Parallel Training of Artificial Neural Networks Using Multithreaded and Multicore CPUs

Olena Schuessler and Diego Loyola

German Aerospace Center, Institute of Remote Sensing, Münchner Straße 20
82234 Weßling, Germany
{Olena.Schuessler,Diego.Loyola}@dlr.de

Abstract. This paper reports on methods for the parallelization of artificial neural networks algorithms using multithreaded and multicore CPUs in order to speed up the training process. The developed algorithms were implemented in two common parallel programming paradigms and their performances are assessed using four datasets with diverse amounts of patterns and with different neural network architectures. All results show a significant increase in computation speed, which is reduced nearly linear with the number of cores for problems with very large training datasets.

Keywords: Neural network training, multithreading and multicore, Pthreads and OpenMP parallelization.

1 Introduction

In recent years we observe a growing interest in artificial neural networks for solving all kinds of classification, function approximation, interpolation and forecasting problems. Neural networks as universal approximators [1] are a very powerful tool which can reproduce extremely complicated non-linear dependencies. Neural networks learn an underlying function from input/output examples and normally the more complicated the problem is the more examples, also called patterns, are needed.

Training a neural network for complicated and multi-dimensional problems normally means using very large amounts of training examples with hundred thousands or even millions of patterns. Such training can take weeks and even months to reach the desired accuracy. In the same way, finding an optimal neural network configuration requires a certain amount of cross-validation experiments, which can be also very time consuming. Therefore there is a need to speedup the training process of neural networks, especially for very large training datasets.

In our days multithreaded and multicore CPUs with shared memory are a cost-effective way of obtaining significant increases in CPU performance. An exponential growth in performance is expected in the near future from more hardware threads and cores per CPU [2]. Researches focused their attention recently on parallelizing a variety of computational intelligence algorithms [3-6] using these new CPUs.

For neural networks two basic approaches of parallelization can be defined: parallelizing the neural network structure and parallelizing the training process. The first approach uses the parallel nature of neural networks, and assigns to each processing node (neuron) a separate thread [3]. All neurons in one layer are processed simultaneously and synchronized before propagating into next layer. The second approach is to assign a part of the training dataset to each thread and process (train) them simultaneously. This approach is covered in [4], where a three-layer perceptrone neural network is parallelized and tested using two and eight threads. The same technique is implemented for dual-core processors in [5].

Our study is more general than the aforementioned papers as it covers a larger diversity of training algorithms, neural network architectures and parallel training implementations. In particular we focus on speeding up the training process for very large datasets with neural networks of complicated structures containing more than three layers, what will allow us to solve complex nonlinear real world problems.

This paper is organized as follows: section 2 gives a short summary of multilayer perceptrone neural networks and the backpropagation learning algorithm; section 3 describes the parallel implementation of the neural network training; section 4 shows the results with different test problems and section 5 presents the conclusions.

2 Multilayer Perceptrone Neural Network

One of the most popular neural network architectures is the multilayer perceptrone network (MLP) [7]. The MLP network is composed of an input layer, one or more hidden layers and an output layer. Each layer contains a certain amount of neurons and all neurons of neighbor layers are interconnected, see Figure 1.

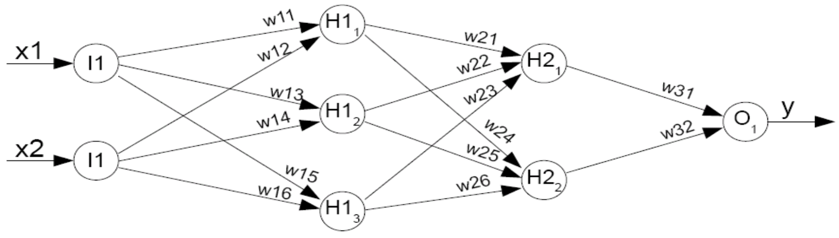

Fig. 1. Architecture of a 4-layer Multilayer Perceptrone Neural Network with 2 input neurons, 2 hidden layers with 3 and 2 neurons, and 1 output neuron

Each connection in the network has an associated weight. The task of each neuron is to compute the weighted sum of its inputs and to transform it to the output signal. This transformation is done with an activation function; the most popular activation functions for MLP are the *sigmoid* and *Gaussian* functions:

$$y = \frac{1}{1 + \exp(-x)} \qquad\qquad y = \exp(-x^2) \tag{1}$$

In a feedforward MLP computations propagate from the input layer through the hidden layers to the output layer that calculates the output signal.

