Parallel Neural Network Learning Through Repetitive Bounded Depth Trajectory Branching

Iuri Mehr   Zoran Obradović*
School of Electrical Engineering and Computer Science
Washington State University
Pullman, WA 99164-2752

Abstract

The neural network learning process is a sequence of network updates and can be represented by sequence of points in the weight space that we call a learning trajectory. *In this paper a new learning approach based on repetitive bounded depth trajectory branching is proposed. This approach has objectives of improving generalization and speeding up convergence by avoiding local minima when selecting an alternative trajectory. The experimental results show an improved generalization compared to the standard back-propagation learning algorithm. The proposed parallel implementation dramatically improves the algorithm efficiency to the level that computing time is not a critical factor in achieving improved generalization.*

1 Introduction

With most currently known algorithms, neural network learning has to be done sequentially resulting with a large amount of computational time and non-optimal generalization. For example, the well known back-propagation algorithm requires a huge amount of nonlinear computations and scales up poorly as tasks become larger and more complex [1]. By increasing the number of hidden units and layers we also increase the probability of experiencing the so called *local minima problem* where the error function is not minimized by further learning (although there exists a lower minimum) [6]. We believe that more efficient and more accurate learning is possible through parallelization.

The state of a neural network can be represented as a single point in the weight space. Each update of the neural network generates a new point in this space. Consequently the learning process can be represented by a sequence of points in the weight space that we call a *learning trajectory.* We will refer to the *standard trajectory* as the trajectory of the weight vector during the learning phase of a *standard algorithm* which can be any gradient-descent learning technique (back-propagation in our experiments it is). The objective of this paper is both more efficient and more accurate neural network learning by exploring a number of learning trajectories in parallel in order to find one that avoids local minima. Trajectories have the same starting point and the best one is selected after a *bounded depth branching* in weight space during the learning phase. At the first branching point, which could be after a single pattern presentation or a whole epoch, a number of new neural network structures are constructed with the same architecture as the original one but with the weight vector branching in various directions. At each following branching point the number of neural networks is increased. When the number of trajectories reaches the maximum supported on the hardware a cross-validation test is performed on all generated networks (trajectories) and a small number of trajectories are kept for further evolution again using bounded depth branching. The proposed algorithm allows a faster minimization of the error function (a smaller number of epochs for convergence) as a result of frequent comparison among several learning trajectories.

2 Branching Algorithms

In real life problems a neural network's error function usually generates a complex surface. Consequently a better trajectory could be found in the vicinity of the standard one using small variations in the branching parameters. In the proposed algorithm new branching points are generated systematically during the weight updating phase and one new trajectory is

*Z. Obradović's research is sponsored in part by the NSF research grant NSF-IRI-9308523. He is also affiliated with the Mathematical Institute, Belgrade, Yugoslavia.

added at each branching point. The *general branching algorithm* implies several steps:

1. Initially follow a single learning trajectory corresponding to the standard learning algorithm.
2. Generate new branching points after a specified number of learning steps.
3. At each new branching point start exploring new learning trajectories in addition to the existing ones.
4. Continue with step 2 until 2^K trajectories are generated such that $2^K \leq C < 2^{K+1}$ where C is maximum number of trajectories supported by the hardware.
5. Using a cross-validation test on all existing trajectories select M of those and abandon the remainder.
6. Starting from M selected trajectories of step 5 continue constucting and exploring new trajectories on new branching points until $2^L M$ trajectories are constructed such that $2^L M \leq C < 2^{L+1} M$.
7. Continue with step 5 until the training error is within a prespecified tolerance or for a prespecified number of steps.

