

Liquid flow time series prediction using feed-forward neural networks and *Rprop* learning algorithm

Laurențiu Leuștean

National Institute for Research and Development in Informatics,
8-10 Averescu Avenue,
71316 Bucharest,
ROMANIA
E-mail: leo@u3.ici.ro

Abstract: This paper presents an application of feed-forward neural networks and *RPROP* algorithm to forecast a time series consisting of the average monthly liquid flow measured at Broșteni hydrometric station from the Motru River. The paper represents the Romanian contribution for one of the Technical Reports of INTAS Project no. 397: “Data Mining Technologies and Image processing: Theory and Applications”, Task 4: “The prognosis of harvest and the base of the weather- and geo-monitoring of a region”.

Keywords: neural networks, liquid flow, time series prediction, *RPROP* learning algorithm.

Introduction

The forecasting of different natural phenomena related to river flow is one of the priorities of hydrology in this moment. In order to obtain an adequate prediction it is necessary to observe the phenomena a long period of time, usually more than 30 years.

The liquid flows display great fluctuations that can determine the flooding of agriculture land or, conversely, can diminish thus badly affecting the water supply. Accordingly, knowing the evolution of flows is crucial for predicting the agriculture output on irrigating or flooding land

Neural networks are computational frameworks consisting of massively connected simple processing units (neurons). These units have an analog to the neurons in the human brain. Because of the highly connected structure, neural networks exhibit some desirable features, such as high speed via parallel computing, resistance to hardware failure, robustness in handling different

types of data, the property of being able to process noisy or incomplete information, learning and adaptation

One of the most popular neural net paradigms is the feed-forward neural network. In a feed-forward neural network, the neurons are usually arranged in layers. One of the most promising algorithms for feed-forward neural networks is *Resilient Backpropagation (RPROP)*, introduced by M. Riedmiller in 1993.

This paper presents an application of feed-forward neural networks and *RPROP* algorithm to forecast a time series consisting of the average monthly liquid flow measured at Broșteni hydrometric station from the Motru River. The data are from the period 1950-1994 [12].

The hydrological basin of the Motru River is situated in the Southwest of Romania and it has an area of 1874 km². The Motru river is 138,8 km long [9, 11].

The average monthly liquid flow represents the water volume that flows

through the section of a river in a month, reported at the unity of time and it expresses in m^3/sec .

We must mention that in hydrology, for the bigger rivers, because of the inertness of the instruments used for measuring the speed of the water there is the probability that the measured liquid flow to be bigger or smaller than the real one. That's why it was established that a tolerance of $\pm 10\%$ is in the normal limits of measurement. It follows that a forecasting is considered correct if the value of the deviation is at most $\pm 10\%$ from the measured value.

The classical predictions of these time series are realized using statistical methods, namely Pearson 3 methods. Neural networks can be a promising alternative to classical prognosis based on statistical methods.

In the first section of the paper we present basic concepts about feed-forward neural networks. In the second section we describe the learning algorithm *RPROP* (*Resilient Backpropagation*). In the next section we present the experimental model. There are analyzed the parameters used by the algorithm and there are presented the results obtained, as the values of the parameters corresponding to these results. In the last section of the paper we present some conclusions.

1. Feed-forward neural networks

1.1 Basic concepts

A neural network is a parallel distributed information processing structure consisting of processing elements (neurons) interconnected via unidirectional signal channels called connections. Each processing element has a single output connection that

branches into as many collateral connections as desired; each carries the same signal - the processing element output signal. The processing element output signal can be of any mathematical type desired. The information processing that goes on within each processing element can be defined arbitrarily with the restriction that it must be completely local; that is, it must depend only on the current values of the input signals arriving at the processing element via incoming connections and on values stored in the processing element's local memory.

Neural networks develop information processing capabilities by learning for examples. Learning techniques can be roughly divided into two categories:

1. supervised learning;
2. unsupervised learning.

