Published in *Soft Computing*, vol. 11, No 2, January, 2007, pp. 169-183.

Multilayer Feedforward Neural Network Based on Multi-Valued Neurons (MLMVN) and a Backpropagation Learning Algorithm

IGOR AIZENBERG

Department of Computer and Information Sciences

Texas A&M University-Texarkana P.O. Box 5518 2600 N. Robison Rd.

Texarkana, Texas 75505 USA

e-mail Igor.Aizenberg@tamut.edu

CLAUDIO MORAGA

Department of Computer Science-1

University of Dortmund 44221 Dortmund Germany

e-mail Claudio.Moraga@udo.edu

Abstract

A multilayer neural network based on multi-valued neurons is considered in the paper. A multi-valued neuron (MVN) is based on the principles of multiple-valued threshold logic over the field of the complex numbers. The most important properties of MVN are: the complex-valued weights, inputs and output coded by the k^{th} roots of unity and the activation function, which maps the complex plane into the unit circle. MVN learning is reduced to the movement along the unit circle, it is based on a simple linear error correction rule and it does not require a derivative. It is shown that using a traditional architecture of multilayer feedforward neural network (MLF) and the high functionality of the multi-valued neuron, it is possible to obtain a new powerful neural network. Its training does not require a derivative of the activation function and its functionality is higher than the functionality of MLF containing the same number of layers and neurons. These advantages of MLMVN are confirmed by testing using parity n, two spirals and "sonar" benchmarks and the Mackey-Glass time series prediction.

Keywords: feedforward complex-valued neural network, derivative free backpropagation learning

I. INTRODUCTION

Neural networks with a backpropagation learning algorithm have started their history from the ideas presented by D.E. Rumelhart and J.L McClelland in [37]. These neural networks are characterized by a multi-layer architecture with a feedforward dataflow through nodes requiring full connection between consecutive layers. This architecture is the result of a "universal approximator" computing model based on Kolmogorov's Theorem [27] (see e.g. [18], [13] and the more comprehensive observation done by R. Hecht-Nielsen in [19]). It has been shown in [21], [39] that these neural networks are universal approximators. So there are two main ideas behind a feedforward neural network. The first idea is a full connection architecture: the outputs of neurons from the previous layer are connected with the corresponding inputs of all neurons of the following layer. The second idea is a backpropagation learning algorithm, when the errors of the neurons from the output layer are being sequentially backpropagated through all the layers from the "right hand" side to the "left hand" side, in order to calculate the errors of all other neurons. One more common property of a major part of the feedforward networks is the use of sigmoid activation functions for its neurons.

It is possible to find hundreds of papers and many books published during last 10-15 years, where the ideas of multilayer neural networks and backpropagation learning were developed. One of the most comprehensive observations is presented e.g. in [17].

At the same time, there is at least one important problem, which is still open. Although it is proven that a feedforward network is a universal approximator, a practical implementation of learning often is a very complicated task. It depends on several factors: the complexity of the mapping to be implemented, the chosen structure of the network (the number of hidden layers, the number of neurons on each layer), and the control over the learning process (usually this control is being implemented using the learning rate). Increasing both the number of hidden layers and neurons on them, we can make the network more flexible to the mapping to be implemented. This corresponds to the well-known Cover's theorem [10] on the separability of patterns, which states that a pattern classification problem is more likely to be linearly separable in a high dimensional feature space than in a low dimensional one, while projected into a high dimensional space nonlinearly. However, the computations in a higher dimensional space require more resources and much more time. The case of multilayer feedforward neural network (MLF, which is also often referred to as MLP – a "multilayer perceptron") and similar networks is not an exclusion. Moreover, increasing the number of hidden neurons increases the risk of overfitting.

In order to minimize the implementation of complex mappings, several supporting algorithms were proposed. A very popular family of kernel-based learning algorithms that use nonlinear mappings from input spaces to high dimensional feature spaces should be mentioned. The best known of them is the support vector machine (SVM) introduced in [40]. Different kernel-based techniques are considered e.g. in [34]. A fuzzy kernel perceptron [9] should be distinguished among the most recent publications, where the kernel-based approach is developed.

Another direction is related to improvement of the MLF learning, search for more sophisticated learning techniques, as well as different modifications of the MLF structure and architecture. From the very recent publications we can mention [12], where the modular feedforward networks, with not fully connected neighboring layers are studied and [32], where the original modification of the MLF learning algorithm is considered.