In the general branching algorithm initially there is a single trajectory corresponding to the standard algorithm. At the first branching point the learning trajectory is split into one corresponding to the standard algorithm and one constructed using branching parameters (discussed in detail in Section 4). The selection of trajectories (step 5) is based on performance on a cross-validation data set. This data set should be fairly small due to the large number of trajectories and computing time requirements. In our experiments we select three trajectories ($M = 3$): the very best on the cross-validation test, the standard (generated by the standard algorithm) and a random choice of the existing trajectories. We propose two ways of generating new branching points. In the short term branching algorithm new branching points are generated on all existing trajectories. In contrast the long term branching algorithm generates a single new branching point on each of the M selected trajectories. The details of these two algorithms follow.

2.1 Short Term Branching Algorithm

In the short term branching algorithm new branching points are generated after each training pattern is presented (Figure 1).

After weight initialization the first training pattern is used to update the network and the learning trajectory is split. The number of trajectories is doubled on each new training pattern (each trajectory is split in two). Let C be the maximum number of trajectories supported by the hardware. Once B trajectories are

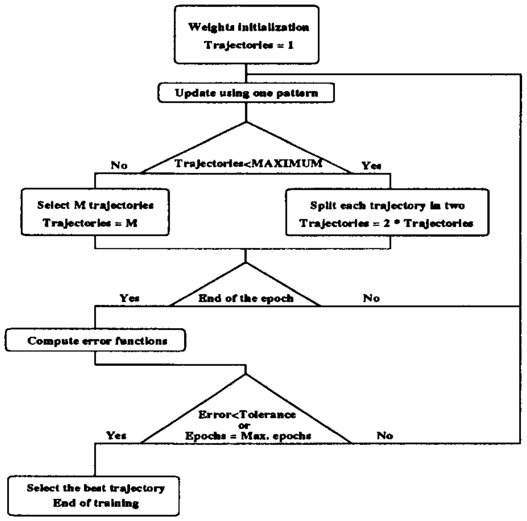

Figure 1: Short term branching algorithm

generated ($B = 2^L M$ where $2^L M \leq C < 2^{L+1} M$), M of them are selected and all others are abandoned. This selection is performed using a cross-validation data set which should be fairly small in order to avoid a long testing time, but at the same time should be representative enough for the learning problem distribution. The process continues and after each epoch the error function is computed for each trajectory. The training terminates when the error function becomes smaller than a specified tolerance on one of the existing trajectories or when a prespecified number of epochs is reached.

It is easy to see that in the short term branching algorithm the number of training patterns used between two selections of M trajectories (step 5 of the general branching algorithm) is given by $\lceil log_2 B \rceil$ on initial branching (steps 1-4 of the general algorithm) and by $\lceil log_2 \frac{B}{M} \rceil$ otherwise (step 6 of the general algorithm). Consequently, a large variety of useful learning trajectories is generated quickly (after a small number of update steps) in the short term branching algorithm.

2.2 Long Term Branching Algorithm

The long term branching algorithm is similar to the previous one except the location of branching points and the number of generated trajectories is different. In contrast to the short term, in the long term branching algorithm (Figure 2), a new trajectory is generated at the end of each epoch. Branching points are always located on one of the M trajectories selected at the