Supervised learning requires a set of examples for which the desired network response is known. The learning process consists then in adapting the network in a way that it will produce the correct response for the set of examples. The resulting network should then be able to generalize (give a good response) when presented with cases not found in the set of examples.

In unsupervised learning the neural network is autonomous; it processes the data it is presented with, finds out about some of the properties of the data set and learns to reflect these properties in its output. What exactly these properties are, that network can learn to recognize, depends on the particular network model and learning method.

One of the most popular neural net paradigms is the feed-forward neural network. In a feed-forward neural network, the neurons are usually arranged in layers [6]. A feed-forward

neural net is denoted as $N_I \times N_1 \times \dots \times N_i \times \dots \times N_L \times N_O$, where:

- N_I represent the number of input units;
- L represent the number of hidden layers;
- N_i represent the number of units from the hidden layer i , $i = \overline{1, L}$;
- N_O represent the number of output units.

By convention, the input layer does not count, since the input units are not processing units, they simply pass on the input vector x . Units from the hidden layers and output layer are processing units. Figure 1 gives a typical fully connected 2-layer feed-forward network with a $3 \times 4 \times 3$ structure.

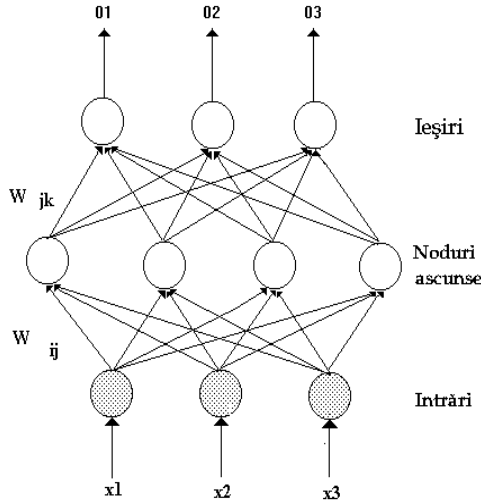


Figure 1: A $3 \times 4 \times 3$ feed-forward neural network.

Each processing unit has an activation function that is commonly chosen to be the sigmoid function:

$$f(x) = \frac{1}{1 + e^{-x}}.$$

The net input to a processing unit j is given by:

$$net_j = \sum_i w_{ij} x_i + \theta_j,$$

where x_i 's are the outputs from the previous layer, w_{ij} is the weight (connection strength) of the link connecting unit i to unit j , and θ_j is the bias of unit j , which determines the location of the sigmoid function on the x axis.

The activation value (output) of unit j is given by:

$$a_j = f(net_j) = \frac{1}{1 + e^{-net_j}}.$$

We remark that:

$$\begin{aligned} f'(net_j) &= \frac{e^{-net_j}}{(1 + e^{-net_j})^2} \\ &= f(net_j)(1 - f(net_j)) \\ &= a_j(1 - a_j) \end{aligned}$$

The objective of different supervised learning algorithms is the iterative optimization of a so called *error function* representing a measure of the performance of the network. This error function is defined as the mean square sum of differences between the values of the output units of the network and the desired target values, calculated for the whole pattern set. The error for a pattern p is given by

$$E_p = \sum_{j=1}^{N_O} (d_{pj} - a_{pj})^2,$$

where d_{pj} and a_{pj} are the target and the actual response value of output neuron j corresponding to the pattern p .

The total error is

$$E = \sum_{p=1}^P \frac{1}{2} E_p = \frac{1}{2} \sum_{p=1}^P \sum_{j=1}^{N_O} (d_{pj} - a_{pj})^2,$$

where P is the number of the training patterns.

During the training process a set of pattern examples is used, each example consisting of a pair with the input and corresponding target output. The patterns are presented to the network

sequentially, in an iterative manner, the appropriate weight corrections being performed during the process to adapt the network to the desired behavior. This iterating continues until the connection weight values allow the network to perform the required mapping. Each presentation of the whole pattern set is named an *epoch*.

