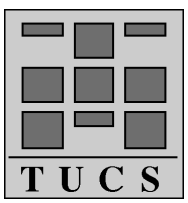
A Retraining Improvement of Feedforward Neural Networks

Iulian Nastac

Turku Centre for Computer Science,
Institute for Advanced Management Systems Research,
Lemminkäisenkatu 14 B, FIN-20520 Turku, Finland
e-mail: Iulian.Nastac@abo.fi

Razvan Matei

Nokia Oy, Takomotie 1, 00380, Helsinki, Finland
e-mail: Razvan.Matei@nokia.com



Turku Centre for Computer Science
TUCS Technical Report No 504
January 2003
ISBN 952-12-1110-5
ISSN 1239-1891

Abstract

The artificial neural networks (ANNs) ability to extract significant information from an initial set of data allows both an interpolation in the a priori defined points, as well as an extrapolation outside the range bordered by the extreme points of the training set. The main purpose of this paper is to establish how a viable ANN structure at a previous moment of time could be re-trained in an efficient manner in order to support modifications of the input-output function. To be able to fulfill our goal, we use an anterior memory, scaled with a certain convenient value. The evaluation of the computing effort involved in the retraining of an ANN shows us that a good choice for the scaling factor can substantially reduce the number of training cycles independent of the learning methods.

Keywords: retraining procedure, weights, scaling factor, number of training cycles, dissimilarity error

Data Mining and Knowledge Management Laboratory

1. Introduction

It is well known that the training process of an artificial neural network (ANN) requires a large number of processing cycles (Hassoun, 1995), which can occasionally reach and even outnumber hundreds of thousands. The necessary time for the learning process (Hagan et al. 1996, Zhang et al. 2000) is directly proportional to the complexity of the application implemented by ANN. At a first glance, this would imply that a small change in the initial project (e.g. some re-evaluation of the ANN performance at time point when a certain amount of experience has been accumulated) might require the repetition of the entire training phase, including all the shortcomings deriving from that, i.e. a long processing time, the possible occurrence of an undesired local minimum of the performance function, etc. Therefore, in this paper we propose a new approach to overcome this disadvantage. The main purpose is to establish an efficient method to re-train a viable ANN structure at a certain moment of time such that it will support variations of the initial input-output function.

To solve this problem we use an anterior memory "scaled" with a certain convenient value. A remembering process of the old knowledge achieved in the first learning phase is used as reference. This way, we are able to evaluate the computing effort involved in the retraining of an ANN.

The structure of this paper is as follows. Section 2 presents the problem concerning the extraction of the useful information from an ANN in order to reduce the re-designing effort. In next section we introduce the retraining procedure and explain the working strategy. The main features of our experimental results are given in Section 4, where we discuss specific aspects. Our conclusions are formulated in the final section of the paper.

2. Background

Usually, feedforward ANNs are well suited to implement different kinds of input-output functions. The number of hidden layers and the number of the neurons for each layer are dependent on the complexity of these functions. After we have established a primary architecture during the first training phase of an ANN, the weights are initialized to small uniformly distributed values (e.g. in the interval (0,0.1)). The values have to be chosen small because during the training process the weights strive statistically to grow and if the learning process is a long one, the excessive growth of the weights may paralyze the learning process (Zeidenberg, 1991). It then follows that the learning algorithm will influence the network insignificantly. The explanation resides in the fact that larger weights values will force large outputs balanced sum concluding towards asymptotic values of the transfer function (e.g. sigmoid function). Experiments proved that, besides the small values of the initial weights, choosing a small learning rate would increase the chance to avoid the paralysis state (Nastac, 2000). The value of the learning rate is relative and one cannot assume that a certain small value of the learning rate for one particular experiment is small enough for another.

Therefore, after having established the architecture and an initialization process, the ANN will be in some arbitrary state of the continuous or discrete weights space. As the information theory states (Girolami, 1999), we can assign a value of uncertainty to the network. The information extracted is growingly important as the uncertainty of the state of a system is higher. The uncertainty rate of a physical system results not only from the number of the possible states that the system could reach during its evolution, but also from the probability value associated with these states.

Let us consider an arbitrary weight w_j in the network structure as a random discrete variable that could take any of the values w_j^i (i=1..n) with the probability $p(w_j^i)$.We can define the entropy of this particular weight as follows:

$$H(w_j) = -\sum_{i=1}^{n} p(w_j^i) \lg(p(w_j^i))$$

If w_j is a continuous random variable with the $f(w_j)$ probability density, the weight entropy can be rewritten as :

$$H(w_j) = -\int_{-\infty}^{+\infty} f(w_j) \lg(f(w_j)\Delta x) dw_j$$

Here, $f(w_j)\Delta x$ represents the probability to fall into one of the Δx segments of the abscissa of the probability density function $f(w_j)$.

The amount of information necessary to find out the final state of the weight is equal to its entropy:

$$I_{w_j} = H(w_j)$$

For practical reasons, we deal with the entropies of some mutually conditioned random variables (weights). The general entropy of the neural networks could then be written as:

$$H_{NN} = H(w_1, w_2, ..., w_N) = H(w_1) + H(w_2 \mid w_1) + ... + H(w_N \mid w_1, w_2, ..., w_{N-1})$$

Computing the ANN entropy formula efficiently is a hard task, if not impossible, when some highly dimensionally structures are involved.