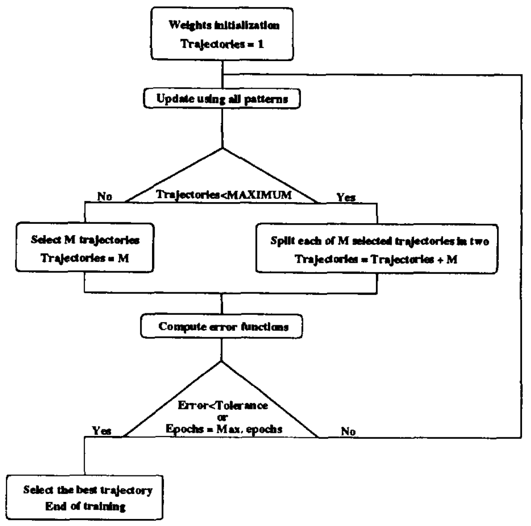

Figure 2: Long term branching algorithm

previous cross-validation test (rather than on all existing trajectories as in the short term branching algorithm). In the long term branching algorithm the number of epochs between two selections of M trajectories (step 5 of the general branching algorithm) is given by B on initial branching (steps 1-4 of the general algorithm) and by $\frac{B}{M}$ otherwise (step 6 of the general algorithm). Consequently in the long term branching algorithm, a large variety of useful learning trajectories is generated after a much larger number of update steps than in the short term branching algorithm. Although in the short term branching algorithm a large variety is obtained quickly (after a few training patterns), those trajectories are not necessarily of high quality (a few patterns are not necessarily representative for the whole input data set). Superiority of one or the other algorithm must be determined experimentally.

3 Parallelization

It was observed during simulations that a sequential implementation of the proposed branching algorithms takes much more time to achieve an acceptable value for the error function than a standard back-propagation (to be discussed in the results section). The most time consuming phase is the cross-validation check on all trajectories. It occurs when a system limit is reached due to the amount of generated trajectories. The computing time can be drastically reduced by distributing all trajectories on separate processing units and running the back-propagation on them in parallel. In such a distributive environment cross-validation can be easily performed on all trajectories in parallel. In Section 3.1 the proposed parallelization is discussed in detail and in 3.2 the efficiency of the parallel algorithm is analyzed.

3.1 Parallel Branching Algorithm

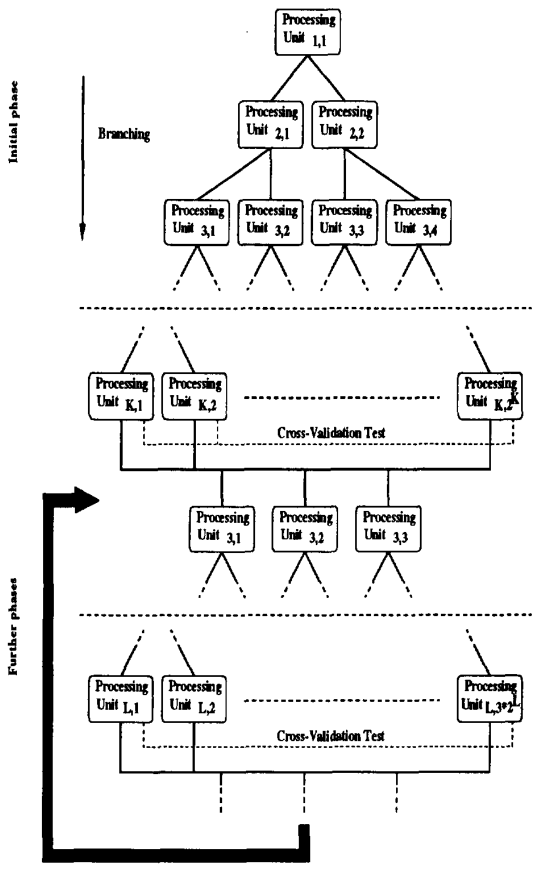

Figure 3: Short term branching paralelization

Figure 3 presents the initial phase (steps 1-4 from Section 2) and further phases (steps 5-6) for the short term branching algorithm implemented on a parallel system. The $Unit_{1,1}$ is a processor loaded with the initial neural network parameters. When an updating step occurs a new neural network is constructed by copying the modified network parameters to a new processor $Unit_{2,1}$. The original network could be also copied to a new processor $Unit_{2,2}$ or kept on the old one that we rename as $Unit_{2,2}$. The process repeats for K steps, when 2^K processing units contain 2^K different neural networks. Obviously this number is limited by the maximum number of available processing units. Each of these units then performs in parallel a cross-validation test evaluating its local neural network. After selection of M of those networks ($M = 3$ on Figure 3) the algorithm continues for L branching steps, such that $M \cdot 2^L \leq 2^K < M \cdot 2^{L+1}$. Again M trajectories are selected using parallel cross-validation and the process repeats as indicated by the arrow in the figure. The parallelization of the long term branching algorithm is similar.