1.2 The Backpropagation algorithm

The Backpropagation algorithm is the most popular supervised learning algorithm for feed-forward neural networks [6].

In this algorithm the minimization of the error function is carried out using a gradient-descent technique. The necessary corrections to the weights of the network for each moment t are obtained by calculating the partial derivative of the error function in relation to each weight w_{ij} . A gradient vector representing the steepest increasing direction in the weight space is thus obtained. The next step is to compute the resulting weight update. In its simplest form, the weight update is a scaled step in the opposite direction of the gradient. Hence, the weight update rule is

$$\Delta_p w_{ij}(t) = -\varepsilon \cdot \frac{\partial E_p}{\partial w_{ij}}(t),$$

where $\varepsilon \in (0,1)$ is a parameter determining the step size and is called the *learning rate*.

The partial derivative of the error for the pattern p is given by

$$\frac{\partial E_p}{\partial w_{ij}}(t) = -\delta_{pj} \cdot a_{pi},$$

where δ_{pj} is the error signal of unit j and is obtained as follows:

-if unit j is an output unit, then

$$\delta_{pj} = f'(net_{pj})(d_{pj} - a_{pj})$$

-if unit j is a hidden unit, then

$$\delta_{pj} = f'(net_{pj}) \sum_k \delta_{pk} w_{jk}.$$

Hence, the error signals δ_{pj} for the output units can be calculated using directly available values, since the error measure is based on the difference between the desired d_{pj} and actual a_{pj} values. However, that measure is not available for the hidden units. The solution is to back-propagate the δ_{pj} values layer by layer through the network.

A *momentum* term was introduced in the Backpropagation algorithm [6]. The idea consists in incorporating in the present weight update some influence of the past iteration. The weight update rule becomes

$$\Delta_p w_{ij}(t) = -\varepsilon \cdot \delta_{pj} \cdot a_{pi} + \alpha \cdot \Delta_p w_{ij}(t-1),$$

where α is the momentum term and determines the amount of influence from the previous iteration to the present one.

The momentum introduces a “damping” effect on the search procedure, thus avoiding oscillation in irregular areas of the error surface and accelerating the convergence in long flat areas. In some situation it possibly avoids the search procedure from being stopped in a local minimum, helping it to skip over those regions without performing any minimization there. In summary, it has been shown to improve the convergence of the Backpropagation algorithm, in general.

2. The algorithm *Resilient Backpropagation (RPROP)*

2.1 Description

The algorithm *Resilient Back-propagation (RPROP)* is a local adaptive learning scheme, performing supervised batch learning in feed-forward neural networks. M. Riedmiller introduced it in 1993. For a detailed discussion see [2, 3, 4, 5].

The basic principle of *RPROP* is to eliminate the harmful influence of the size of the partial derivative on the weight step. As a consequence, only the sign of the derivative is considered to indicate the direction of the weight update. To achieve this, we introduce for each weight w_{ij} its individual *update-value* $\Delta_{ij}(t)$, which solely determines the size of the weight-update.

It is introduced a second learning rule, which determines the evolution of the update-value $\Delta_{ij}(t)$. This estimation is based on the observed behavior of the partial derivative during two successive weight-steps and it is similar to the adaptation rule of the learning rate from the algorithms *Delta-Bar-Delta* [1] and *SuperSAB* [8].

$$\Delta_{ij}(t) = \begin{cases} \eta^+ \cdot \Delta_{ij}(t-1), & \text{if } \frac{\partial E}{\partial w_{ij}}(t) \cdot \frac{\partial E}{\partial w_{ij}}(t-1) > 0 \\ \eta^- \cdot \Delta_{ij}(t-1), & \text{if } \frac{\partial E}{\partial w_{ij}}(t) \cdot \frac{\partial E}{\partial w_{ij}}(t-1) < 0 \\ \Delta_{ij}(t-1), & \text{else} \end{cases}$$

where $0 < \eta^- < 1 < \eta^+$.