The theory (Girolami, 1999) tells us that the weights initialization with random distributed values leads to the idea that the weights space entropy has a maximum value before the effective learning process begins. By choosing some uniform probability distribution we could increase the chances to find the optimum solution through some learning technique. On the other hand, in this case, searching is not efficient as far as the number of required training cycles is concerned. This is an unavoidable phenomenon when one first tries to design an ANN.

Let us suppose that after a while the input-output function of the ANN has to be reconsidered. If we restart from scratch, we will train the network exactly in the same way as in the first phase. Therefore, the computing effort remains the same. As an alternative, we can try to use some of the previous knowledge acquired in the first phase, while designing the network, in order to reduce the re-designing effort. The question is how to extract the useful information from the first project and use it in an efficient way in the retraining phase of the network.

If we define the initial system as X and the re-designed system as Y, then the quantity of the information over the system Y obtained through the system X is:

$$I_{X \to Y} = H(Y) - H(Y \mid X)$$

This result can be seen as diminishing the entropy of the system Y. Although theoretically clear, obtaining $H(Y|X)$ in practice represents a great challenge.

Since any ANN is a highly parallel system, the computing engine of the system cannot be found by using a sequential algorithm. To our knowledge, so far, the network itself is the only one that can administer its own information from its weights. Interpreting the network weights in another way was considered irrelevant until now.

3. Retraining Procedure

In this section, we will describe our practical information extracting mechanism directly from the weights of a reference ANN which was already trained and it is perfectly functional at the present time. We use this information to train a structurally identical ANN, or even the same network that has some variations of the global transformation input-output function.

Our proposed procedure reduces the reference network weights by a *scaling factor* γ ($0<\gamma<1$). These reduced weights are used as initial weights for the training sequence of the second network. At the end we compare the network convergence speed (i.e., the number of cycles required until an imposed error is reached) obtained in both cases. Note that, before the training phase, the reference network had its weights initialized to random uniformly distributed values. In case we systematically achieve some smaller convergence speed for the second network, the global training time will then decrease as a consequence, hence our mechanism proves beneficial.

The retraining mechanism will be analyzed in the following cases: *same function with same training set* (identical with the one used in the first training phase, which preceded the scale reduction process), *same function with different training set*, and *different function with a new training set*.

In order to properly evaluate the proposed procedure, we will test its effects on four training models: BP (back-propagation), momentum, ALR (Adaptive Learning Rate) and the ALR-momentum combination. This way, we try to emphasize that our procedure has similar results regardless of the learning algorithm used.

To ease the testing work, we establish some preliminary conditions without restraining the generality of the results. Thus, the tested feedforward neural networks start running with one input - one output and continue with increased number of inputs. If the ANN has more outputs, then each of them leads to a hyper-surface in a s+1 dimensional space, where s represents *the number of the inputs*. As the training techniques are fairly similar for the ANNs with an arbitrary number of inputs and outputs, it implies that the achieved results will be true even in the general case of a highly dimensional ANN.

We limited the training set to a *relative* uniform distribution of data. However according to (Hagan et al. 1996, Setiono et al. 1997), we tried to pick more data from the case of continuous functions where the derivative had higher values.

All the above conditions do not restrain the generality of the method, instead they try to comply with the limited computing resources available and also help the understanding by providing intuitive results.

In order to find the necessary training cycles and the learning error, we have considered the following pre-defined two-dimensional functions:

$$f:[b,c]\to[d,e]$$

or multi-dimensional functions:

$$f:[b_1,c_1]\times...\times[b_s,c_s]\to[d,e]$$

After training, the function ϑ generated by the ANN will be closer to the desired function *f*.

Next, we propose the, so called, *dissimilarity error* E_d as a means to evaluate the resemblance between the graphical representations of *f* and ϑ (or *f* and another function *g*). This error is defined as:

$$E_d(f,\vartheta,v)=\frac{\sum_{j=1}^{v}|f(x_j)-\vartheta(x_j)|}{v}$$

In the previous equation, Σ represents the sum of the modules of the differences between these two functions measured in v equidistant points.

For network training purposes, we chose a finite number *n* of pairs $(x_i, f(x_i))$, $i=1..n$, from the function's graph. Although we could have taken equidistant points on the abscissa, we can also accept some slight variations, to be closer to the real situations where we do not know the exact interval of the values for the function *f*. In case we know the function *f* entirely, we really do not need an ANN to simulate the function but only a memory instead, with the whole set of values that provide us with *f(x),* for any *x*. If, on the contrary we only know partially the function *f*, then the ANN is the right choice because of its interpolation and extrapolation capabilities.

The approximation error E_a used in the training sequence is given by the following formula:

$$E_a=\frac{\sum_{j=1}^{n}(f(x_j)-\vartheta(x_j))^2}{n}$$

where *n* is the number of input-output learning pairs.

After selecting the pairs $(x, f(x))$, we can effectively train the network with its weights previously initialized to small random values. The learning phase stops when the approximation error E_a falls below a fixed error limit E_l. After that, we note the number of training cycles such that the condition $E_a \le E_l$ is satisfied.