Observe that the amount of interprocessor communication in the proposed parallel algorithm is quite small. In fact, the algorithm has a tree structure with one-directional communication from parent to child units only. Consequently, the parallel learning algorithm is applicable for a hypercube implementation using one dimension of egdes per step and using consecutive dimensions of edges in consecutive steps. Such algorithms (called *normal* algorithms) are suitable for efficient implementation on any bounded-degree variation of the hypercube such as the butterfly and the shuffle-exchange graph network [4].

3.2 Performance Analysis

In our analyzis P is the number of available processing units; t_t is the time required for network parameters transfer between two processors; t_v is the time required for a single cross-validation test and t_c is the time required for selecting the three branches after a cross-validation step. For simpler analyses in branching algorithms after a cross-validation testing we will assume selection of three trajectories ($M = 3$).

Let us first analyze the short term branching algorithm. By c_p we denote the computing time needed by the standard algorithm to update the neural network when a new training pattern is presented. If the neural network is updated after each pattern presentation then the computing time of an epoch for the standard algorithm on a sequential machine is

$$T_s = Nc_p$$

where N is size of the training set. In the short term branching algorithm the number of branches for a cross-validation test (except for the initial test) is

$$B = 3 \cdot 2^{\lceil log_2 \frac{P}{3} \rceil}$$

If we implement the short term branching algorithm on a sequential machine emulating P processing units, the computing time of an epoch (except the initial epoch) is

$$T_{s,s} \approx \frac{N}{\lceil log_2 \frac{P}{3} \rceil} \left[c_p \left(B + \frac{B}{2} + \cdots + 3 \right) + t_v B + t_c \right]$$

Consequently,

$$\frac{T_{s,s}}{T_s} \approx \frac{1}{\lceil log_2 \frac{P}{3} \rceil} \left[(B + \frac{B}{2} + \frac{B}{4} + \cdots + 3) + \frac{t_v B + t_c}{c_p} \right]$$

which shows that a sequential implementation of the short term branching algorithm is computationally very expensive.

On the other side, the computing time of an epoch (except the initial epoch) for the short term branching algorithm on a parallel system of P processors is

$$T_{p,s} \approx T_s + \left(t_c + t_v + t_t log_2 \frac{P}{3} \right) \frac{N}{\lceil log_2 \frac{P}{3} \rceil}$$

So, for a parallel implementation

$$\frac{T_{p,s}}{T_s} \approx 1 + \frac{t_t}{c_p} + \frac{t_c + t_v}{c_p \lceil log_2 \frac{P}{3} \rceil} \qquad (1)$$

which shows that the algorithm is more suitable for parallel system rather that distributed system implementation where t_t is significantly larger.

Let us now analyze the long term branching algorithm. In this case the underlying standard sequential algorithm performs one neural network update per one epoch. We define c_e to be the computing time between two neural network updates by this standard algorithm. As earlier let B be the number of branches for the cross-validation test (except the initial test).

If we implement the long term branching algorithm on a sequential machine emulating P processing units, the computing time between two cross-validation ($\frac{B}{3}$ epochs) is

$$T_{s,l} \approx 3 \left(1 + 2 + \cdots + \frac{B}{3} \right) c_e + Bt_v + t_c$$

or, after simplification

$$T_{s,l} \approx B \frac{B+3}{6} c_e + Bt_v + t_c$$

The computing time for those $\frac{B}{3}$ epochs on a parallel system of P processors is

$$T_{p,l} \approx \frac{B}{3}(c_e + t_t) + t_v + t_c$$

The computing time for $\frac{B}{3}$ epochs for the standard algorithm is

$$T_l = \frac{B}{3} c_e$$

Consequently

$$\frac{T_{s,l}}{T_l} \approx \frac{B+3}{2} + \frac{3t_v}{c_e} + \frac{3t_c}{Bc_e}$$

and

$$\frac{T_{p,l}}{T_l} \approx 1 + \frac{t_t}{c_e} + \frac{3(t_c+t_v)}{Bc_e} \qquad (2)$$

which shows that a sequential implementation of the long term branching algorithm is still expensive (B is large) but a parallel implementation is time efficient.