In words, the adaptation rule works as follows. Every time the partial derivative of the corresponding weight w_{ij} changes its sign, which indicates that the last update was too big and the algorithm has jumped over a local minimum, the update-value $\Delta_{ij}(t)$ is decreased by the factor η^- . If the

derivative retains its sign, the update-value is slightly increased in order to accelerate convergence in shallow regions.

Once the update-value for each weight is adapted, the weight-update itself follows a very simple rule: if the derivative is positive (increasing error), the weight is decreased by its update-value, if the derivative is negative, the update-value is added:

$$\Delta w_{ij}(t) = \begin{cases} -\Delta_{ij}(t), & \text{if } \frac{\partial E}{\partial w_{ij}}(t) > 0 \\ \Delta_{ij}(t), & \text{if } \frac{\partial E}{\partial w_{ij}}(t) < 0 \\ 0, & \text{else} \end{cases}$$

$$w_{ij}(t+1) = w_{ij}(t) + \Delta w_{ij}(t).$$

However, there is one exception. If the partial derivative changes sign, that is the previous step was too large and the minimum was missed, the previous weight-update is reverted:

$$\Delta w_{ij}(t) = -\Delta w_{ij}(t-1), \\ \text{if } \frac{\partial E}{\partial w_{ij}}(t) \cdot \frac{\partial E}{\partial w_{ij}}(t-1) < 0$$

Due to that ‘backtracking’ weight-step, the derivative is supposed to change its sign once again in the following step. In order to avoid a double punishment of the update-value, there should be no adaptation of the update-value in the succeeding step. In practice this can be done by setting $\frac{\partial E}{\partial w_{ij}}(t-1) = 0$ in the Δ_{ij} update-rule above.

It should be noted that by replacing $\Delta_{ij}(t)$ by a constant update-value Δ , the weight-update rule yield the so-called *Manhattan* rule.

The partial derivative of the total error is given by

$$\frac{\partial E}{\partial w_{ij}}(t) = \frac{1}{2} \sum_{p=1}^P \frac{\partial E_p}{\partial w_{ij}}(t) = -\frac{1}{2} \sum_{p=1}^P \delta_{pj} \cdot a_{pi}$$

Hence, the signal errors δ_{pj} must be accumulated for all P training patterns. This means that the weights are updated only after the presentation of all training patterns

In [10] it is introduced a *weight-decay* parameter α . This parameter determines the relationship of two goals, namely to reduce the output error (the standard goal) and to reduce the size of the weights (to improve generalization). The composite error function is:

$$E = \frac{1}{2} \sum_{p=1}^P \sum_{j=1}^{N_o} (d_{pj} - a_{pj})^2 + \frac{1}{10^\alpha} \cdot \sum_{i,j} w_{ij}^2.$$

Note that the *weight-decay* parameter α denotes the exponent, to allow comfortable input of very small values. Hence, a choice of $\alpha=4$ corresponds to a ratio of weight decay term to output error of 1:10000.

We shall use this version of *RPROP* in this paper.

2.2 Parameters

In order to reduce the number of freely adjustable parameters, often leading to a tedious search in parameter space, the increase and decrease factors η^+ and η^- are set to fixed values. The choice of the decrease factor η^- was led by the following considerations. If a jump over a minimum occurred, the previous update-value was too large. For it cannot be derived from gradient information how much the minimum was missed, we have to estimate the correct value. In average it will be a good guess to halve

the update-value, so we choose $\eta^- = 0.5$.

The increase factor η^+ , on the one hand, has to be large enough to allow fast growth of the update-value in shallow regions of the error function, but on the other hand the learning process can be considerably disturbed if a too large increase factor leads to persistent changes of the direction of the weight-step. After several experiments [4], M. Riedmiller decided to constantly fix the increase parameter to $\eta^+ = 1.2$. Slight variations of this value did neither improve nor deteriorate convergence time.