2.1 Backpropagation Learning

Once we have computed the output of the neural network, we can compare it to a desired output value. The difference between these values is the network error. In order to reduce this error and to get a good match between network output and expected values we can iteratively adjust the weights in the network, until we reach a good agreement. One of the most common algorithms for adjusting the weights is the backpropagation algorithm [8] which propagates the error of output layer back to the input layer and changes the network weights.

There are two approaches for backpropagation training: in incremental (on-line) learning we update the weights after each pattern is presented to the network and in batch learning we accumulate error values over all patterns and then update the weights. The classical algorithm of batch training is described in Figure 2.

```
Neural network training algorithm
1.  Initialize weights in the network; set desired error value $e_{max}$ and maximum number of
    iterations $itr_{max}$; initialize the delta weights and number of iterations with zero
2.  For each pattern $p$ in training set $T$ do:
3.      Calculate the error of output neuron $e_{out}^{p} = o^{p} - t^{p}$, where $o^{p}$ is output of the
        network and $t^{p}$ is expected (target) value for pattern $p$
4.      Backpropagate: calculate the errors of neurons in hidden layers
        $e_{hid}^{p} = e_{out}^{p} \cdot w_{hid} \cdot dy(p)$, where $dy$ is derivative of activation function $y$
5.      Calculate delta weights $\Delta w_{ij}{}^{p} = \delta_j \cdot o_i$, where $\delta_j = dy_j(p) \cdot e_{out}^{p}$ for neurons of
        output layer, and $\delta_j = dy_j(p) \cdot \sum_{k \in outp(j)} \delta_k \cdot w_{jk}$ for neurons of hidden layer.
6.      Accumulate the new calculated delta weights $\Delta w_{ij} = \Delta w_{ij} + \Delta w_{ij}{}^{p}$
7.  End For
8.  Update weights in the networks $w_{ij} = w_{ij} + \eta \cdot \Delta w_{ij}$, where $\eta$ is the learning rate
9.  Compute mean square network error (MSE) on training set $T$ as $e_{out} = \frac{1}{|T|} \sum_{p \in T} (e_{out}^{p})^2$
    where $|T|$ is the cardinality of $T$, i.e. the number of patterns in training dataset
10. If $e_{out} > e_{max}$ and $itr < itr_{max}$ then increment the number of iterations and return to 2.
```

Fig. 2. Algorithm for neural network backpropagation in batch learning mode

3 Parallelized Neural Network Training

The most common way to create parallel programs for shared memory systems is multithreading. Functional parallelism works through specially programmed thread functions and data parallelism works through shared virtual address space of a process which can be accessed from every thread.

3.1 Parallel Backpropagation Learning

To parallelize the backpropagation algorithm we decided to use the batch training approach as it is relatively easy to adapt for multithreaded and multicore CPUs. First we divide the training dataset T into equal parts $T_1, T_2, \ldots, T_N$, where N is the number of threads. Then steps 2 to 7 of the algorithm presented in Figure 2 can run in parallel threads that independently performs backpropagation of each pattern in the dataset T_k assigned to it. When every thread have finished steps 2 to 7 for its training dataset T_k, the corresponding delta weights $\Delta w_{ij}^{T_k}$ are then accumulated together

$$\Delta w_{ij} = \sum_{k=1}^{N} \Delta w_{ij}^{T_k} \tag{2}$$

After that, steps 8 and 9 of the algorithm are executed sequentially: update weights of the network and compute the new network error. If the expected network accuracy is not reached then a next iteration is started by repeating steps 2 to 7 in parallel. In order to avoid overfitting during the training we use the early stopping method [9]: the training is stopped when the network error in an independent test dataset (with patterns not included in the training dataset) increases.