At the same time it is very attractive to consider a different solution, which will preserve a traditional MLF architecture, however, it will be based on the use of the different basic neurons. We will consider in this paper a *multilayer neural network based on multi-valued neurons* (MLMVN). A multi-valued neuron (MVN) was introduced in [6]. It is based on the principles of multiple-valued threshold logic over the field of the complex numbers formulated in [4] and then developed in [5]. A comprehensive observation of MVN, its properties and learning is presented in [1]. The most important properties of MVN are: the complex-valued weights, inputs and output coded by the k^{th} roots of unity and the activation function, which maps the complex plane into the unit circle. It is very important for us that MVN learning is reduced to the movement along the unit circle. The MVN learning algorithm is based on a simple linear error correction rule and it does not require a derivative. Moreover, it converges very quickly. Different applications of MVN have been considered during the last years. We will mention some of them: MVN has been used as a basic neuron in cellular neural networks [1], as a basic neuron of neural-based associative memories [1], [25], [8] and as the basic neuron of different pattern recognition systems [8], [2].

It is very important that the functionality of a single MVN is higher than the functionality of a single neuron with a sigmoid activation function [1]. So it is very attractive to consider a multilayer neural network with the same architecture as a MLF, but with MVN as a basic neuron. It should be mentioned that feedforward networks based on neurons with the complex-valued weights have been already considered, for example in [29], [35]. But in these papers a classical sigmoid function and a classical backpropagation algorithm were generalized for the complex-valued case. We will consider here a different solution.

It is also important to mention that the development of the complex-valued neural networks is becoming more and more popular. It is possible to refer to the recently published book [20], where, for example, the MVN-based associative memory presented in [7] and the complex domain backpropagation presented in [35] are observed in more details.

In the Section II we will remind some basic ideas related to MVN and its training. A modified continuous-valued activation function also will be introduced. In the Section III the MLMVN will be introduced. We will consider a backpropagation learning algorithm for it, which does not require a derivative of the activation function. Simulation results will be presented in the Section IV. Using some standard benchmarks, we will show the efficiency of MLMVN. It will be shown that such popular problems as parity *n*, two spirals, "sonar" and the Mackey-Glass time series prediction can be solved using a simpler and smaller network than the known ones.

II. Discrete and Continuous MVN

Let us remind some basic ideas related to MVN and its training. A single MVN performs a mapping between *n* inputs and a single output ([6], [1]). This mapping is described by a multiple-valued (*k*-valued) function of *n* variables $f(x_1,...,x_n)$ with *n*+1 complex-valued weights as parameters:

$$f(x_1,...,x_n) = P(w_0 + w_1x_1 + ... + w_nx_n) \tag{1}$$

where $x_1,...,x_n$ are the variables, on which the performed function depends and $w_0, w_1, ..., w_n$ are the weights. The values of the function and of the variables are complex. They are the *k*th roots of unity: $\varepsilon^j = \exp(i2\pi j/k)$, $j \in \{0, k\text{-}1\}$, *i* is an imaginary unity. *P* is the activation function of the neuron:

$$P(z)=\exp(i2\pi j/k),\ if\ \ 2\pi j/k \le \arg z < 2\pi\ (j+1)/k \qquad (2)$$

where j=0, 1, ..., k-1 are values of the k-valued logic, $z = w_0 + w_1x_1 + ... + w_nx_n$ is the weighted sum , arg z is the argument of the complex number z. Equation (2) is illustrated in Fig. 1. Function (2) divides a complex plane onto k equal sectors and maps the whole complex plane into a subset of points belonging to the unit circle. This is exactly a set of kth roots of unity. Function (2) was initially proposed by N. Aizenberg et. al. in [4] as a k-valued predicate *csign*, which is a key element of multiple-valued threshold logic over the field of the complex numbers [5], [1].

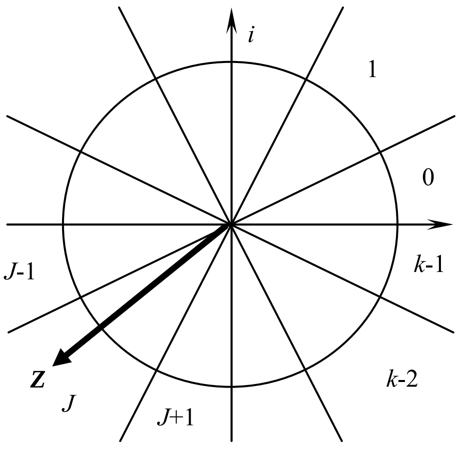


Fig. 1 Geometrical interpretation of the MVN activation function

MVN training is reduced to the movement along the unit circle. This movement does not require a derivative of the activation function, because it is impossible to move in the incorrect direction. Any direction of movement along the circle will lead to the target. The shortest way of this movement is completely determined by the error that is a difference between the "target" and the "current point", i.e. between the desired and actual outputs, respectively. This MVN property is very important for the further development of the learning algorithm for a multilayer network. Let us consider how it works.