At the beginning of the algorithm, all update-values Δ_{ij} are set to an initial value Δ_0 . For Δ_0 directly determines the size of the first weight-step, it is preferably chosen in a reasonable proportion to the size of initial weights. A good choice may be $\Delta_0 = 0.1$. However, the choice of this parameter is not critical at all, for it is adapted as learning proceeds. Even for much larger or much smaller values of Δ_0 fast convergence is reached.

In order to prevent the weights from becoming too large, the maximum weight-step determined by the size of the update-value is limited. The upper bound is set by the second parameter of *RPROP*, Δ_{\max} . The default upper bound is set somewhat arbitrarily to $\Delta_{\max} = 50.0$. Usually, the convergence is rather insensitive to this parameter as well. Nevertheless, for some problems it can be advantageous to allow only very cautious (namely small) steps, in order to prevent the algorithm getting stuck too quickly in suboptimal local minima. This is realized by setting the maximum update-value to a considerably smaller value, e.g. $\Delta_{\max} = 0.1$. The minimum

step size is constantly fixed to $\Delta_{\min} = 1e^{-6}$.

The third parameter of the algorithm is the weight-decay parameter α

2.3 Algorithm

The following pseudo-code fragment shows the kernel of the *RPROP* adaptation and learning process. The **min** (**max**) operator is supposed to deliver the minimum (maximum) of two numbers; the **sign** operator returns +1 if the argument is positive, -1 if the argument is negative, and 0 otherwise.

$$\forall i, j : \Delta_{ij}(t) = \Delta_0$$

$$\forall i, j : \frac{\partial E}{\partial w_{ij}}(t-1) = 0$$

Repeat

- **Compute Gradient** $\frac{\partial E}{\partial w}(t)$;
- **For all weights and biases** {
 - if** $(\frac{\partial E}{\partial w_{ij}}(t) \cdot \frac{\partial E}{\partial w_{ij}}(t-1) > 0)$ **then** {
 - $\Delta_{ij}(t) = \min(\Delta_{ij}(t-1) * \eta^+, \Delta_{\max})$
 - $\Delta w_{ij}(t) = -\text{sign}(\frac{\partial E}{\partial w_{ij}}(t)) * \Delta_{ij}(t)$
 - $w_{ij}(t+1) = w_{ij}(t) + \Delta w_{ij}(t)$
 - $\frac{\partial E}{\partial w_{ij}}(t-1) = \frac{\partial E}{\partial w_{ij}}(t)$
 - else if** $(\frac{\partial E}{\partial w_{ij}}(t) \cdot \frac{\partial E}{\partial w_{ij}}(t-1) < 0)$ **then** {
 - $\Delta_{ij}(t) = \max(\Delta_{ij}(t-1) * \eta^-, \Delta_{\min})$
 - $\frac{\partial E}{\partial w_{ij}}(t-1) = 0$
 - else if** $(\frac{\partial E}{\partial w_{ij}}(t) \cdot \frac{\partial E}{\partial w_{ij}}(t-1) = 0)$ **then** {

$$\Delta w_{ij}(t) = -\text{sign}(\frac{\partial E}{\partial w_{ij}}(t)) * \Delta_{ij}(t)$$

$$w_{ij}(t+1) = w_{ij}(t) + \Delta w_{ij}(t)$$

$$\frac{\partial E}{\partial w_{ij}}(t-1) = \frac{\partial E}{\partial w_{ij}}(t)$$

}

}

Until (converged)

2.4 Discussion

To summarize, the basic principle of *RPROP* is the direct adaptation of the weight update-values Δ_{ij} . In contrast to learning-rate based algorithms, only the sign of the partial derivative is used to perform both learning and adaptation. As a result, the adaptation effort is not blurred by un-foreseeable gradient-behaviour. This leads to a transparent and yet powerful adaptation process, that can be straightforward and very efficiently computed with respect to both time and storage consumption.

Besides fast convergence, one of the main advantages of *RPROP* lies in the fact that for many problems the choice of at most one parameter (the *weight-decay* parameter) is needed to obtain optimal or at least nearly optimal convergence times.