In Figure 3 we see a comparison between training and testing steps using sequential batch learning and the corresponding steps using the proposed parallel batch learning. It is important to note that in our parallelized training algorithm we do not move/copy the training datasets between threads, which would take too long time in case of large datasets, but we only update/copy the relative few neural network weights.

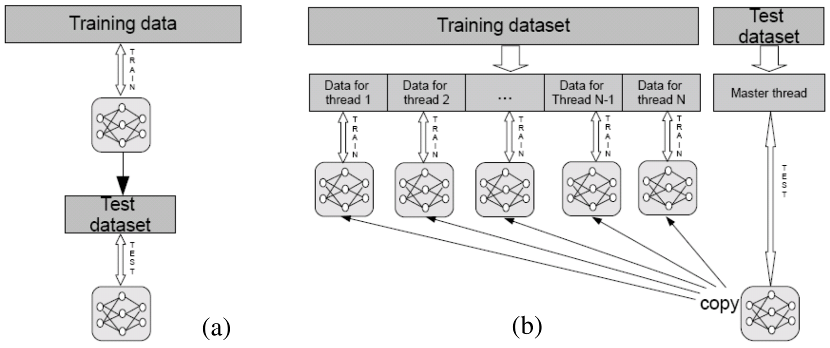

Fig. 3. Schematic representation of the neural network training and testing procedure for (a) sequential and (b) parallelized backpropagation implementation

Similar to the parallelization of batch training algorithm, we created parallel versions of other two popular training algorithms which for some tasks proved to be faster and converge better: RProp [10] and QuickProp [11].

3.2 Multithreaded Training Implementation

The most common approaches of multithreading programming are POSIX Threads (Pthreads) [12], which is library based and requires parallel coding, and OpenMP [13], which is based on compiler directives and can also use serial code. The OpenMP implementation of algorithms is usually more straightforward and doesn't require specifically parallel coding, whereas the Pthreads implementation gives more control over the parallelization but requires specifically parallel programming [14]. The performance of Pthreads-based and OpenMP-based algorithm parallelization is problem and platform specific [15].

In this work we implement different neural network training algorithms using both parallelization techniques and compare their performances. Figure 4 shows in detail the parallelization scheme developed for the neural network training. Here we see the division of work between the master thread and its slave threads.

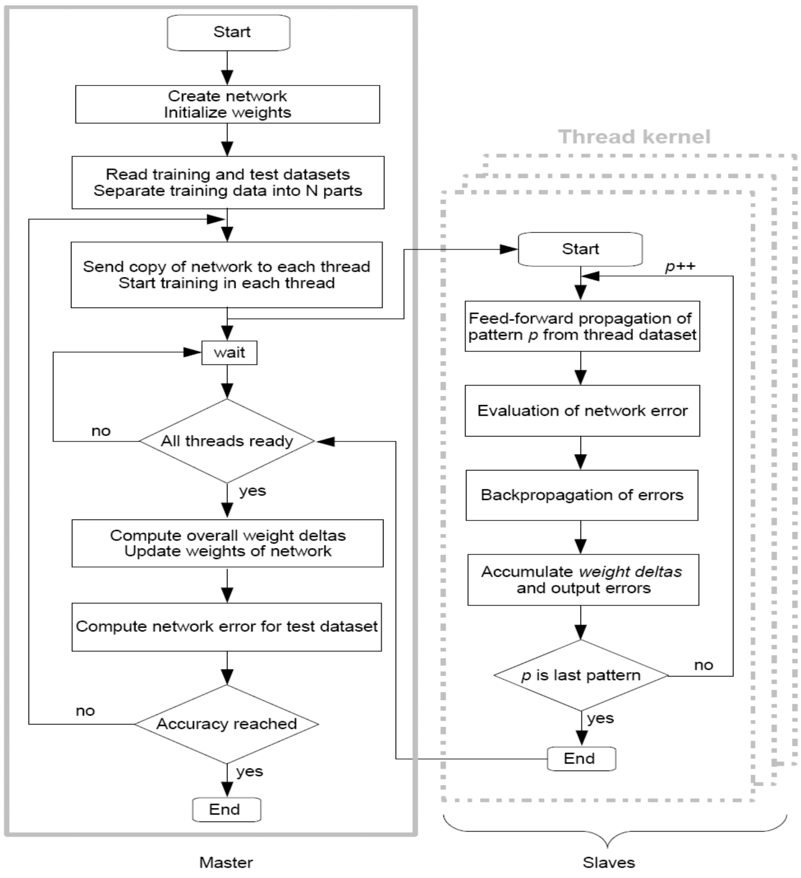
Start
Create network
Initialize weights
Read training and test datasets
Separate training data into N parts
Send copy of network to each thread
Start training in each thread
wait
no
All threads ready
yes
Compute overall weight deltas
Update weights of network
Compute network error for test dataset
no
Accuracy reached
yes
End
Master
Thread kernel
Start
p++
Feed-forward propagation of pattern p from thread dataset
Evaluation of network error
Backpropagation of errors
Accumulate weight deltas and output errors
p is last pattern
no
yes
End
Slaves