Our main intent is to reduce the number of the learning cycles. The retraining procedure needs the completion of two major steps. Firstly, we have to get the set of weights by inheriting them from the reference ANN, and secondly we have to reduce these weights by a scale factor γ (e.g. in range (0.1,0.9)). The newly obtained weights represent the base for the new learning process. In order to decide the optimum scale factor and the consequences of the scale reduction procedure itself, the retraining phase should be performed for more than one value of γ. Then, for different training models, we will observe how the number of training cycles varies with γ. This factor will finally reach an optimum value γ_{opt} , the one that causes a minimum number of training cycles.

Once we have established the retraining principle, the only thing that remains is to test its practical utility. Therefore we build an efficient benchmark algorithm of the retraining procedure as follows:

1) Decide the initial function f and training set
2) Choose the network architecture
3) Select the training procedure
4) Initialize the network weights with small uniformly distributed values

5) Train and hold the *training cycles number V*
6) Assign L values to scale factor γ
7) Reduce by each γ_i the weights obtained at step 5
8) Repeat learning procedure for each set of weights and memorize $V(\gamma_i)$
9) Repeat step 8 for another training set of function f or for the case of function g with a similar graph

As an observation, we mention that the set of values for the functions f, g and ϑ could be easily extended to other larger interval of values. The neural function ϑ is reshaping itself after each training cycle. We initially chose networks which implement two-dimensional functions just to simplify the simulation and to lower the burden of interpreting the results. The generalization is straightforward, due to the fact that any effective algorithm remains the same for an arbitrary number of inputs and/or outputs.

4. Results and Discussion

In this section we will illustrate our results, the working mode and the influence of the retraining sequence over an ANN, by considering different families of functions.

First, for *one input - one output* case, choosing the arbitrary initial function f (where f:[-1,1]→[-1,1]), we used the graph showed in Fig. 1.

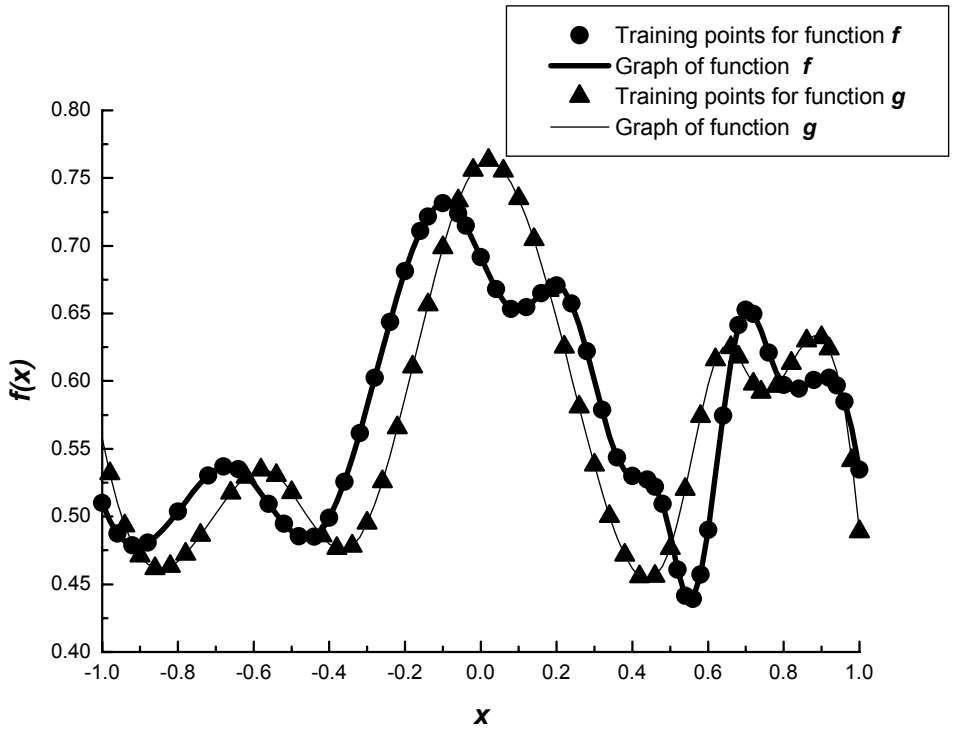


Fig.1. The graphs of functions f and g

This function was copied by a neural network with one input – one output following the training set from 54 points that could be seen on the graph. In addition, we used another

function g that had a shape near to function f (see Fig. 1). To verify the robustness of the algorithm we chose a number of 50 pairs for function g by comparison with 54 in the case of function f.

The dissimilarity error between f and g, measured for 100 points was $E_d(f,g,100)=0.045$. This means that the approximate area between these two curves has the value of $E_d \cdot (c-b)=0.045 \cdot 2=0.09$. We can easily observe that the functions are very similar. On the same graph f, we selected a different training set built of 54 points $((x'_i f(x'_i))$, $i=1..54$).

The used network architecture has two hidden layers, each of them with 5 neural cells. Selecting the BP model, we trained the network starting from randomly and uniformly distributed weights. We observed that 2228 learning cycles were necessary. Then, the training process was resumed for a number of $L=9$ values of the scaling factor γ ($\gamma_1=0.1$, $\gamma_2=0.2$, … $\gamma_9=0.9$), in order to learn the function f using the training set $(x_i, f(x_i))$, $i=1..54$. The results can be seen in Fig. 2, which graphically represents the number of training cycles for each retraining with γ_i . We marked with a horizontal line the number of cycles necessary in the first training process, before scaling.

