To appear: World Transputer Congress '95, 4 - 6 September 1995, Harrogate, UK

Parallel Back-Propagation for Sales Prediction on Transputer Systems

Frank M. Thiesing, Ulrich Middelberg, Oliver Vornberger
University of Osnabrück, Dept. of Math./Comp. Science
D-49069 Osnabrück, Germany
frank@informatik.uni-osnabrueck.de

Abstract. In this paper artificial neural networks are adapted to a short term forecast for the sale of articles in supermarkets. The data is modelled to fit into feedforward multilayer perceptron networks that are trained by the back-propagation algorithm. For enhancement this has been parallelized in different manners. One batch and two on-line training variants are implemented on parallel Transputer-based PARSYTEC systems: a GCel with T805 and a GC/PP with PowerPC processors and Transputer communication links. The parallelizations run with both the runtime environments PARIX and PVM.

1 Introduction

Time series prediction for economic processes is a topic of increasing interest. In order to reduce stock-keeping costs, a proper forecast of the demand in the future is necessary. In this paper we use artificial neural networks for a short term forecast for the sale of articles in supermarkets. The nets are trained on the known sales volume of the past for a certain group of related products. In addition information like changing prices and advertising campaigns is also given to the net to improve the prediction quality. The net is trained on a window of inputs describing a fixed set of recent past states by the *back-propagation* algorithm [1].
For enhancement the algorithm has been parallelized in different manners: First the training set can be partitioned for the batch learning implementation. The neural network is duplicated on every processor of the parallel machine, and each processor works with a subset of the training set. After each epoch of training the weight changes are broadcasted and merged.
The second way is the parallel calculation of the matrix products that are used in the learning algorithm. The neurons in each layer are partitioned into p disjoint sets and each set is mapped on one of the p processors. The new activations are distributed after each training pair. We have implemented this on-line training in two variants: For the first parallelization one matrix product is not determined on one processor, but it is calculated while the partial sums are sent around on a processor cycle. The second method tries to reduce communication time. Therefor it needs an overhead in both storage and number of computational operations.

The parallel implementations take place on parallel Transputer-based PARSYTEC systems: a GCel with T805 and a GC/PP with PowerPC processors and Transputer communication links. The parallelizations run with both the runtime envirionments PARIX and PVM.

2 Sales forecast by neural networks

In our project we use the sale information of 53 articles of the same group in a supermarket. The information about the number of sold articles and the sales revenues in DM (Deutsche Mark, German currency unit) are given weekly starting September 1994. In addition there are advertising campaigns for articles often combined with temporary price reductions. Such a campaign lasts about two weeks and has a significant influence on the demand on this article. Sale, advertising and price for two different articles are shown in figures 1 and 2.

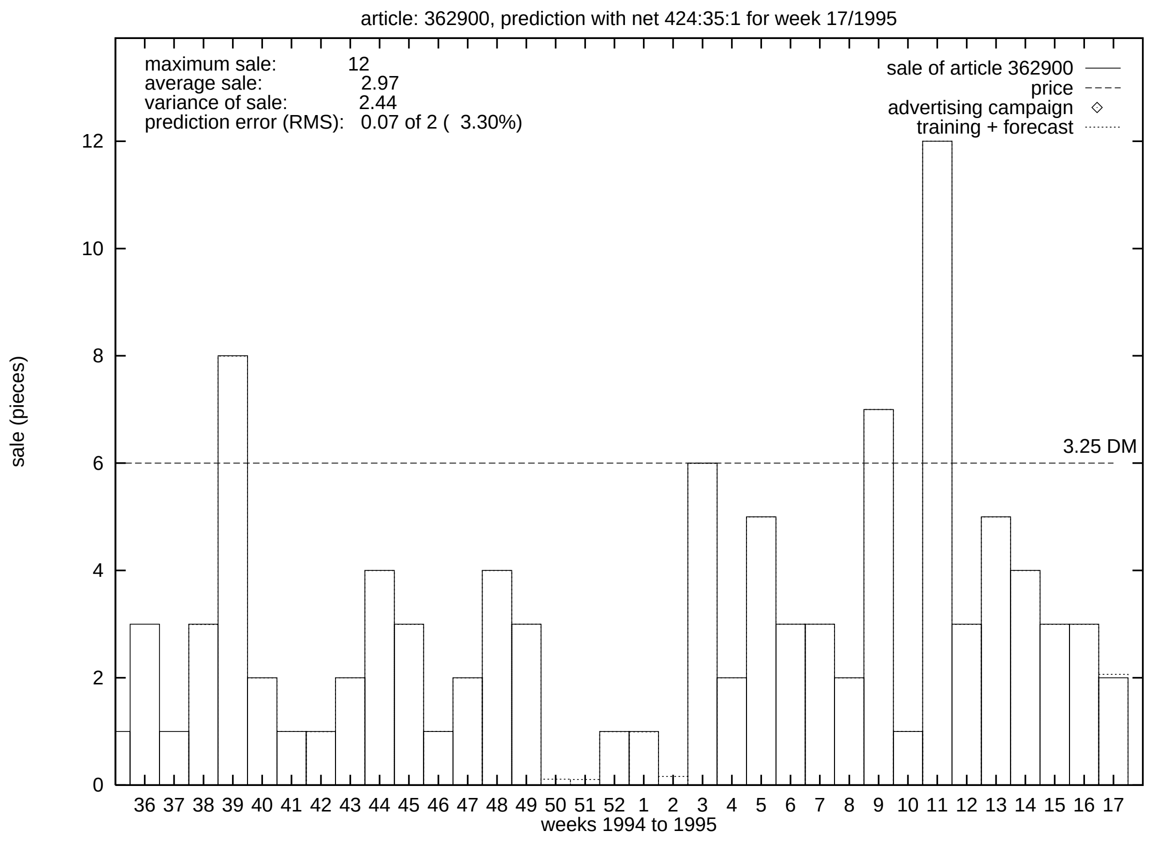


Figure 1: article 362900 (without advertising)

The aim is to forecast the sale of an article for the next week by neural networks. We use a feedforward multilayer perceptron network (see figure 4) with one hidden layer together with the back-propagation training method [3], [4], [5].