Another often discussed aspect of common gradient descent is that the size of the derivative decreases exponentially with the distance between the weight and the output layer, due to the limiting influence of the slope of the sigmoid activation function. Consequently, the weights far away from the output layer are less modified and do learn much slower. Using *RPROP*, the size of the weight-step is only dependent on the sequence of signs, not on the magnitude of the derivative. For that reason, learning is spread equally all over the

entire network; weights near the input layer have the equal chance to grow and learn as weights near the output layer.

RPROP suffers from the same problem as do any of the adaptive learning algorithms [1, 3, 8]. Because the adaptation is based on an estimation of the topology of the error function, both adaptation and weight update can be first performed after the whole gradient information is available, in other words after each pattern has been presented and the gradient of the sum of pattern errors is known. Accordingly, adaptive procedures are typically based on *batch learning* or *learning by epoch*. This possibly reduces their efficiency on redundant training sets compared to simple stochastic gradient descent and poses problems on their use with variable training sets.

Nevertheless, the results obtained in [3, 5] are very promising and confirm the quality of *RPROP* algorithm with respect to both convergence time and robustness.

3. Experimental model

A time series is a sequence of time-ordered data values that are measurements of some physical process. A time series forecasting problem can be easily mapped to a feed-forward neural network [7]. The number of input units corresponds to the number of input data terms. The number of output units represents the forecast horizon. One-step-ahead forecast can be performed by a neural network with one output unit, and k -step-ahead forecast can be mapped to a neural network with k output units.

This paper presents an application of feed-forward neural networks and *RPROP* algorithm to forecast a time series consisting of the average monthly

liquid flow measured at Broșteni hydrometric station from the Motru River. The data are from the period 1950-1994 [12]. A forecasting is correct if its value presents a deviation of at most $\pm 10\%$ from the measured value.

The time series has 528 terms. The first 500 terms were used as training patterns and the last 28 terms of the time series were used to test the forecasting performance of the models.

We used in our experiments the program *SNNS (Stuttgart Neural Network Simulator) 4.0*, a software simulator for neural networks on Unix workstations developed at the Institute for Parallel and Distributed High Performance Systems (Institut für Parallele und Verteilte Höchstleistungsrechner, IPVR) at the University of Stuttgart since 1989.

SNNS is an efficient and flexible simulation environment for research on and application of neural nets, consisting of four main components: simulator kernel, graphical user interface, batch simulator version *snnsbat*, and network compiler *snns2c*. The simulator kernel operates on the internal network data structures of the neural nets and performs all operations on them. The graphical user interface XGUI, built on top of the kernel, gives a graphical representation of the neural networks and controls the kernel during the simulation run. In addition, the user interface can be used to directly create, manipulate and visualize neural nets in various ways. Complex networks can be created quickly and easily. An online help system, partly context-sensitive, is integrated, which can offer assistance with problems. *SNNS* is implemented completely in ANSI-C and XGUI is based upon X11 Release 5 from MIT and the Athena Toolkit [10].

In this paper, we used the batch simulator version *snnsbat*, which we improved in order to choose from an input file the network structure, the values of the parameters and the number of runs for each such choice and also to display the results in the desired appearance.

All the experiments were carried out on a Pentium at 133 MHz and with 24MB RAM, on Linux operating system.

The feed-forward neural networks used in the experiments had 12 input units, one hidden layer and one output unit. We tested different neural network structures by varying the number of hidden units and we used different values for the parameters appearing in the algorithm *RPROP*.

For each neural network structure and parameter setting, 20 experiments were run with different random initial weights, uniformly distributed over the intervals $[-1,1]$ and $[-0.5, 0.5]$. The reported forecast errors are the averages of the 20 runs. The data series were normalized to the range $[0,1]$ before feeding into the neural networks:

$$data_{(0,1)j} = \frac{largst_data - data}{largst_data - smallest_data}$$

Forecasts from the neural network outputs were transformed to the original data scale before the percentage of good forecasts was reported.