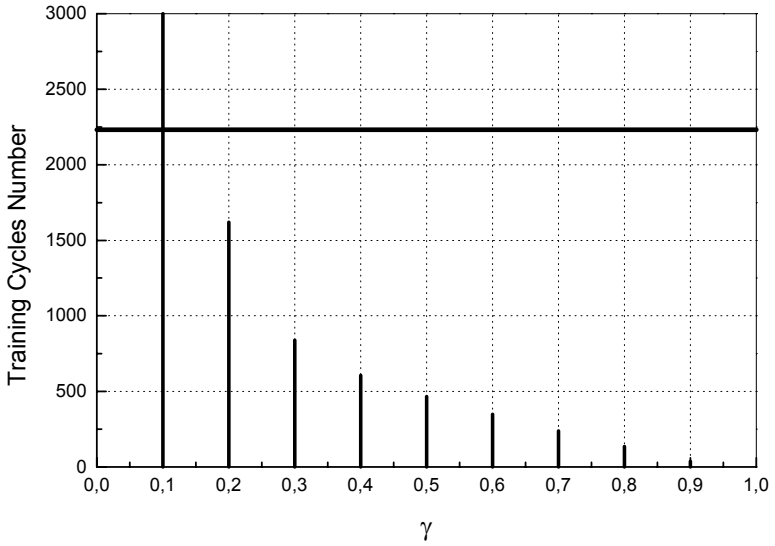


Fig. 2. Retraining procedure through BP method for the same function f with the same points

By using the training set $(x'_i, f(x'_i))$, $i=1..54$ in order to learn the function f, we obtain a similar graph with that of Fig. 2. The number of retraining cycles decreases progressively with the increase of γ. We observed that the variations are very small when using different points for the retraining process.

In order to see whether the obtainable data is useful to learn a new function like g, we repeated the retraining procedure starting from the initial weights, scaled using the training set (50 points pairs) of the function g. The resulted graph is presented in Fig.3. We can see some differences when we compare the histograms from Fig. 2 and Fig. 3. Even if initially the number of the cycles strives to decrease as γ increases, from value 0.7 we have a constant increase on the graph. For $\gamma \geq 0.2$, we stay below the threshold of the initial training cycles number. For higher values of the γ (over 0.8), we could be

suspicious because an over-learning phenomenon is likely to appear in Fig. 3. In this case the network has difficulties in making generalizations for increasing values of γ.

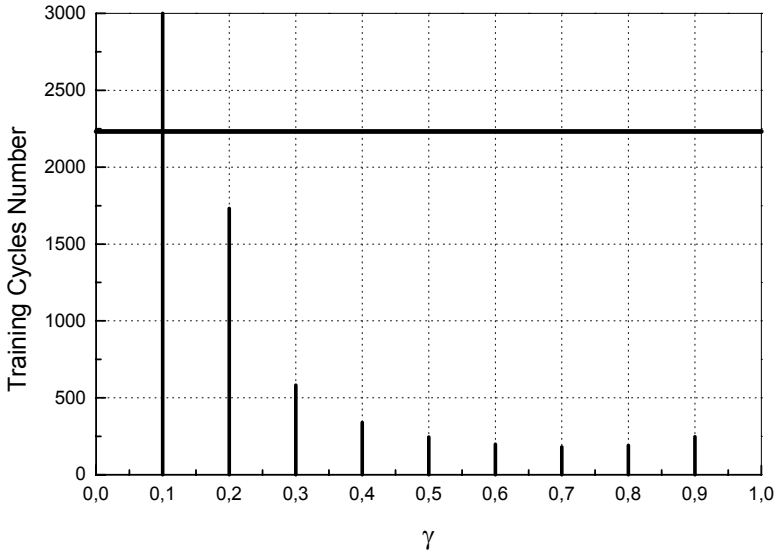


Fig. 3. Retraining procedure through BP method for the different function *g*

These kinds of graphs will be very useful in our final evaluation. Obviously we are interested in those values of γ, for which the number of retraining cycles is below the horizontal line because in those cases the training procedure behaves much better.

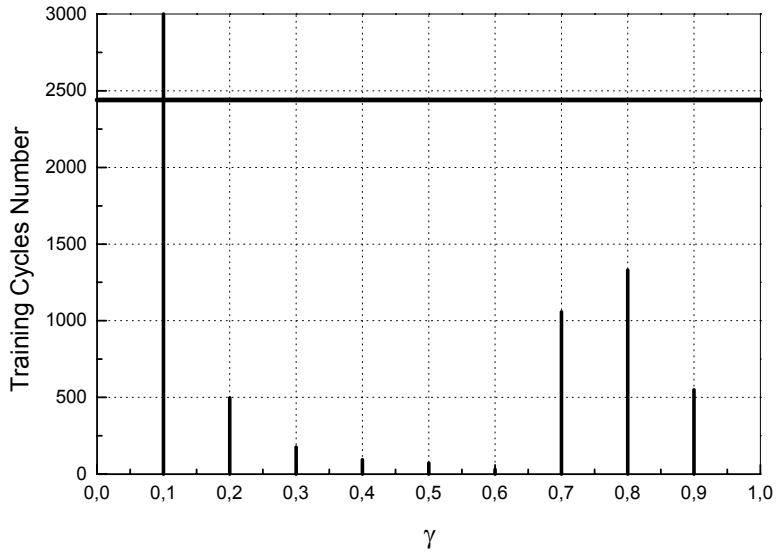


Fig. 4. Retraining procedure through momentum method for the same function *f* with the same points