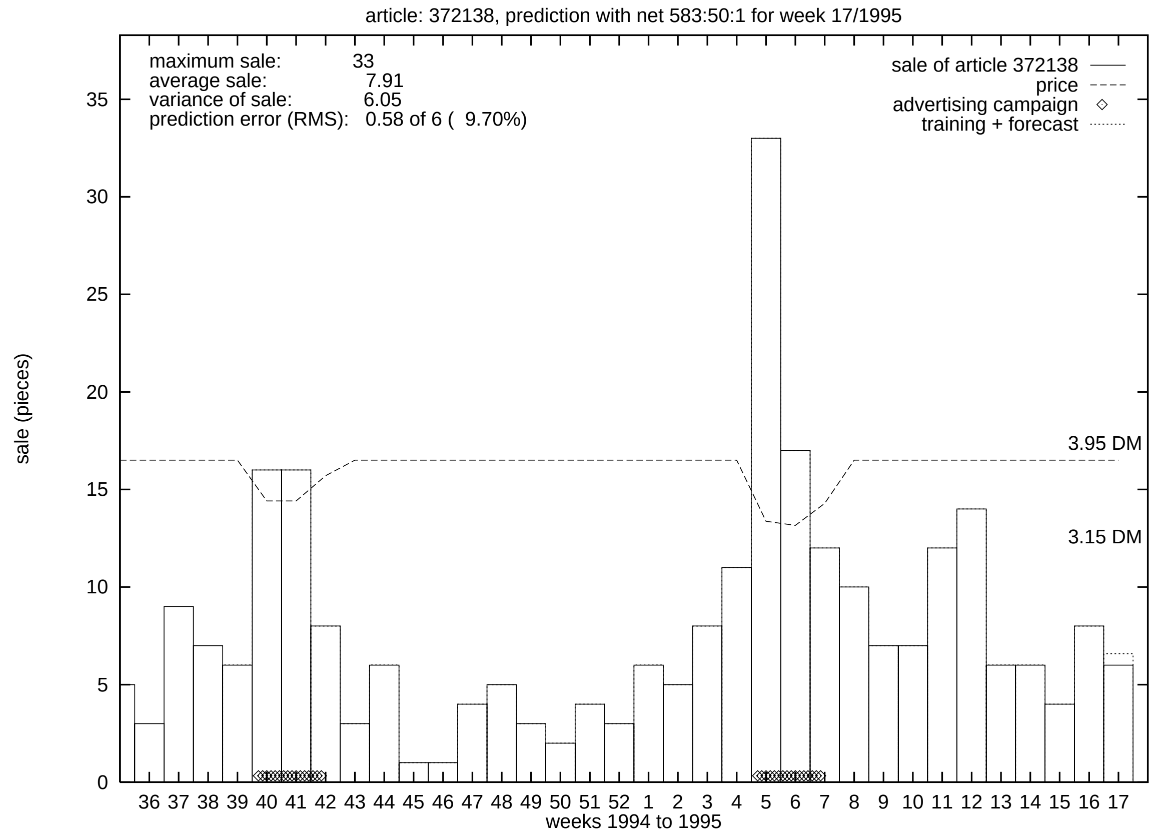
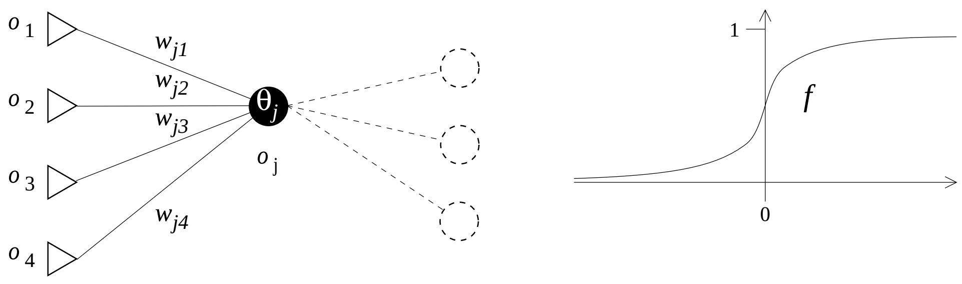


Figure 2: article 372138 (with advertising)

For prediction the past information of n recent weeks is given to the input layer. The only result in the output layer is the sale for the next week. So there is a window of n weeks in the past and one week in the future. Both the input and output together are called a training pair. One training of all training pairs is called an *epoch*.