The analyzis also shows that a parallel implementation of the long term branchin g algorithm is more efficient than the short term branching implementation ($\frac{T_{p,l}}{T_l} < \frac{T_{p,s}}{T_s}$). However, better experimental generalization results were obtained using the short term branching algorithm as will be discussed in the following section.

4 Results

The proposed branching technique can be used to improve the generalization of any gradient descent learning algorithm. In our experiments back-propagation is used as the standard algorithm. Theoretical results concerning the efficient method for branching as well as simulation results are presented further in this section.

4.1 Branching Trajectories Construction

It is known that the back-propagation algorithm is not an optimal minimization technique. Figure 4 represents an example of a standard trajectory generated by the back-propagation algorithm. From the branching point (x_0, E_0) a better trajectory could be found in the neighborhood of the standard trajectory as indicated in the figure.

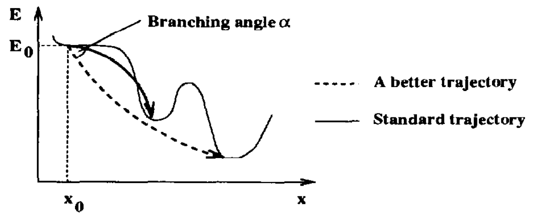

Figure 4: Standard and *better* trajectories

For non-trivial learning problems it is reasonable to assume that there are such better trajectories in the vicinity of the standard one. Consequently, for the construction of branching trajectories we decided to use limited variations in the branching parameter α (the angle between the standard and the new trajectories).

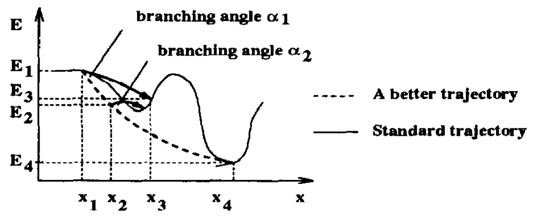

Figure 5: Avoiding local minimum using branching

In branching algorithms it is important to construct a variety of trajectories. For example in Figure 5 at the branching point (x_2, E_2) the training error is smaller than in point (x_3, E_3) obtained by the back-propagation from (x_1, E_1). However the branching should not be stopped at (x_2, E_2) since the global minimum is at (x_4, E_4). Consequently the branching parameter α has to be large enough to generate a trajectory that leads to the global minimum. Also, branching would not be efficient using small learning rates because branching trajectory will stay too close to the standard one.

We will derive some expressions for the branching parameters in order to determine efficient methods for the construction of branching trajectories consistent with previous observations. Let

$$W = [w_1, \ldots, w_p]$$

be a weight vector which is composed by all weights in the network. A network updating step implies a change in W which will eventually affect all components

$$W + \Delta W = [w_1 + \delta w_1, \ldots, w_p + \delta w_p]$$

The weight space is spanned by $w_1, w_2, \ldots, w_p$, and so ΔW could be expressed as a vector in this space. Let

$$\Delta W = a_1\overline{w_1} + a_2\overline{w_2} + \cdots + a_p\overline{w_p}$$

where $\overline{w_i}$, $i = 1, \ldots, p$, are normalized vectors representing a basis in the weight space and $a_i = -\eta\frac{\partial E}{\partial w_i}$ is the weight update in w_i direction. We are looking for a vector ΔW^* which will generate the first point $W^* = W + \Delta W^*$ of a new trajectory. This unknown vector could be expressed as

$$\Delta W^* = b_1\overline{w_1} + b_2\overline{w_2} + \cdots + b_p\overline{w_p}$$

Let α be the angle between ΔW and ΔW^*. The scalar product will give the expression for $cos\alpha$

$$cos\alpha = \frac{\Delta W \Delta W^*}{|\Delta W||\Delta W^*|}$$

Or, in terms of projection in the weight space

$$cos\alpha = \frac{\sum_{i=1}^{p} a_i b_i}{\sqrt{\sum_{i=1}^{p} a_i^2}\sqrt{\sum_{i=1}^{p} b_i^2}} \tag{3}$$

We assumed earlier that there is a better trajectory in the vicinity of the standard one. We can use equation (3) to constrain the angle between these two trajectories.