Fig. 4. Parallel implementation of the backpropagation learning algorithm. Here the master thread divides training data into equal parts. Each slave trains the neural network using the subset of the training data assigned to it. When all threads are ready, the master thread computes the overall weights update of the neural network.

In the master thread we first create a new neural network (or load it from a file if we want to continue training of existing neural network). Then the master thread reads the training dataset T and separates them into equally large dataset T_1, T_2,..T_N , initializes N slaves and sends to each of them a separate part of the training dataset T_k together with a copy of original neural network. Each slave computes the weight delta updates based on the current copy of the neural network and its training dataset T_k. As soon as all slaves have finished their computations, they send accumulated *weight deltas* to the master thread and the master thread combines them to update weights of the reference neural network and computes the network output error (in our implementation mean square error (MSE)) for the test dataset. This process is repeated until the MSE value of the test dataset is smaller than a given threshold or until a maximum number of training epochs is reached.

4 Simulations and Results

4.1 Datasets Used for Simulations

We test the sequential and parallelized training algorithms on four different problems. Information on the corresponding datasets, neural network configurations and algorithms used for the training are shown in the Table 1.

Table 1. Simulation datasets and corresponding network configurations

Dataset				**Neural network**		
Name	*Number of patterns*	*Number of inputs*	*Number of outputs*	*Training algorithms*	*Number of layers*	*Number of neurons*
Characters recognition	3823	64	10	Backprop & RProp	5	177
Surface interpolation	7840	2	1	Backprop & QuickProp	4	53
Ozone extrapolation	120072	2	1	Backprop & RProp	4	33
O_2 A-band simulation	1998855	8	62	Backprop & QuickProp	5	126

The first dataset is a typical classification problem of handwritten digits [16] and has a few thousand patterns; four of them are shown in Figure 5(a). The second dataset corresponds to a surface interpolation problem shown in Figure 5(b) with few thousand patters computed using the function $z = \sin((x^2 + y^2)^{1/2}) / (x^2 + y^2)^{1/2}$.

The third dataset describes a spatial extrapolation problem for the global concentration of ozone on the atmosphere as measured by satellites [17]. The inputs are latitude and longitude and expected total ozone is the output, see Figure 5(c). This dataset contains more than one hundred thousand patterns. The fourth dataset is from a function approximation problem with almost two million training patterns of oxygen

A-band reflectivity simulations for various geophysical conditions [18]. An example of such training patterns is shown in the Figure 5(d).