Let ε^q be a desired output of the neuron (see Fig. 2). Let $\varepsilon^s = P(z)$ be an actual output of the neuron. The MVN learning algorithm based on the error correction learning rule is defined as follows [1]:

$$W_{r+1} = W_r + \frac{C_r}{(n+1)}(\varepsilon^q - \varepsilon^s)\,\overline{X}\,, \tag{3}$$

where X is an input vector[1], n is the number of neuron inputs, $\overline{X}$ is a vector with the components complex conjugated[2] to the components of vector X, r is the number of the learning iteration, W_r is a current weighting vector (to be corrected), W_{r+1} is the following weighting vector (after correction), C_r is a learning rate. The convergence of the learning process based on the rule (3) is proven in [1]. What is the sense of the rule (3)? It ensures such a correction of the weights that a weighted sum is moving from the sector s to the sector q (see Fig. 2). The direction of this movement is completely defined by the difference $\delta = \varepsilon^q - \varepsilon^s$.Thus $\delta = \varepsilon^q - \varepsilon^s$ determines the MVN error. According to (3) a correcting item $\Delta w_i = \frac{C_r}{(n+1)}(\varepsilon^q - \varepsilon^s)\,\overline{x}_i$, $i = 0,1,...,n$, which is added to the corresponding weight w_i $(i = 0,1,...,n)$ of the weighting vector W_r in order to correct it and to obtain the weighting vector W_{r+1}, is proportional to δ. Here $\overline{x}_i\,(i = 0,1,...,n)$ is the ith component of the input vector $\overline{X}$ with the components complex conjugated.

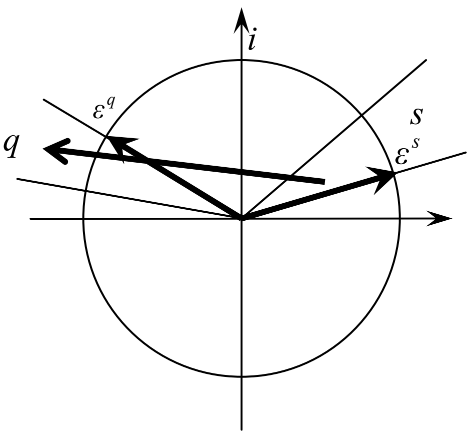


Fig. 2 Geometrical interpretation of the MVN learning rule

[1] We will add to the n-dimensional vector X an n+1th component $x_0 \equiv 1$ realizing a bias, in order to simplify mathematical expressions for the correction of the weights

[2] Here and further $\overline{x}$ is a number complex conjugated to x and $\overline{X}$ is a vector with the components complex conjugated to the components of X.

The correction of the weights according to (3) changes the value of the weighted sum exactly on δ. Indeed, let $z = w_0 + w_1x_1 + \ldots + w_nx_n$ be a current weighted sum. Let us correct the weights according to the rule (3) (we take *C*=1):

$$\tilde{w}_0 = w_0 + \frac{\delta}{(n+1)}; \quad \tilde{w}_1 = w_1 + \frac{\delta}{(n+1)}\bar{x}_1; \quad \ldots; \quad \tilde{w}_n = w_n + \frac{\delta}{(n+1)}\bar{x}_n .$$

The weighted sum after the correction is obtained as follows:

$$\begin{aligned} \tilde{z} &= \tilde{w}_0 + \tilde{w}_1x_1 + \ldots + \tilde{w}_nx_n = (w_0 + \frac{\delta}{(n+1)}) + (w_1 + \frac{\delta}{(n+1)}\bar{x}_1)x_1 + \ldots + (w_n + \frac{\delta}{(n+1)}\bar{x}_n)x_n = \\ &= w_0 + \frac{\delta}{(n+1)} + w_1x_1 + \frac{\delta}{(n+1)} + \ldots + w_nx_n + \frac{\delta}{(n+1)} = \\ &= w_0 + w_1x_1 + \ldots + w_nx_n + \delta = z + \delta \ . \end{aligned} \tag{4}$$

Equation (4) shows the importance of the factor $\frac{1}{n+1}$ in the learning rule (3). This factor shares the error δ uniformly among the neuron's weights.