Another desired feature of a branching algorithm is computational efficiency. A simple efficient approach is to compute the branching vector ΔW^* as

$$b_i = \begin{cases} 0 & \text{if i=j} \\ a_i & \text{otherwise} \end{cases} \tag{4}$$

However that could generate too small variation for a large p as now

$$cos\alpha = \sqrt{1 - \frac{a_j^2}{\sum_{i=1}^{p} a_i^2}}$$

For small p an acceptable large variation can be obtained by selecting $a_j = max\{a_i\}$.

Based on all those observations we explored five methods for determining branching trajectories. In our experiments the first neural network $W + \Delta W^* = [w_1 + b_1, \dots, w_p + b_p]$ of a new trajectory W^* is constructed from the previous W network using one of the following methods:

1. Compute b_i, $i = 1, \dots, p$, using equation (4), where j is given by $a_j = min\{a_i\}$.
2. Compute b_i, $i = 1, \dots, p$, using equation (4), where j is given by $a_j = max\{a_i\}$.
3. Compute b_i as $[b_1, \cdots, b_p] = [0, \cdots, 0, a_{k+1}, \cdots, a_p]$ where k is the largest integer such that $\alpha < 45°$ (α is computed using equation (3)).
4. Compute b_i as $[b_1, \cdots, b_p] = [a_1, \cdots, a_{j-1}, 2a_j, a_{j+1}, \cdots, a_p]$, where j is given by $a_j = max\{a_i\}$.
5. Compute b_i, $i = 1, \dots, p$, as $b_i = -\eta^* \frac{\partial E}{\partial w_i}$, $\eta^* \neq \eta$, where η and η^* are the learning rates for the standard back-propagation trajectory and for the new trajectory.

Method 1 generates very small branching angles and is useful only for large learning rates. A large branching angle is obtained using method 2. However it still does not guarantee success if the update process consists of only the largest weight change. Method 3 ensures an appropriate variety in exploring the weight space but it implies a large amount of computation. Method 4 is similar to method 2 except that it achieves a larger variety of trajectories. Finally, method 5 uses a number of different learning rates with the objective to explore a large neighborhood along the standard trajectory. Observe that in method 5 there is no need to compute the branching angle since $cos\alpha = 1$. However, this simple method does not give good generalization results as it will be discussed in the next section. In the following section we present a quantitative approach to the comparison between these five methods.

4.2 Simulation results

We focused our simulation on two classification problems previously used in benchmark tests. In both problems input vectors should be classified to one of two classes. The first is the well known two spirals problem [3] which is designed to be hard to learn and the second is a real-life problem concerning breast-cancer diagnosis [5].

Algorithm	Run time	Tr.err.
Standard back-propagation	45 min.	10%
Construction method 1	45 min.	46%
Construction method 2	45 min.	40%
Construction method 3	45 min.	44%
Construction method 4	45 min.	40%
Construction method 5	45 min.	45%

Table 1: Trajectory construction methods comparison

The sequential implementation of the branching algorithms is significantly more time consuming than the standard back-propagation. Consequently our first goal was to test which of the five proposed methods for construction of the branching trajectories (discussed in the previous section) generates most useful branches in the same amount of time needed for standard back-propagation convergence. Table 1 presents a comparison between the proposed trajectory construction methods. The results where obtained using a neural network structure with 2 inputs, 6 hidden units and one output trained for the two spirals problem (learning rate $\eta = 0.05$, 50 training patterns).