The parameters used by the algorithm are:

- Δ_0 We set this parameter to 0.1.
- Δ_{max} We also set this parameter to 50.0.
- α -the *weight-decay* parameter. We used in the experiments the following values for α : 0, 1, 5, 10

- Nh -the number of units from the hidden layer. We used neural networks with 0-12 hidden units.

The next tables present the results obtained after 2000 epochs:

- **Table 1:** the results obtained in the experiments with random initial weights, uniformly distributed over the intervals $[-1,1]$.
- **Table 2:** the results obtained in the experiments with random initial weights, uniformly distributed over the intervals $[-0.5,0.5]$.
- **Table 3:** the results obtained by using neural networks with direct connections from the input units to the output units, the initial weights being uniformly distributed over $[-0.5,0.5]$. The insertion of direct connections is equivalent to adding linear components to the network.

Table 1

	$\alpha=0$	$\alpha=3$	$\alpha=5$	$\alpha=10$
Nh=0	17.00%	17.00%	17.00%	17.00%
Nh=1	10.00%	10.00%	14.00%	10.00%
Nh=2	19.35%	13.30%	20.55%	20.15%
Nh=3	15.45%	14.00%	18.90%	15.20%
Nh=4	13.55%	16.60%	15.80%	16.35%
Nh=5	11.60%	21.20%	13.15%	13.10%
Nh=6	11.25%	22.85%	16.05%	10.80%
Nh=7	12.75%	22.60%	12.55%	9.05%
Nh=8	8.75%	19.05%	10.25%	12.65%
Nh=9	9.15%	20.10%	10.75%	11.80%
Nh=10	8.70%	19.60%	9.40%	7.55%
Nh=11	8.65%	17.65%	7.50%	7.15%
Nh=12	8.45%	17.65%	7.65%	7.85%

Table 2

	$\alpha=0$	$\alpha=3$	$\alpha=5$	$\alpha=10$
Nh=0	17.00%	17.00%	17.00%	17.00%
Nh=1	10.00%	10.00%	14.00%	10.00%
Nh=2	23.70%	13.30%	26.65%	21.25%
Nh=3	17.05%	15.60%	19.90%	21.60%
Nh=4	14.40%	20.45%	16.70%	13.50%
Nh=5	13.70%	22.55%	13.45%	14.35%
Nh=6	13.35%	21.70%	10.80%	12.55%
Nh=7	10.00%	20.60%	12.05%	12.15%
Nh=8	10.75%	20.80%	10.05%	11.15%
Nh=9	9.55%	20.00%	9.05%	9.65%
Nh=10	9.95%	19.90%	7.95%	9.50%
Nh=11	8.80%	19.80%	12.00%	10.65%
Nh=12	10.45%	19.55%	9.60%	8.00%

Table 3

	$\alpha=0$	$\alpha=3$	$\alpha=5$	$\alpha=10$
Nh=1	14.45%	11.55%	15.00%	14.90%
Nh=2	13.30%	18.85%	14.65%	15.35%
Nh=3	16.10%	18.75%	15.55%	13.45%
Nh=4	12.35%	19.10%	14.70%	13.55%
Nh=5	10.90%	20.80%	9.75%	10.55%
Nh=6	12.00%	17.90%	12.60%	11.95%
Nh=7	10.15%	18.50%	11.50%	11.15%
Nh=8	10.15%	16.70%	10.70%	13.95%
Nh=9	7.65%	15.65%	10.75%	9.55%
Nh=10	9.25%	17.95%	10.15%	9.25%
Nh=11	7.65%	15.20%	9.60%	9.85%
Nh=12	8.35%	15.85%	7.60%	8.20%

4. Conclusions

We remark that the best results obtained with neural networks without direct connections were:

Table 1

- 22.85% with Nh=6, $\alpha = 3$;
- 22.60% with Nh=7, $\alpha = 3$;
- 21.20% with Nh=5, $\alpha = 3$;
- 20.55% with Nh=2, $\alpha = 5$;
- 20.15% with Nh=2, $\alpha = 10$;
- 20.10% with Nh=9, $\alpha = 3$.