2.1 Artificial neural networks

Artificial neural networks consist of simple calculation elements, called neurons, and weighted connections between them. In a *feedforward multilayer perceptron* (figure 4) the neurons are arranged in layers and a neuron from one layer is fully connected only to each neuron of the next layer. The first and last layer are the *input* respectively *output* layer. The layers between them are called *hidden*. Values are given to the neurons in the input layer; the results are taken from the output layer. The outputs of the input neurons are propagated through the hidden layers of the net. Figure 3 shows the algorithm each neuron performs.
The activation a_{hj} of a hidden or output neuron j is the sum of the incoming data multiplied by the connection weights like in a matrix product. The individual *bias* value θ_{hj} is added to this before the output o_{hj} is calculated by a sigmoid function f:



$$o_j = f(\, w_{j1}\, o_1 + w_{j2}\, o_2 + w_{j3}\, o_3 + w_{j4}\, o_4 + \theta_j \,)$$

Figure 3: How a perceptron works

$$o_{hj} := f(a_{hj})$$

f is a bijective function $f :]-\infty, \infty[\to]0, 1[$ because the output has to be $o_{hj} \in [0, 1]$. We use

$$f(a) := \frac{1}{1 + \exp(-a)}.$$

Such a feedforward multilayer perceptron can approximate any function after a suitable amount of training. Therefor known discrete values of this function are presented to the net. The net is expected to learn the function rule [2].
The behaviour of the net is changed by modification of the weights and bias values. The back-propagation learning algorithms we use to optimize these values is described later together with its parallelizations.

2.2 Preprocessing the input data

An efficient preprocessing of the data is necessary to input it into the net. All information must be normalized to fit into the interval $[0, 1]$. We assume that the necessary information is given for T weeks in the past. With the following definitions

$$\begin{aligned} ADV_i^t &:= \text{number of advertising days for article } i \text{ within week } t \\ SAL_i^t &:= \text{sale of article } i \text{ within week } t \\ MAXSAL_i &:= \max_{1 \le t \le T} \left\{ SAL_i^t \right\} \end{aligned}$$

we have decided to use the following inputs for each article i and week t:

$$adv_i^t := \frac{ADV_i^t}{6} \text{ (campaign days of 6 opening days)}$$

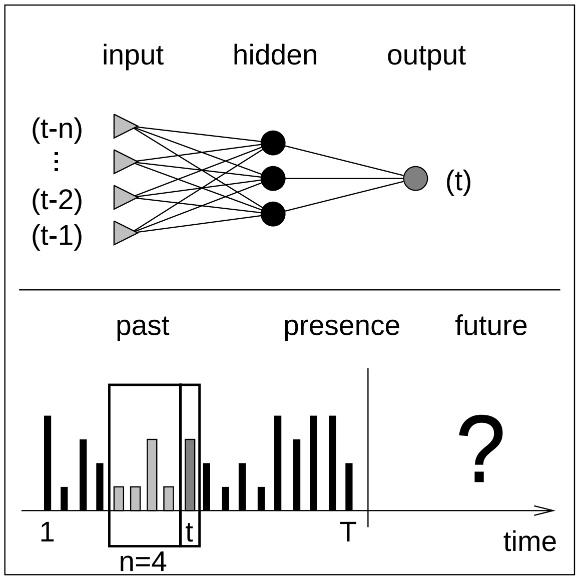


Figure 4: Feedforward multilayer perceptron for time series prediction

$$pri_i^t := \left\{ \begin{array}{lcl} 1.0 & : & \text{price increases} \\ 0.5 & : & \text{price keeps equal} \\ 0.0 & : & \text{price decreases} \end{array} \right\} \text{within week } t$$

$$sal_i^t := \frac{SAL_i^t}{MAXSAL_i} \cdot 0.8$$

For each article i and recent week t we use a three-dimensional vector:

$$vec_i^t := \left(adv_i^t, pri_i^t, sal_i^t\right)$$

For a week t in the future the vector is reduced by the unknown sale:

$$\widehat{vec}_i^t := \left(adv_i^t, pri_i^t\right)$$

To predict the sale for one article within a week t, we use a window of the last n weeks. So we have the following input vector for each article i:

$$input_i^t := \left(vec_i^{t-n}, vec_i^{t-n+1}, ..., vec_i^{t-1}, \widehat{vec}_i^t\right)$$

Because all the considered articles belong to one product group, we have quite a constant sales volume of all products. An increasing sale of one article in general leads to a decrease of the other sales. Due to this, we concatenate the *input* vectors of all p articles to get the vector given to the input layer:

$$INPUT^t := \left(input_1^t, input_2^t, ..., input_p^t\right)$$

The sale of article i within week t (sal_i^t) is the requested nominal value in the output layer that has to be learned by one net for this $INPUT^t$ vector. So we have p nets and the i-th net adapts the sale behaviour of article i. Therefore we have a training set with the following pairs (see figure 4):

$$(INPUT^t, sal_i^t) \text{ with } n \leq t \leq T$$

To forecast the unkown sale sal_i^{T+1} for any article i within a future week $T+1$ we give the following input vector to the trained i-th net:

$$INPUT^{T+1}$$

The output value of this net is expected to be the value sal_i^{T+1}, which has to be re-scaled to the value for the sale of article i within week $T+1$:

$$SAL_i^{T+1} = \frac{sal_i^{T+1} \cdot MAXSAL_i}{0.8}$$

2.3 Empirical results of the prediction

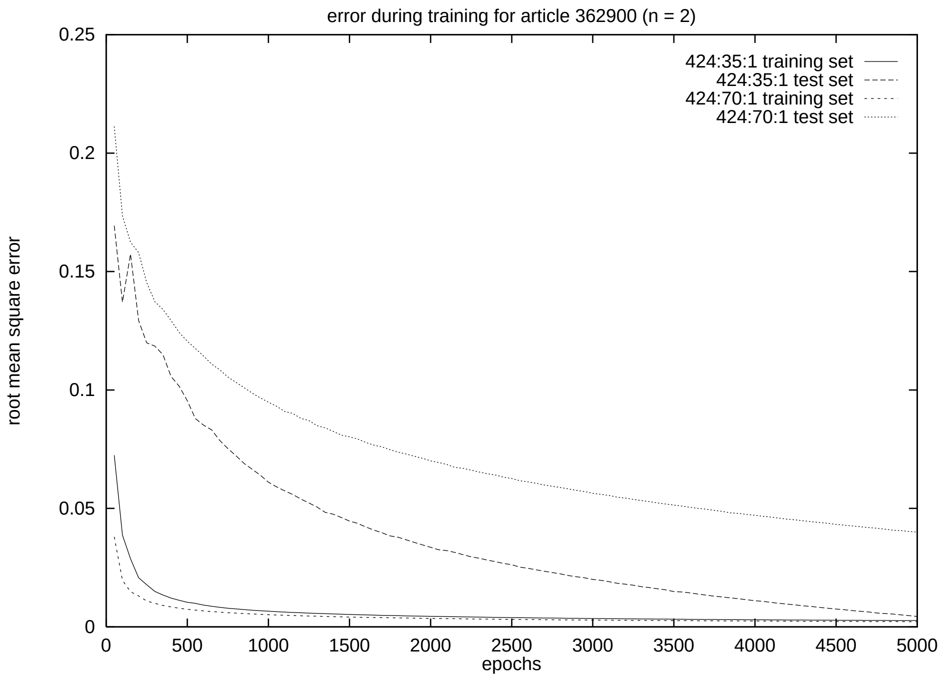


Figure 5: root mean square error of two nets for article 362900 with $n = 2$

Here we present the behaviour of eight nets. We have trained nets for

1. article 362900 respectively article 372138 (see figures 1 and 2)
2. with two respectively three weeks past information ($n = 2$ respectively $n = 3$)
3. where the number of hidden neurons is $\frac{1}{6}$ respectively $\frac{1}{12}$ of the number of input neurons.

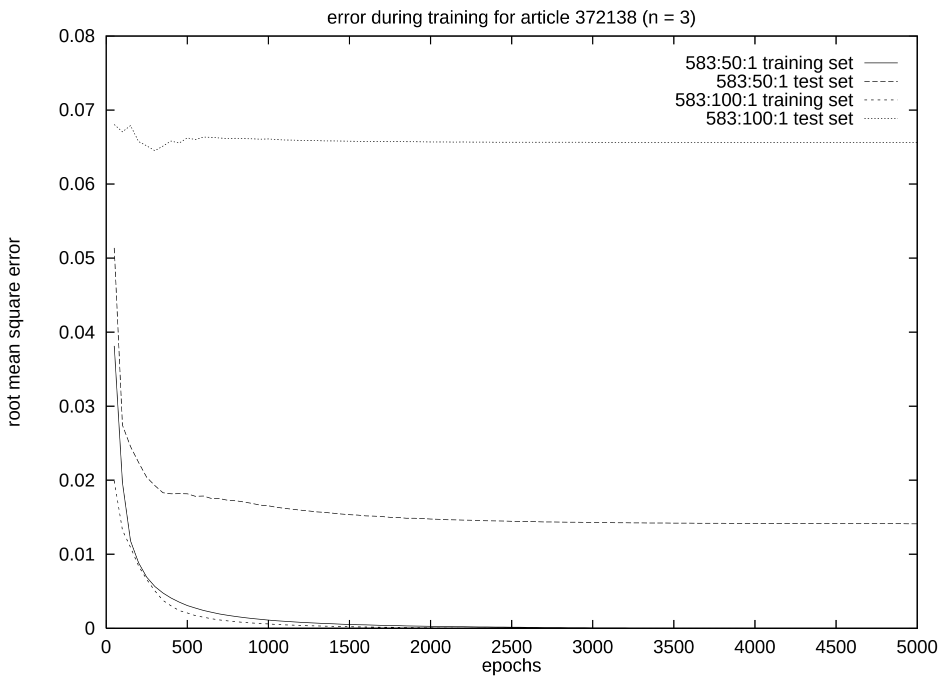


Figure 6: root mean square error of two nets for article 372138 with $n = 3$

We are using the information of 53 articles in the input layer. The topology of the net is described by the syntax: `(input neurons:hidden neurons:output neurons)`.
The given data is split into a training set (week 36/1994 to week 16/1995) and a test set (week 17/1995). The test set is not trained and only considered to check whether the net has generalized the behaviour of the time series.
With $n = 2$ we have 31 pairs in the training set and one in the test set, with $n = 3$ we have 30 pairs in the training set and one in the test set. The figures 5 and 6 show the root mean square error on the training and the test set, while the nets are learning 5000 epochs. This error is going down immediately on the training set, especially for the larger nets.