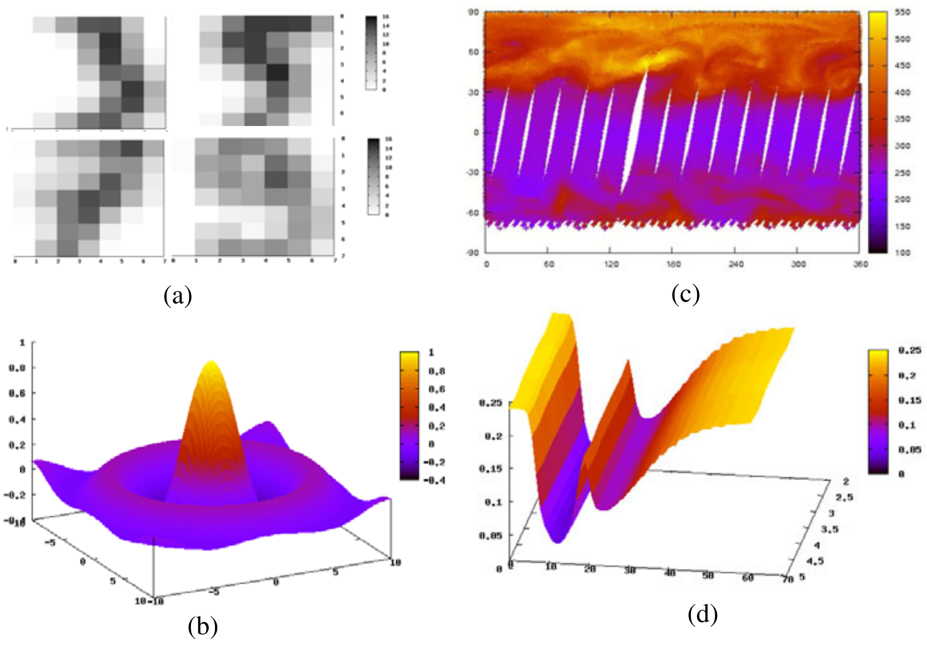

Fig. 5. (a) Example patterns for handwritten characters 3, 5, 7 and 9 from the Character recognition dataset; (b) plot of the Surface interpolation dataset; (c) satellite data of global ozone concentration from the Ozone extrapolation dataset; (d) example of pattern from O_2 A-band dataset (reflectivity as function of wavelength and cloud-top height)

4.2 Efficiency of Parallelization

For the simulations we used different networks configurations as described in Table 1. A fixed learning rate of 0.7 is used in all simulations.

To compare the speedup S of the parallelized training over sequential training, we use the following equation

$$S = \frac{\tau_{single}}{\tau_{parall.}} \cdot 100\% \tag{3}$$

where τ_{single} and τ_{parall} are the time of computation for the sequential and parallelized versions, both measured over 1000 epochs of training.

We also compute ideal speedup S_{ideal} for each problem according to Amdahl [19]

$$S_{ideal} = \frac{1}{\left(\frac{f}{n} + 1 - f\right)} \cdot 100\% \tag{4}$$

where f is the fraction of the code which is parallelizable and n is the number of cores.

The efficiency E of the parallelized training is calculated using

$$E = \frac{\tau_{single}}{n \cdot \tau_{parall.}} \tag{5}$$

The single and parallel trainings were started using the same initial values of network weights. Each experiment was repeated 10 times using different network configurations and then the average time needed for the training was computed. The computers used for our simulation have 2 Quad-Core L5420 Intel CPUs (2.5 GHz) with 8 GByte RAM each

Figure 6 shows the computed ideal speedup for each problem and the measured speedup of parallelized network training for the four simulation datasets using both parallelization techniques: POSIX threads on the left and OpenMP on the right.

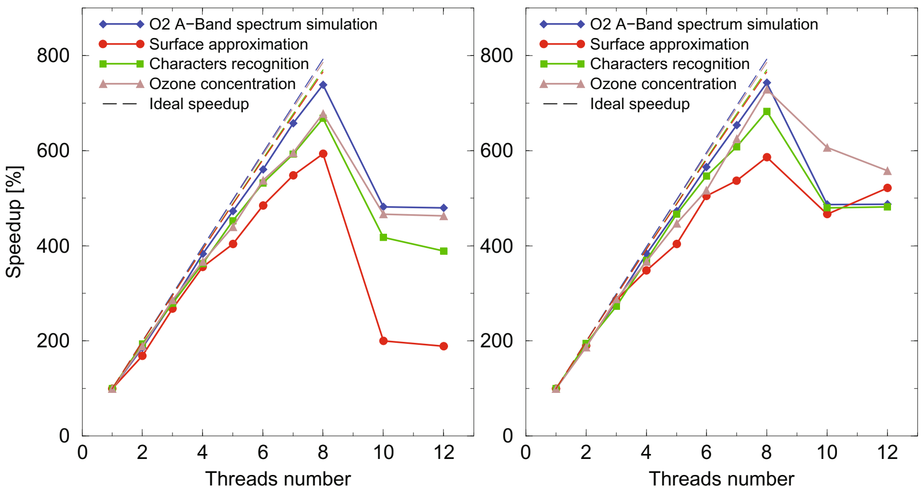

Fig. 6. Measured speedup (solid lines) of neural network training parallelization as function of the used threads and expected ideal speedup (dashed lines). Slightly better results for the four simulation datasets are obtained on an 8-core computer with the parallel training implemented in OpenMP (right) compared to the one implemented in Pthreads (left).