Evidently, the activation function (2) is discrete. More exactly, it is piece-wise discontinuous, it has discontinuities on the borders of the sectors. Let us modify the function (2) in order to generalize it for the continuous case in the following way. Let us consider, what will happen, when $k \to \infty$ in (2). It means that the angle value of the sector (see Fig. 1) will go to zero. It is easy to see that the function (2) is transformed in this case as follows:

$$P(z) = \exp(i\,(\arg z)) = e^{iArg\, z} = \frac{z}{|z|}, \tag{5}$$

where *z* is the weighted sum, Arg *z* is a main value of its argument and |*z*| is a modulo of the complex number *z*.

The function (5) maps the complex plane into the whole unit circle, while the function (2) maps a complex plane just on a discrete subset of the points belonging to the unit circle. The function (2) is discrete, while the function (5) is continuous. We will use here exactly the function (5) as the activation function for the MVN. Both functions (2) and (5) are not differentiable as functions of a complex variable, but this is not important, because their differentiability is not required for MVN training. The learning rule (3) will be modified for the continuous-valued case in the following way:

$$W_{r+1} = W_r + \frac{C_r}{(n+1)}(\varepsilon^q - e^{iArg\,z})\,\overline{X} = W_r + \frac{C_r}{(n+1)}\left(\varepsilon^q - \frac{z}{|z|}\right)\overline{X}\,, \tag{6}$$

where r is the number of the learning iteration.

To prove the convergence of the learning algorithm based on the learning rule (6), we have to take into account the following. The convergence in the continuous-valued case means that the following condition must be satisfied for the error: $\left|\arg(\varepsilon^q) - \arg(e^{iArg\,z})\right| < \lambda$, where λ determines the precision of the learning. However, in this case the learning rule (6) is reduced to the learning rule (3) for the discrete case, because for the argument of the neuron's output we obtain the following condition: $\arg(\varepsilon^q) - \lambda < \arg(e^{iArg\,z}) < \arg(\varepsilon^q) + \lambda$. This means that the continuous-valued learning is reduced to the discrete-valued learning in a $\frac{\pi}{\lambda}$ - valued logic. Indeed, it follows from the last considerations, that we obtain exactly $\frac{\pi}{\lambda}$ sectors on the complex plane (see Fig. 1) and $k = \frac{\pi}{\lambda}$ in (2). But for the discrete-valued case the convergence of the learning algorithm is proven for any k in [1].

It is also interesting to consider the following modification of (6):

$$W_{m+1} = W_m + \frac{C_m}{(n+1)}\tilde{\delta}\,\overline{X}\,, \tag{7}$$

where $\tilde{\delta}$ is obtained from $\delta = \varepsilon^q - \frac{z}{|z|} = T - \frac{z}{|z|}$ using a normalization by the factor $\frac{1}{|z|}$:

$$\tilde{\delta} = \frac{1}{|z|}\delta = \frac{1}{|z|}\left(T - \frac{z}{|z|}\right), \tag{8}$$

Learning according to the rule (7)-(8) makes it possible to squeeze a space for the possible values of the weighted sum. Using (7)-(8) instead of (6), we can reduce this space to the respectively narrow ring, which will include the unit circle inside (see Fig. 3). Indeed, if $|z|<1$ and we correct the weights according to (7)-(8), then according to $|\tilde{z}|>|z|$, $\tilde{z}$ will be closer to the unit circle than *z*, approaching the unit circle from "inside". If $|z|>1$ and we correct the weights according to (7)-(8), then according to $|\tilde{z}|<|z|$, $\tilde{z}$ will be closer to the unit circle than *z*, approaching the unit circle from "outside".