As expected, none of the five networks corresponding to the proposed trajectory construction methods converged in 45 minutes using sequential short term branching algorithm. The results confirm theoretical assertions made in the previous section concerning quality of learning for each of the five methods. Methods 2 and 4 appeared to generate most useful trajectories in a given amount of time. The removal of the largest weight change (method 2) was slightly more efficient and so for further testing of the branching algorithms we used this trajectory construction method.

Data	Spirals	Breast cancer
Configuration	2-6-1	10-10-1
Learning rate	0.05	0.05
Number of patterns	100	250
Training error	21%	4%
Generalization	73%	76%
Epochs	50000	25000
Run time	45 min.	65 min.

Table 2: Standard back-propagation

Data	Spirals	Breast cancer
Configuration	2-6-1	10-10-1
Learning rate	0.1	0.4
Number of patterns	100	250
Training error	18%	3.5%
Generalization	77%	78%
Epochs	7000	5000
Sequential run time	14 hours	24 hours
Parallel run time	47 min.	63 min.

Table 3: Short term branching algorithm

Tables 2 and 3 present the network structure, learning parameters and the results obtained on those two problems for the standard back-propagation and for the short term branching algorithm. The structure of the neural network architecture for each of the problems was selected according to the complexity of the input data space. The training process stopped when learning trajectories became stationary. In the branching algorithm we used larger learning rates than in the back-propagation algorithm in order to reduce the number of epochs for convergence, which turned out to be a valid assumption. In both experiments the branching algorithm converged in smaller number of epochs with improved generalization. Running the standard back-propagation for only 7000 epochs (as for the branching algorithm) will result in a large training error (32% after 7000 versus 21% after 50000 epochs) and a poor generalization ($\approx$60% versus 73%). Using learning rate $\eta = 0.1$ (as for the branching algorithm) the standard back-propagation requires a much longer time for convergence than using $\eta = 0.05$.

For the parallel run time we considered $P = 256$ processing units organized in a mesh structure like the Touchstone Delta System [2]. The time required for branch selection t_c, for network parameters transfer t_t, for cross validation test t_v, and for network update using one pattern c_p were computed based on Touchstone Delta System characteristics. For a neural network structure corresponding to the two spirals problem (2 inputs, 6 hidden units, one output) we have $t_c = 1ms$, $t_t = 10\mu s$, $t_v = 1ms$ and $c_p = 50\mu s$. Using these values in equation (1) for the two spirals problem we have $T_{p,s}/T_s = 7.43$. Parallel run time (47 minutes) is computed using this ratio and branching algorithm convergence in 7000 epochs versus 50000 needed by the standard back-propagation.

The neural network for the breast cancer problem is larger (10 inputs, 10 hidden units, one output) and consequently $t_t = 20\mu s$, $t_v = 2ms$ and $c_p = 100\mu s$ while t_c remains the same as in the two spirals problem since the same number of branches is generated. This gives $T_{p,s}/T_s = 4.84$ and a parallel run time of 63 minutes which is less than run time for the standard back-propagation convergence. Here the parallel branching algorithm is faster then the standard one since it converges in 5000 epochs versus 25000 needed by the back-propagation.

It is easy to see that the parallel branching algorithm is more appropriate for larger than for smaller neural networks. This follows from equation (1) since t_t, t_v and c_p grow at the same rate with the size of the problem while t_c remains constant. ¿From equation (1) it is also easy to see that the branching algorithm is more appropriate for highly parallel rather than distributed systems. In the case of a distributed system the transmission time t_t is dominant and much larger then t_c and t_v which together with c_p can be derived exactly running a test program on a network of workstations.