Table 1: Training times of different nets on SPARC 20-50

net sizes	424:35:1	424:70:1	583:50:1	583:100:1
5000 epochs	1955 sec	3896 sec	3670 sec	7345 sec

More important is the error on the test set — the prediction error. This is better for the smaller nets. They need more epochs to learn the rule of the time series, but because of this they can generalize their behaviour better.
The prediction error of the smaller nets in means of sales can be seen from figures 1

and 2. For the week 17/1995 the forecasted sales are drawn dotted. For both articles the error is smaller than one piece. The time for training the nets on a sequential SUN SPARC 20-50 can be seen from table 1.
Summarizing the results of the forecasting quality we can say that the error on the test set can be reduced to an acceptable low level. The re-transformation to the sale values shows that we can predict the sale for the next week with an sufficient accuracy after an enormous training effort.

3 Parallelization

To reduce the computation time for training, the back-propagation algorithm has been parallelized in three ways. Before the different parallelizations are described the back-propagation algorithm is explained.

3.1 Back-Propagation algorithm

The backpropagation algorithm consists of two phases: the forward phase where the activations are propagated from the input to the output layer, and the backward phase, where the error between the observed actual and the requested nominal value in the output layer is propagated backwards in order to modify the weights and bias values.

3.1.1 Forward Propagation

The idea of the forward phase is shown in figure 7. The weights of the needed receptive connections of neuron j are one row of the weight matrix. The following values are calculated:

$$\begin{aligned} \text{activation of neuron } j\text{: } a_j &:= \sum_i o_i \cdot w_{ji} + \theta_j \\ \text{output of neuron } j\text{: } o_j &:= f(a_j) = \frac{1}{1+\exp(-a_j)} \end{aligned}$$

3.1.2 Backward Propagation

For the backward phase (figure 7) the neuron j in the output layer calculates the error between its actual output value o_j, known from the forward phase, and the expected nominal target value t_j:

$$\delta_j := (t_j - o_j) \cdot \underbrace{f'(a_j)}_{o_j \cdot (1-o_j)}$$

The error δ_j is propagated backwards to the previous hidden layer.
The neuron i in a hidden layer calculates an error δ'_i that is propagated backwards again to its previous layer. Therefor a column of the weight matrix is used.

$$\delta'_i := (\sum_j w_{ji} \cdot \delta_j) \cdot \underbrace{f'(a_i)}_{o_i \cdot (1-o_i)}$$

To minimize the error the weights of the projective edges of neuron i and the bias values in the receptive layer have to be changed. The old values have to be increased by:

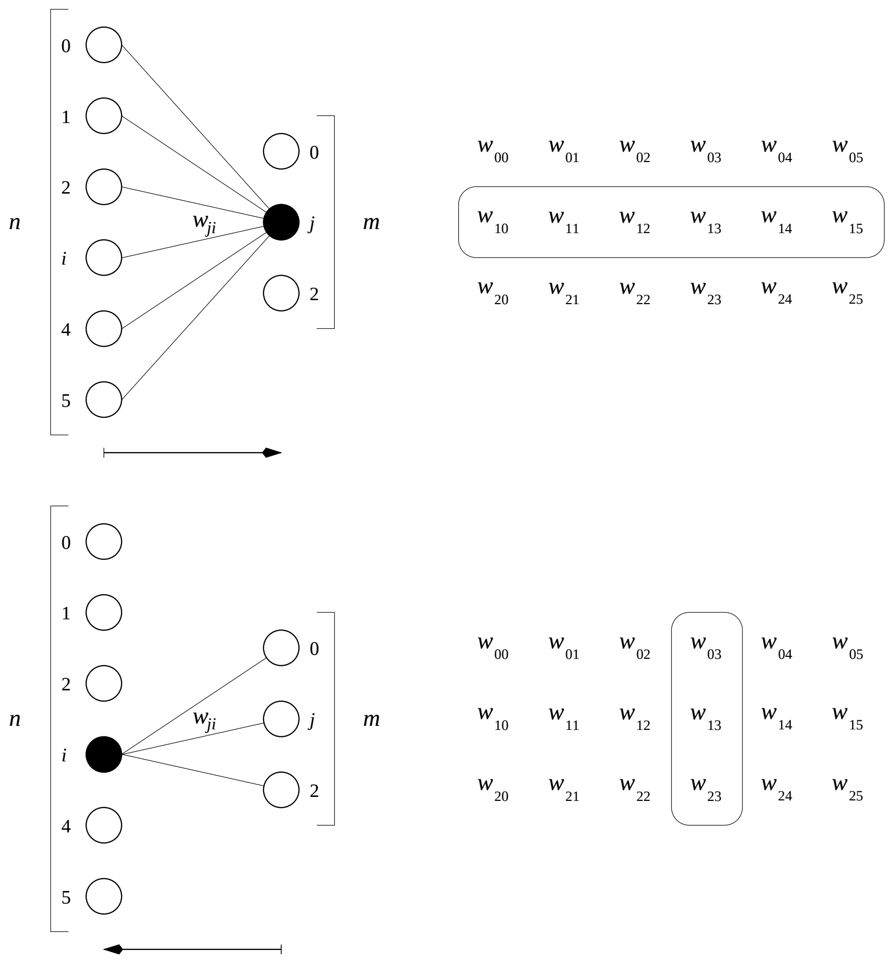

0
1
2
i
4
5
n
w_ji
0
j
2
m
w_00 w_01 w_02 w_03 w_04 w_05
w_10 w_11 w_12 w_13 w_14 w_15
w_20 w_21 w_22 w_23 w_24 w_25

Figure 7: Forward and backward propagation

$$\begin{aligned} \Delta w_{ji} &= \eta \cdot \delta_j \cdot o_i \\ \Delta \theta_j &= \eta \cdot \delta_j . \end{aligned}$$

η is the training rate and has an empirical value: $\eta \approx 1$.
The back-propagation algorithm optimizes the error by the method of gradient descent, where η ist the length of each step.