Under the same conditions choosing any one from among the momentum, ALR or ALR-momentum training methods we noticed a general decreasing behavior of the

training cycles number in accordance with the increase of the scaling factor γ, with the significance that in some situations (see Fig. 4) we could notice an important increasing behavior once $\gamma \geq 0.7$.

As a remark, regardless of the training procedure, using a neural network with a single hidden layer of 10 neural cells the observed values during the simulations were pretty much alike (maybe slightly increased) in comparison with previous presented situations for two hidden layers neural networks.

Maintaining the same initial conditions, we tested the effect of the hidden layers number. Hence, we changed the architecture from the one with two hidden layers, summing 10 neural cells, to the other with three hidden layers (each layer with 5 neural cells). Starting again the scaling procedure for each of the used training methods, we noticed that the over-learning phenomenon vanished in the BP case for different function, but appears in the ALR case. For the momentum and ALR-momentum models there are no major changes, concluding that in general, the results of the retraining procedure are independent from the number of hidden layers.

As a result of analyzing the data collected until now, regardless of the learning algorithm used, we get that varying the scale value, the number of retraining cycles starts from a higher value than the reference cycles number, and then it progressively decreases, in accordance with the increases of γ, much below the reference value, especially for $\gamma \geq 0.3$. Sometimes, for the values of γ higher than 0.6, we have significant jumps of the learning cycles number that are associated with the network paralysis or over-learning phenomenon.

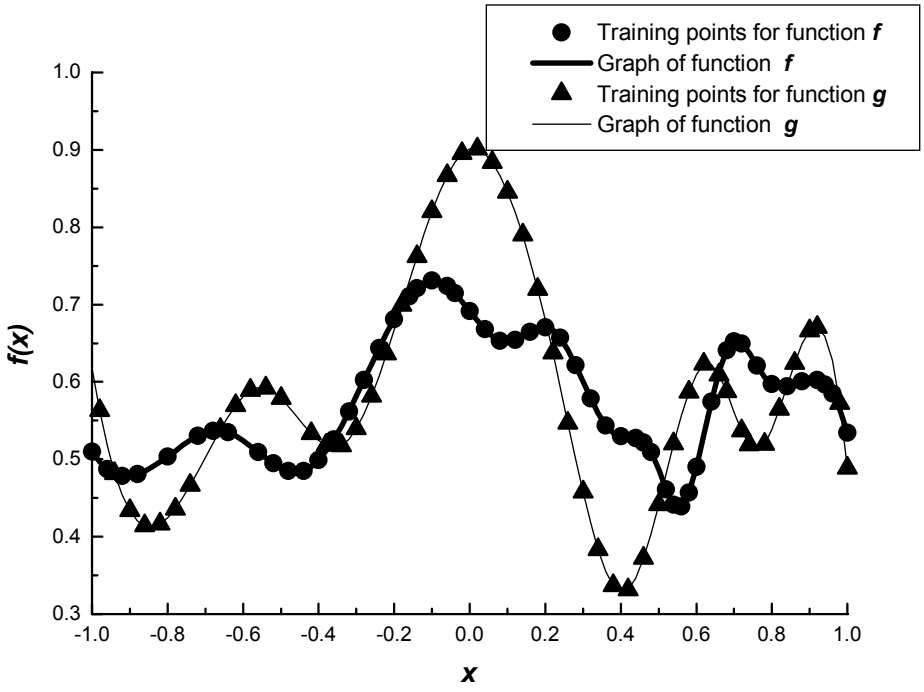


Fig. 5. Significant differences between the graphs of the functions *f* and *g*

However, our method may not always provide spectacular results concerning the decrease of the training cycle number. Thus, it is possible that the retraining procedure is not efficient at limit situations. For example, in Fig. 5 when the graph of *f* is quite

different from the graph of g, the retraining procedure does not lead to smaller values of the cycles number (see Fig. 6, ALR case, where $E_d(f,g,100)=0.1$). The graph of training cycles number descends below the initial threshold just for $\gamma = 0.4$ and 0.5. If the dissimilarity error between these two functions grew more, then we would have no reason to apply retraining method because we cannot achieve a reduction of the training cycle number. This leads to the conclusion that the scale reduction procedure must be carefully applied when the goal function g is too much different from f, previously known from an earlier training.