We can see that the best speedup is achieved when we use as many server threads as available cores. In case of using more threads than cores, the time for context switches and scheduling lower considerably the efficiency of parallelization.

Figure 7 shows the efficiency of the parallelization for the four problems for different number of threads/cores. Increasing the number of used threads causes some synchronization overhead but a tremendous speedup is obtained in any case.

Best efficiency and speedup are achieved for the O2 A-Band spectrum simulation problem (740% with Pthreads and 743% with OpenMP) for 8 threads. This can be explained with the large size of the training dataset (1998855 patterns) and relative small network structure (126 neurons). Each core is highly loaded with training of the

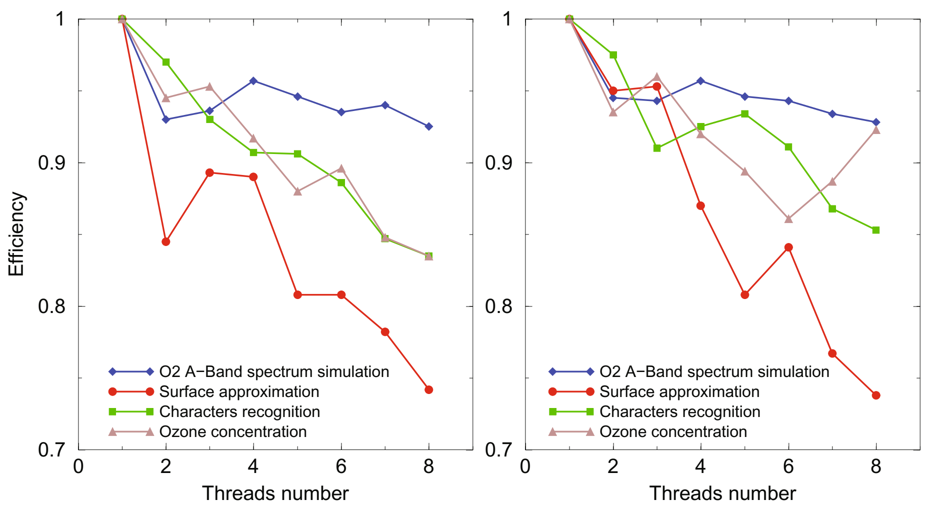

Fig. 7. Efficiency of the neural network training parallelization on an 8-core computer as function of the used threads. The efficiency is highest and stays nearly constant for the problem with the larger number of patterns (O2 A-Band spectrum) in both OpenMP (right) and Pthreads (left) implementations.

assigned patterns and relative short synchronization time is needed for updating the relative few weights of the master network and copying them back to the slave threads. For smaller datasets the efficiency is lower because the time for synchronization is relatively high compared with the time for training. We also observe that the training algorithm parallelization with OpenMP performs slightly better than the one implemented with POSIX threads.

5 Conclusions

In this paper we proposed a method for parallelization of neural network training based on the backpropagation algorithm and implemented it using two different multithreading techniques (OpenMP and POSIX threads) applicable to the current and next generation of multithreaded and multicore CPUs. The main motivation to parallelize the training process of neural network was to speedup the training process of very large training datasets.

The speedup and efficiency of the proposed parallel algorithm and multithread implementations were analyzed using four different datasets. A nearly ideal speedup was reached for very large training datasets. For example in case of the O2 A-Band spectrum simulation problem with almost two million patters we obtained an increase in computation speed very close to the expected ideal, that is 743% when 8 cores were used.