3.1.3 Modified back-propagation

The optimal configuration of a feedforward multilayer perceptron network with its input, hidden and output layers is very difficult to find. Too many hidden neurons lead to a net that is not able to extract the function rule and takes more time for learning. With a lack in hidden neurons it is not possible to reach any error bound. Input and output layers are determined by the problem and the function that is to be approximated.

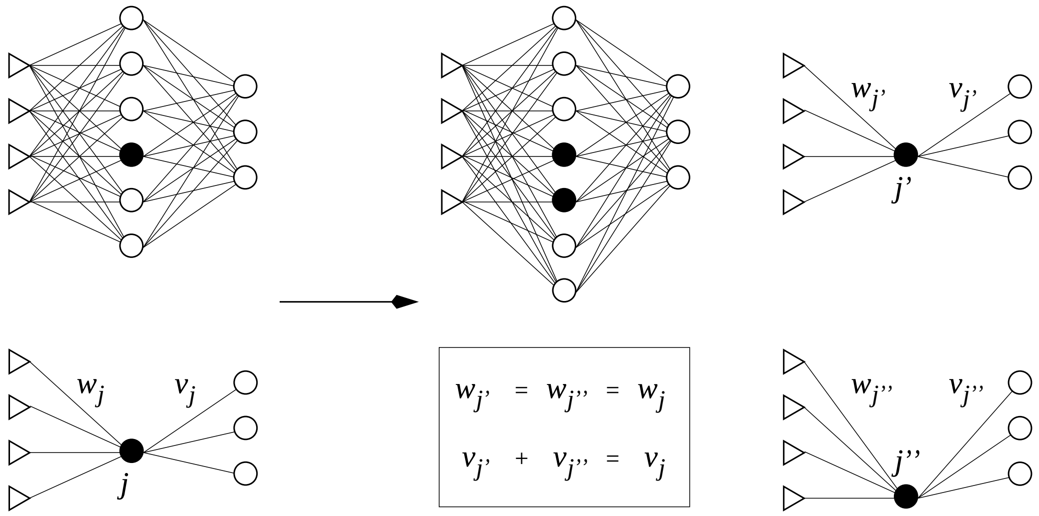


Figure 8: Modified back-propagation with neuron splitting

The modified back-propagation algorithm [9] is able to increase the quality of a net by monotonic net incrementation. The training starts with a net of a few hidden neurons. Badly trained neurons are split periodically while learning the training set. The old weights are distributed at random between the two new neurons (cf. Figure 8). This is done until a maximum number of neurons within a hidden layer is reached. By training the net with the modified back-propagation algorithm a better minimum of the error is reached in shorter time.

3.2 Batch Learning

For parallel batch learning the training set is divided and learned separately with some identical copies of the net in parallel [6]. The weight corrections are summed up and globally corrected in all nets after each epoch.
Communication is only necessary for the calculation of the global sum of the weight corrections after each epoch. In addition to this a global broadcast has to be performed after the master node has calculated the random numbers for the new weights after splitting, but this happens very rarely.
The batch learning is different from the on-line training concerning the convergence speed and the quality of approximation. There are training problems where the batch learning algorithm is more suitable than the on-line training and vice versa.

3.3 On-line training

The on-line training changes all the weights within each backward propagation after every item from the training set. Here the parallelization is very fine-grained. The vector-matrix-operations have to be calculated in parallel. This needs a lot of communication.

The responsibility for the calculation of the activations and output of the neurons is distributed among the processors. The hidden and output layers are partitioned into p disjoint sets and each set is mapped on one of the p processors. Therefor the weight matrices are distributed in rows and these are distributed among the processors. When splitting is necessary the neuron and its weights are broadcasted around the processor cycle. The responsibility for this neuron is given to the first processor with the lowest load.
After each step of the propagation the new output vector in the layer has to be broadcasted because each processor needs the whole receptive layer to calculate the activations of its own neurons for the next step.
The backward phase is more complicated: for the error propagation each processor needs the columns of the weight matrices to calculate the error in the previous layer, but the weights are stored and updated in the rows of the weight matrix due to the operations in the forward propagation. There are two different methods to implement the parallel calculation of the errors.