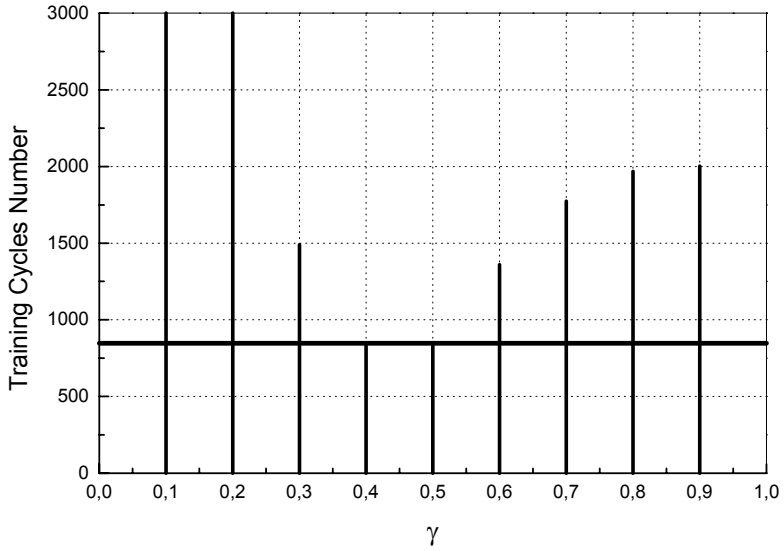


Fig. 6. Retraining procedure for the function g in the case of significant difference between f and g

For *two inputs – one output* case, we used a function $f : [0,4]\times[0,4] \rightarrow [0,5]$ (see Fig.7) given by the following analytic formula:

$$f(x, y) = \frac{\sin(x^2)\cos y}{2} + \frac{xy}{4} + 0.5$$

Then we established another function g (where $g : [0,4]\times[0,4] \rightarrow [0,5]$, see Fig. 8):

$$g(x, y) = \frac{\sin(x(x+0.1))\cos(y-0.2)}{2} + \frac{xy}{4} + 0.5\,,$$

which is not much different than f.

The dissimilarity error between f and g, measured in v=1000 points, is $E_d(f,g,v)$=0.08. The volume between these two surfaces has the value of $E_d \cdot (c_1-b_1)\cdot(c_2-b_2)= 0.08\cdot4\cdot4=1.28$. We can easily observe that these shapes are almost similar.

Two training sets of 400 points each one were collected from the surfaces of Fig. 8 and Fig. 9. We used *two inputs - one output* network architecture with two hidden layers (10 neural cells each one).

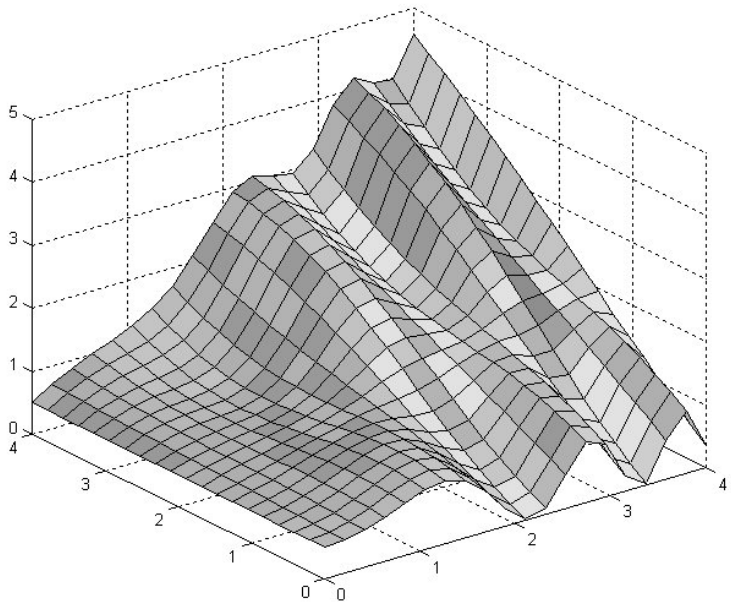


Fig. 7. The graph of the function *f*

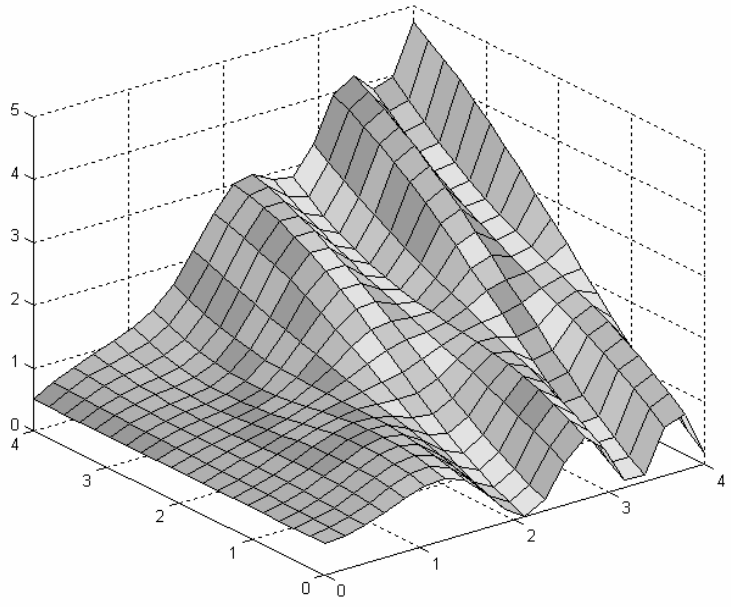


Fig. 8. The graph of the function *g*

We compared again the training cycles number, obtained after each scale reduction, with the initial number of training cycles, which was necessary the first time in order to learn the shape of function *f*. When we started from uniformly distributed random

weights values, we represented that through a horizontal line in an already known manner. We repeated the working mode used in the *one input - one output* section for all four training algorithms (BP, momentum, ALR, ALR-momentum) without considering the case of the *same function with different training set*. Studying the obtained results, we did not observe noticeable differences in behavior more than in the *one input - one output case*. Training with the BP method led to a powerful over-learning phenomenon that appeared for $\gamma = 0.9$ (see Fig. 9).

