1990. – pp.364-367.
3. Paugam-Moisy H. On parallel algorithm for backpropagation by partitioning the training set / Proc. Fifth Int. Conf. Neural Networks and Their Applications. Nimes, France. 1992. – pp.53-65.
4. Kumar V., Shekhar S, Amin M.B. A scalable parallel formulation of backpropagation algorithm for hypercubes and related architectures / IEEE Trans. on Parallel and Distributed Systems, 1994. Vol.5. Issue 10. – pp.1073-1090.
5. Prechelt L. Data locality and load balancing for parallel neural network learning / Proc. Workshop on Compilers for Parallel Computers. Spain, 1995. – pp.111-127.
6. Misra M. Parallel environments for implementing neural networks / Neural Computing Surveys, 1997. Vol.1. – pp.48-60.
7. Strey A. EpsilonNN – A tool for the abstract specification and parallel simulation of neural networks / System analysis - Modeling - Simulation, special issue on Simulation of Artificial Neural Networks, 1999. Vol.34. No.4.
8. Torresen J., Tomita S., Landsverk O. The relation of weight update frequency to convergence of BP / Proc. World Conf. Neural Networks (WCNN'1995). Washington, USA. 1995. – pp.679-682.
9. Wilson D.R., Martinez T.R. The general inefficiency of batch training for gradient descent learning / Neural Networks. 2003, Vol.16. Issue 10. – pp.1429-1451.
10. Bottou L., LeCun Y. Large scale online learning / Advances in Neural Information Processing Systems 16 (2003). MIT Press, 2004. – pp.217-224.

11. Tsaregorodtsev V.G. General inefficiency of batch gradient usage for neural network training / Proc. XII Conf. "Neuroinformatics and their applications". Krasnoyarsk, Russia. 2004. – pp.145-151. (in Russian).
12. Tsaregorodtsev V.G. Perspectives for parallelization of neural-network data processing and analysis software/ Proc. III Conf. "Mathematics, Informatics, Control". Irkutsk, Russia. 2004. – 6p. (in Russian).