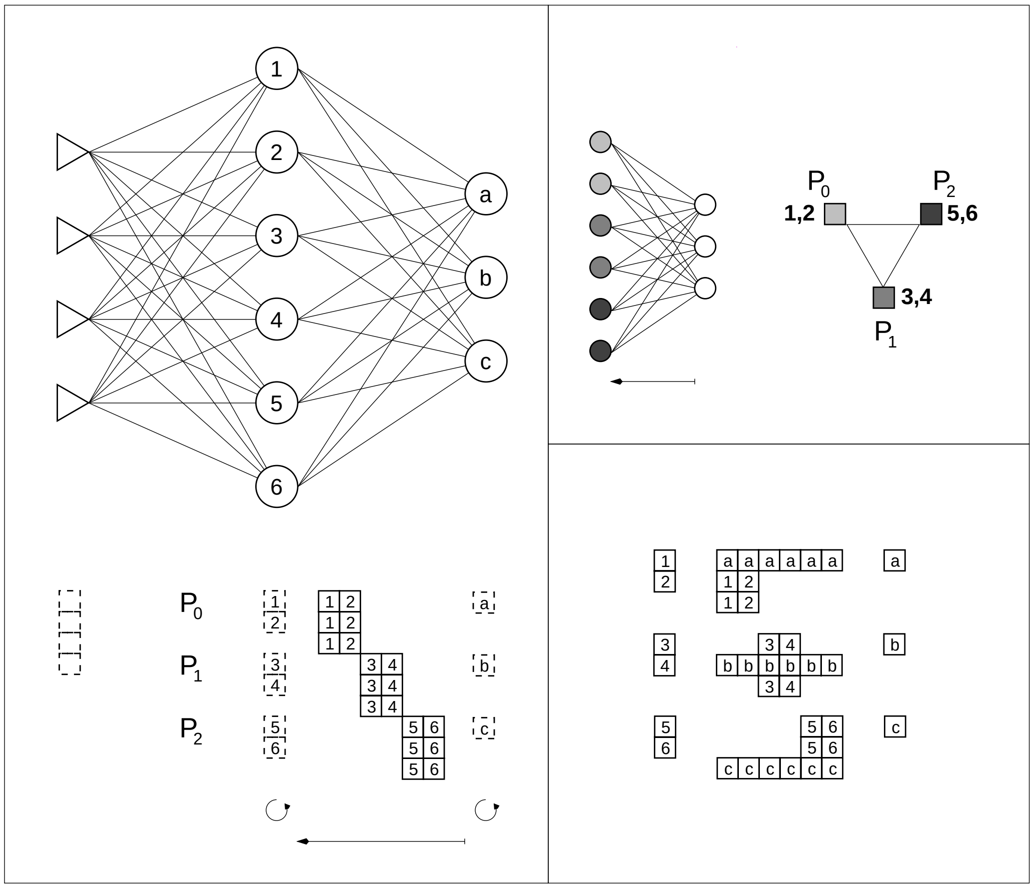


Figure 9: Parallel on-line backward propagation of Yoon et.al.

3.3.1 Parallel backward propagation of Morgan et.al.

The first method is described by Morgan et.al. [7]. Here the matrix products are calculated in parallel, while the subsums are sent around the processor cycle. At the end each processor has found its δ'.

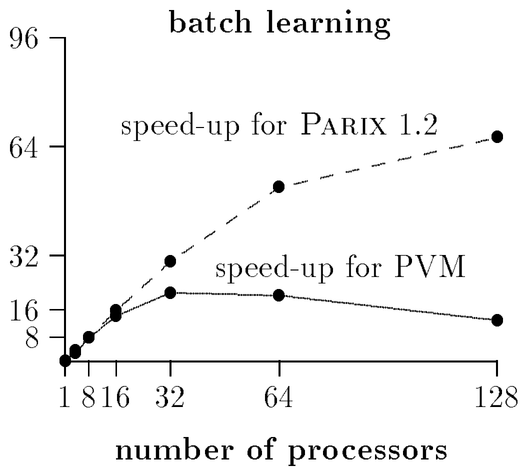
$$\delta'_k := (\sum_j w_{jk} \cdot \delta_j) \cdot o_k \cdot (1 - o_k)$$

The advantage of this parallelization is that the weight matrices are stored and modified only once, distributed on all processors. Therefor this parallelization has to calculate the partial sums of the error between each of the $p-1$ communication steps.

3.3.2 Parallel backward propagation of Yoon et.al.

To reduce communication time between the processors we have also implemented the idea of Yoon, Nang and Maeng [8]. For each parallel calculated activation of a neuron its receptive and projective weights are stored on the responsible processor. Figure 9 shows the distribution of the neurons and the weight matrices among three processors. In each backward step one processor updates the weights of its projective and receptive neurons. So we have to store the weight matrices twice with the same overhead in calculation. The advantage of this parallelization is that we can do the $p-1$ communications for the broadcast without interruptions after the error calculation is finished.

4 Implementations and experimental speed-ups



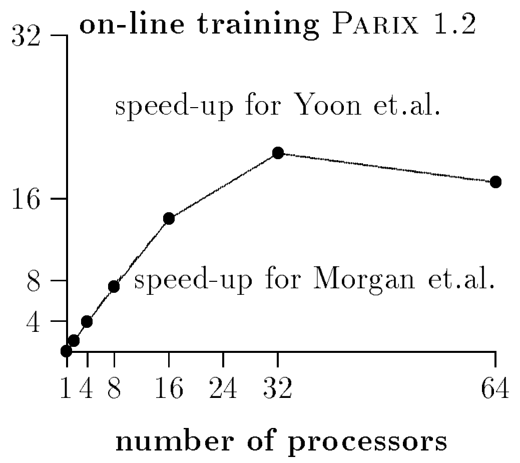


Figure 10: Speed-ups for GCel with T805

The parallel implementations take place on Transputer-based Parsytec systems. We use a GCel with T805 and a GC/PP with PowerPC 601 CPUs and Transputer communication links. The parallelization runs with both the runtime environments Parix Version 1.2 (GCel) resp. 1.3 (GC/PP) and PVM/Parix 1.1.