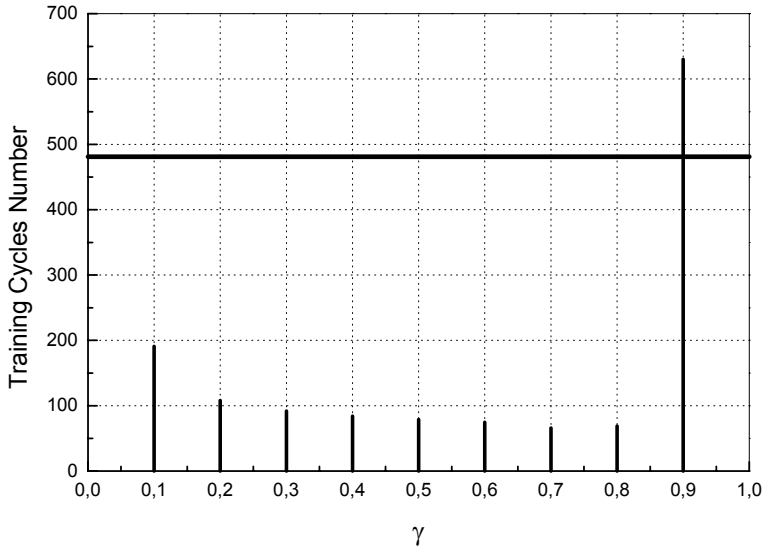


Fig. 9. Retraining procedure through BP method for the different function *g*

In this situation, the number of the retraining cycles outnumbered clearly the initial level. The same behavior, less spectacular, could be observed in the ALR-momentum case, for $\gamma = 0.7$ (see Fig. 10).

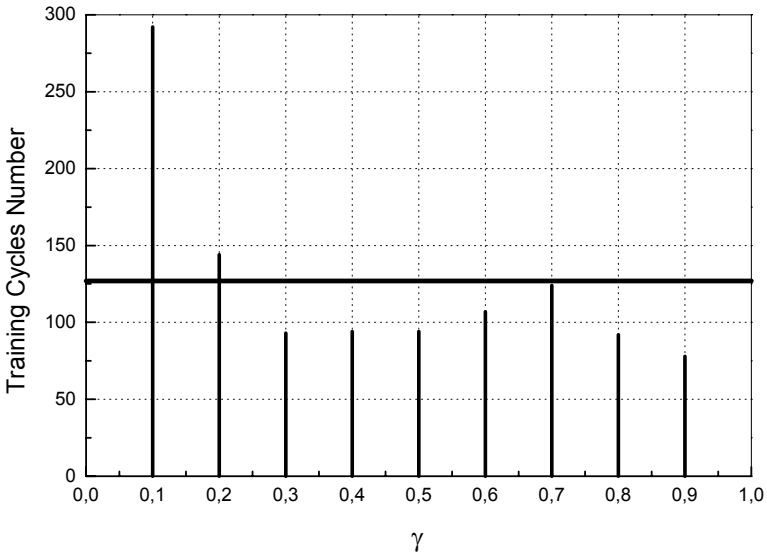


Fig. 10. Retraining procedure (ALR-momentum) for the different function *g*

The shapes of the graphs given by the neural network functions after the training process are very close to the desired goal. For example, training through momentum method (scale reduction by $\gamma = 0.6$), in the case of the function g, produced the graph from Fig.11, which is astonishingly similar to the one from Fig. 8.