Figures 6 and 7 presents learning speed in terms of epochs an d time for the spirals problem using the back-propagation and the short term branching algorithm. Figure 6 shows that a better learning performance is attained in a smaller number of epochs using the branching algorithm. Figure 7 shows the superiority of the standard algorithm versus sequential implementation of the branching algorithm in terms of computing time. Figure 7 also shows a trajectory resulting

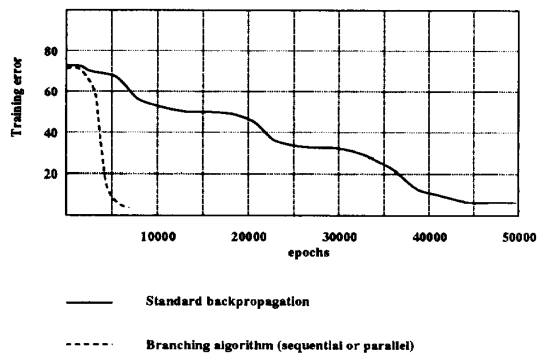

Figure 6: Number of training epochs comparison

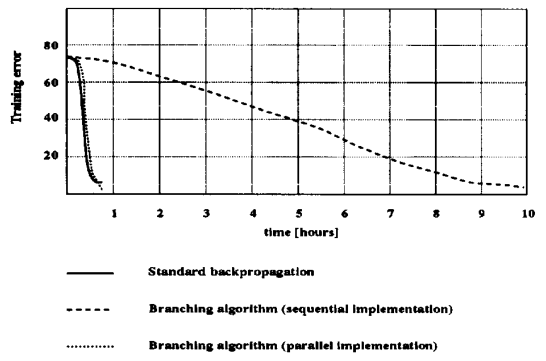

Figure 7: Computing time comparison

from a parallel implementation. This trajectory was computed based on the equation (1) derived earlier. It indicates that branching algorithms are highly efficient when implemented on a parallel machine.

Simulation results indicate that the long term branching algorithm requires considerably more epochs for convergence comparing to the short term branching algorithm. In fact we stopped the training before the error function became stationary and at that time the testing showed a small improvement in generalization (1%). This is consistent with theoretical observations from Section 2.2 that the long term branching algorithm requires a much larger number of update steps to achieve a useful variety of the learning trajectories.

5 Summary

We have proposed and analyzed a bounded depth branching approach to neural network learning. The experimental results show improved generalization compared to the standard back-propagation learning algorithm. This branching process is very expensive if implemented on a sequential machine as it requires a large amount of hardware resources. The proposed parallel implementation dramatically improves the algorithm efficiency to the level that computing time is no longer a critical factor in achieving improved generalization.

The idea of bounded depth branching is applicable to other neural network learning algorithms. Extension from the back-propagation to improvement of any other gradient descent learning algorithm is straightforward. Further research topics include application of this branching idea to non-gradient descent methods such as constructive learning algorithms.

Acknowledgement

The authors would like to thank Justin Fletcher for programming help and for useful comments on the first draft of this paper.

References

[1] S. E. Fahlman. Faster-learning variations on back-propagation: An empirical study. In D. Touretzky, editor, *Proceedings, 1988 Connectionist Models Summer School*, volume 1, pages 38–51. Morgan Kaufmann, San Mateo, 1988.

[2] Intel Supercomputer Systems Division, Beaverton, OR. *Touchstone Delta System User's Guide*, October 1991.

[3] K. J. Lang and M. J. Withbrock. Learning to tell two spirals apart. In D. Touretzky, editor, *Proceedings, 1988 Connectionist Models Summer School*, volume 1, pages 52–59. Morgan Kaufmann, San Mateo, 1988.

[4] F. T. Leighton, editor. *Introduction to Parallel Algorithms and Architectures.* Morgan-Kaufmann, San Mateo, CA 94403, 1992.

[5] S. M. Weiss et al. An empirical comparison of pattern recognition, neural nets, and machine learning classification methods. *Machine Learning*, 3(6):781–787, November 1990.

[6] L. F. Wessels and E. Barnard. Avoiding false local minima by proper initialization of connections. *IEEE Transactions on Neural Networks*, 3(6):899–905, November 1992.