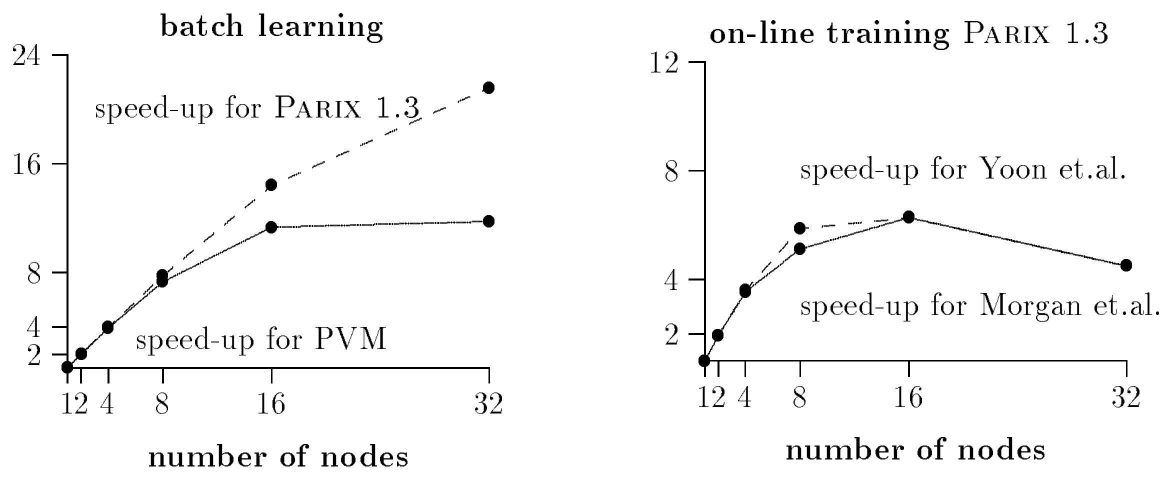


Figure 11: Speed-ups for GC/PP with PowerPC

The figures 10 and 11 show the reached speed-ups with both systems. The results show that the batch learning algorithm scales better than the on-line training. The reason for this are the very high communication demands for each pair trained in contrast to the one communication per epoch for parallel batch learning. As expected the on-line parallelization of Morgan et.al. is worse for small nets and less processors than that of Yoon et.al. according to the uninterrupted communications. This difference vanishes for relatively large number of neurons and processors.
The comparison of both hardware architectures shows that the T805 system scales better than the GC/PP. The computational power of the PowerPC is much higher than that of the T805. But the ratio of the communication and CPU performance is higher on T805 systems.

5 Conclusions and future research

It has been shown that feedforward multilayer perceptron networks can learn to approximate the time series of sales in supermarkets. For a special group of articles neural networks can be trained to forecast future demands on the basis of the past data.
The back-propagation algorithm has been parallelized to reduce the enormous training times. Three different parallelizations have been implemented on Parsytec parallel systems. The batch learning is best for large training sets on systems with bad communication performance. The on-line parallelization works well with large numbers of neurons and scales on systems with very good communication performance in proportion to the computing performance.
The variant of Yoon et.al. has less communication demands than that of Morgan et.al. This advantage appears especially for small nets.
For the future the modelling of the input vectors should be improved: especially season and holiday information have to be given to the net; the value of changing prices can

be modelled quantitatively. One important aim will be the reduction of input neurons. By correlation analysis some of the hundreds of single time series should be merged or denied. This will lead to smaller nets with shorter training times.
To handle the complex amount of data within an acceptable time the presented parallelizations of the back-propagation algorithm are worthwile and necessary.

Acknowledgement

Thanks to the *Paderborn Center for Parallel Computing* (PC^2) at the University of Paderborn, Germany. All the measurements with T805 and PowerPC processors took place there.

References

[1] D.E. Rumelhart, G.E. Hinton, R.J. Williams. *Learning internal representations by error propagation.* In D.E. Rumelhart and J.L. McClelland (Eds.), Parallel Distributed Processing: Explorations in the Microstructure of Cognition, Vol. 1, pp. 318-362, MIT 1987.

[2] D.E. Rumelhart, B. Widrow, M.A. Lehr. *The Basic Ideas in Neural Networks.* Communications of the ACM Vol.37, No.3, pp. 87-92, March 1994.

[3] Z. Tang, P.A. Fishwick. *Feed-forward Neural Nets as Model for Time Series Forecasting.* TR91-008, University of Florida, 1991.

[4] V.R. Vemuri, R.D. Rogers. *Artificial Neural Networks – Forecasting Time Series.* IEEE Computer Society Press 5120-05, 1994.

[5] P.C. McCluskey. *Feedforward and Recurrent Neural Networks and Genetic Programs for Stock Market and Time Series Forecasting.* Master thesis CS-93-36, Department of Computer Science, Brown University, Providence, Rhode Island, September 1993.

[6] A. Petrowski. *A pipelined implementation of the back-propagation algorithm on a parallel machine.* Artificial Neural Networks, pp. 539-544, Elsevier Science Publishers B.V. (North-Holland), 1991.

[7] N. Morgan, J. Beck, P. Kohn, J. Bilmes, E. Allman, J. Beer. *The Ring Array Processor: A Multiprocessor Peripheral for Connectionist Applications.* Journal of Parallel and Distributed Computing 14, pp. 248-259, Academic Press Inc., 1992.

[8] H. Yoon, J.H. Nang, S.R. Maeng. *A distributed backpropagation algorithm of neural networks on distributed-memory multiprocessors.* Proceedings of the 3rd symposium on the Frontiers of Massively Parallel Computation, pp. 358-363, IEEE 1990.

[9] I. Glöckner. *Monotonic incrementation of backpropagation networks.* Proceedings of the International Conference on Artificial Neural Networks (ICANN93), 1993.