Table 2

- 26.65% with Nh=2, $\alpha = 5$;
- 23.70% with Nh=2, $\alpha = 0$;
- 21.70% with Nh=6, $\alpha = 3$;
- 21.60% with Nh=3, $\alpha = 10$;
- 21.25% with Nh=2, $\alpha = 10$.

Hence, the best correct percentage is 26.65%. We see that better results were got in the experiments with random initial weights, uniformly distributed over $[-0.5, 0.5]$ and with neural networks with less hidden units. The networks with more hidden units learned better the training examples, but they got worse results on the testing examples. We also notice that better correct percentages were obtained with a value of *weight-decay parameter* $\alpha = 3$.

The best results obtained with direct connections were:

Table 3

- 20.80% with $N_h=5$, $\alpha = 3$;
- 19.10% with $N_h=4$, $\alpha = 3$;
- 18.85% with $N_h=2$, $\alpha = 3$;
- 18.75% with $N_h=3$, $\alpha = 3$;
- 18.50% with $N_h=7$, $\alpha = 3$.

Hence, adding direct connections from the input units to the output units did not improve the results.

Experiments showed that the performances depend on the network structure and parameters setting. Combining these factors the performances can be improved.

The forecasting of the hydrological system from an area using neural networks represents a new method of investigations in the hydrological field and can contribute to hydrological prognosis improvement.

REFERENCES

1. Jacobs R., **Increased rates of convergence through learning rate adaptation**, Neural Networks, Vol. 1, 1988, pp. 295-307.
2. Riedmiller M., **Untersuchungen zu Konvergenz und**

Generalisierungs-verhalten überwachter Lernverfahren mit dem SNNS, in A. Zell, editor, SNNS 1999 Workshop Proceedings, Stuttgart, September 1993.

3. Riedmiller M., **Advanced supervised learning in multi-layer perceptrons-from Backpropagation to adaptive learning algorithms**, International Journal on Computer Standards and Interfaces, Vol. 16, 1994, pp. 265-278.
4. Riedmiller M., **Rprop-description and implementation details**, Technical Report, University of Karlsruhe, January 1994.
5. Riedmiller M. and Braun H., **A direct adaptive method for faster backpropagation learning: The RPROP algorithm**, in H. Ruspini, editor, Proceedings of the IEEE International Conference on Neural Networks (ICNN), San Francisco, USA, 1993, pp. 586-591.
6. Rumelhart D.E., Hinton G. and Williams R., **Learning internal representations by error propagation**, in D. E. Rumelhart, J. McClelland, & the PDP Research Group, Parallel Distributed Processing: Explorations in the Microstructure of Cognition, Vol. I Foundations, MIT Press, Cambridge, MA, 1986, pp. 318-364.
7. Tang Z. and Fishwick P., **Feed-forward neural nets as models for time series forecasting**, Technical report, University of Florida, Gainesville, 1993.
8. Tollenaere T., **SuperSab: Fast adaptive backpropagation with good scaling properties**, Neural Networks, Vol. 3, 1990, pp. 561-573.
9. Ujvary L., **Geografia apelor României**, Scientific Ed., Bucharest, 1972 (in Romanian).

10. *****, SNNS-Stuttgart Neural Network Simulator, User Manual, Version 4.0**, University of Stuttgart, Institute for Parallel and Distributed High Performance Systems, Technical Report 6/1995.
11. *****, Monografia hidrologică a bazinului hidrografic al râului Jiu, Studii de hidrologie XV, I S C H** Bucharest, 1966 (in Romanian).
12. *****, Anuarele hidrologice din perioada 1944-1994**, C.N.A. - I.M.H. Bucharest (in Romanian).