References

1. Pinkus, A.: Approximation theory of the MLP model in neural networks. Acta Numerica 8, 143–195 (1999)
2. Sodan, A.C., Machina, J., Deshmeh, A., Macnaughton, K., Esbaugh, B.: Parallelism via Multithreaded and Multicore CPUs. Computer 43(3), 24–32 (2010)
3. Seiffert, U.: Artificial Neural Networks on Massively Parallel Computer Hardware. In: ESANN 2002 Proceedings - European Symposium on Artificial Neural Networks, April 24-26, pp. 319–330. Bruges, Belgium (2002)
4. Turchenko, V., Grandinetti, L.: Efficiency Analysis of Parallel Batch Pattern NN Training Algorithm on General-Purpose Supercomputer. In: Omatu, S., Rocha, M.P., Bravo, J., Fernández, F., Corchado, E., Bustillo, A., Corchado, J.M. (eds.) IWANN 2009. LNCS, vol. 5518, pp. 223–226. Springer, Heidelberg (2009)
5. Tsaregorodtsev, V.: Parallel Implementation of back-Propagation Neural Network Software on SMP Computers. In: Malyshkin, V.E. (ed.) PaCT 2005. LNCS, vol. 3606, pp. 186–192. Springer, Heidelberg (2005)
6. Lotrič, U., Dobnikar, A.: Parallel Implementations of Recurrent Neural Network Learning. In: Kolehmainen, M., Toivanen, P., Beliczynski, B. (eds.) ICANNGA 2009. LNCS, vol. 5495, pp. 99–108. Springer, Heidelberg (2009)
7. Gallant, S.: Perceptron-based learning algorithms. IEEE Transactions on Neural Networks 1(2), 179–191 (1990)
8. Rummelhart, D., Hinton, G., Williams, R.: Learning Internal Representations by Error Propagation. In: Rumelhart, D.E., McClelland, J.L. (eds.) Parallel Distributed Processing, vol. I, pp. 318–362. MIT Press, Cambridge (1986)
9. Prechelt, L.: Automatic early stopping using cross validation: quantifying the criteria. Neural Networks 11(4), 761–767 (1998)
10. Riedmiller, M., Braun, H.: Rprop - A Fast Adaptive Learning Algorithm. In: Proceedings of the International Symposium on Computer and Information Science VII, Technical Report (1992)
11. Fahlman, S.: An Empirical Study of Learning Speed in back-Propagation Networks. Computer Science Technical Report, CMU-CS-88-162 (1988)
12. Butenhof, D.R.: Programming with POSIX Threads. Addison-Wesley, Reading (1997) ISBN 0-201-63392-2
13. Quinn, M.J.: Parallel Programming in C with MPI and OpenMP. McGraw-Hill Inc., New York (2004) ISBN 0-07-058201-7
14. Kuhn, B., Petersen, P., O'Toole, E.: OpenMP versus threading in C/C++. Concurrency: Practice and Experience 12, 1165–1176 (2000)
15. Stamatakis, A., Ott, M.: Exploiting Fine-Grained Parallelism in the Phylogenetic Likelihood Function with MPI, Pthreads, and OpenMP: A Performance Study. In: Chetty, M., Ngom, A., Ahmad, S. (eds.) PRIB 2008. LNCS (LNBI), vol. 5265, pp. 424–435. Springer, Heidelberg (2008)
16. Alpaydin, E., Kaynak, C.: Optical Recognition of Handwritten Digits Data Set, `http://archive.ics.uci.edu/ml/datasets/`
17. Loyola, D., Coldewey-Egbers, M., Dameris, M., Garny, H., Stenke, A., Van Roozendael, M., Lerot, C., Balis, D., Koukouli, M.: Global long-term monitoring of the ozone layer - a prerequisite for predictions. International Journal of Remote Sensing 30(15), 4295–4318 (2009)
18. Loyola, D.: Applications of Neural Network Methods to the Processing of Earth Observation Satellite Data. Neural Networks 19(2), 168–177 (2006)
19. Tang, G., D'Azevedo, E., Zhang, F., Parker, J., Watson, B., Jardine, P.: Application of a hybrid MPI/OpenMP approach for parallel groundwater model calibration using multi-core computers. Computers & Geosciences 36(11), 1451–1460 (2010)