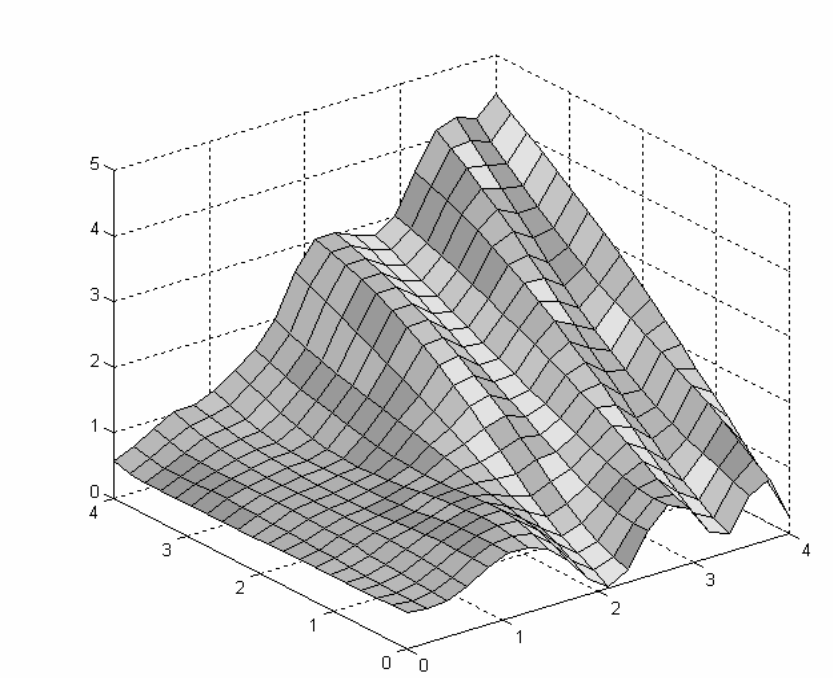


Fig. 11. The graph of the neural network function ϑ (case $\gamma = 0.6$ for momentum method) $E_d(g, \vartheta, v)$=0.063

The dissimilarity error, computed in $v = 10000$ points between the graph extracted from the neural network and the graph of the function g, is $E_d(g, \vartheta, v)$=0.063. Similar forms of the neural function could be achieved for all cases where the number of the training cycles decreased below the initial level (given by the training based on the uniformly distributed weights) and especially for $\gamma \in [0.3, 0.7]$ where minimum values of the E_d were encountered.

We performed similar experiments for other neural networks with 3 and 4 inputs. We could not notice different behaviors as far as the evolution of the training cycles number varying with the parameter γ is concerned.

5. Conclusions

In this paper, we have proposed a procedure for retraining those ANNs, which require modifications of their input-output functions. We described the information extracting mechanism directly from the weights of a reference ANN that was already functional. These weights were reduced by a scale factor γ, and handled as initial

weights for the new training sequence. Using our procedure, we obtained a significant decrease in the number of the training cycles compared to the classical way.

Based on the simulations performed for multiple combinations of the input parameters, we conclude that the optimal γ has its value around 0.5. Increasing this coefficient is not justified by the over-learning possibility and the implicit paralysis of the neural networks. In addition, values below 0.5 lead to a behavior very similar to a lack of the memory that remains by scaling.

We noticed that the phenomena in the ANN behaviour are almost the same when the retraining procedure is applied, regardless of the learning method used. Even if some of the techniques used are more efficient than others, applying the scaling method leads to a somehow similar ratio with respect to the decrease of the training cycles. This ratio depends on the analyzed case, more precisely on the neural architecture, functions dissimilarity, imposed error limit, etc.

The results and the graphs were selected in a non-preferential manner from more than 200 retraining simulation sessions. In 32% of the analyzed cases, for $\gamma \geq 0.7$, we noticed that the number of the training cycles has an ascending trend, and this was associated with the over-learning phenomenon. The aforementioned percentage kept itself relatively independent in almost all the situations given by the parameter modifications, i.e. inputs number, layers number, cells number of each hidden layer, etc.

The performed simulations lead to similar behavior, independently of the inputs number of the tested models. The similarity between the training methods for networks with an arbitrary number of inputs and/or outputs implies the fact that the results are also valid for the case of a highly dimensional space. This reason allows us to generalize the conclusions regarding the values of γ to networks with any dimension.

The main advantage of ANNs is their fastest computing speed. Unfortunately they need too much time for training process therefore we showed a way to decrease this time. Research is being conducted to implement the retraining procedure for ANNs that perform prediction tasks.

Acknowledgements

We gratefully acknowledge the assistance of the Turku Centre for Computer Science and the Institute for Advanced Management Systems Research, Finland, which supported the final part of this work. Special appreciation is expressed to Prof. Barbro Back for useful discussion and many helpful suggestions on this and earlier versions of this paper.

References

Girolami, M., 1999. *Self-Organising Neural Networks*. Springer-Verlag. London.

Hagan, M.T., Demuth, H.B. and Beale, M., 1996. *Neural Networks Design*. MA: PWS Publishing. Boston.

Hassoun, M. H., 1995. *Fundamentals of Artificial Neural Network*. MA: MIT Press. Cambridge.

Nastac, I., 2000. *Contributions in Technical Systems Quality Modelling through the Artificial Intelligence Methods*. Ph.D. dissertation, Polytechnic University of Bucharest. Romania.

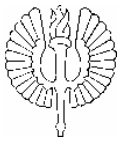
Setiono, R. and Liu, H., 1997. Neural network feature selector. In *IEEE Trans. Neural Networks*, Vol. 8, No. 3, 654-662.

Zeidenberg, M., 1991. *Neural Networks in Artificial Intelligence*. Ellis Horwood Ltd. London.

Zhang, Y., Peng, P. Y. and Jiang, Z. P., 2000. Stable neural controller design for unknown nonlinear systems using backstepping. In *IEEE Trans. Neural Networks*. Vol. 11, No. 6, 1347-1360.

Turku Centre for Computer Science
Lemminkäisenkatu 14
FIN-20520 Turku
Finland

http://www.tucs.fi/



University of Turku
• Department of Information Technology
• Department of Mathematics

Åbo Akademi University
• Department of Computer Science
• Institute for Advanced Management Systems Research

Turku School of Economics and Business Administration
• Institute of Information Systems Science