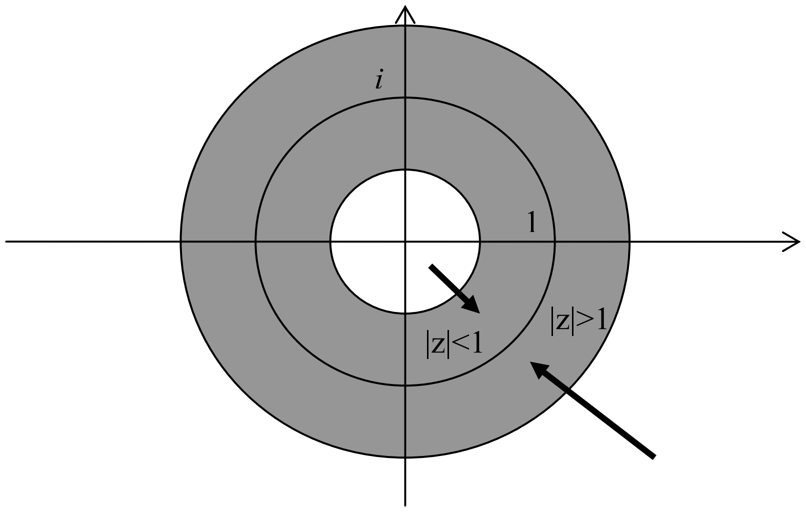


Fig. 3. Normalization of the weighted sum z by the factor 1/|z| (see (7)- (8))

This approach can be useful in order to make *z* more smooth, as a function of the weights and to exclude a situation, when a small change in any of the weights or the inputs will lead to a significant change of *z*. We will return below to the choice of the learning rule, when we will discuss a learning algorithm for the MVN-based neural network (see Sections III.B and IVIV.A).

III. Multilayer MVN-based Neural Network and a Backpropagation Learning Algorithm

A. General considerations

Let us consider a multilayer neural network with traditional feedforward architecture, when the outputs of neurons of the input and hidden layers are connected with the corresponding inputs of the neurons from the following layer. Let us suppose that the network contains one input layer, *m*-1 hidden layers and one output layer. We will use here the following notations.

Let

T_{km} - be a desired output of the *k*th neuron from the output (*m*th) layer

Y_{km} - be the actual output of the *k*th neuron from the output (*m*th) layer.

Then a global error of the network taken from the *k*th neuron of the output (*m*th) layer can be calculated as follows:

$$\delta^*_{km} = T_{km} - Y_{km} \text{ - error taken from the } k^{\text{th}} \text{ neuron of the output } (m^{\text{th}}) \text{ layer} \tag{9}$$

δ^*_{km} will denote here and further a global error of the network. We have to distinguish it from the local errors δ_{km} of the particular output neurons because the error δ^*_{km} consisted not only of the error δ_{km} of the output neuron, but of the errors of the hidden neurons.

The learning algorithm for the classical feedforward network is derived from the consideration that the global error of the network in terms of the *square error* (SE) must be minimized. The (normalized) square error is defined as follows:

$$\mathrm{E}(W) = \frac{1}{2}\sum_{k=1}^{N_m}(\delta^*_{km})^2(W) \tag{10}$$

where *W* denotes the weighting vectors of all the neurons of the network and N_m indicates the number of output neurons. It is a principal assumption that the error depends not only on the weights of the neurons at the output layer, but on all neurons of the network.

The functional of error may be defined as follows:

$$E_{ms} = \frac{1}{N}\sum_{s=1}^{N} E_s \ , \tag{11}$$

where E_{ms} denotes the (normalized) *mean square error*, *N* is the total number of patterns in the training set and E_s denotes the (normalized) square error of the network for the *s*th pattern.

The minimization of the functional (11) is reduced to the search for those weights for all the neurons that ensure a minimal.

Let us briefly remind how it works in the case of a MLF.

For the next analysis, the following notation will be used. As shown in eq. (1), we identify a weight as usual with a subscript index of the variable, to which it is associated. Here we have to associate a weight with two more indices to identify the corresponding neuron and layer. In what follows, since we want to identify, to which neuron a weight is applied, we will write as superscript of a weight the indices corresponding to the respective neuron and layer. Accordingly w_i^{jk} denotes the weight corresponding to the *i*th input of the *j*th neuron at the *k*th layer. Furthermore, let z_{jk}, y_{jk} and $Y_{jk} = y_{jk}(z_{jk})$ represent the weighted sum (of the input signals), the activation function and the output signal of the *j*th neuron at the *k*th layer, respectively. Let N_k be the number of neurons in the *k*th layer (notice that this means that neurons of the *k*+1st layer will have exactly N_k inputs.) Finally recall that $x_1,\dots,x_n$ denote the inputs to the network (and as such, also the inputs to the neurons of the first layer).

The correction of the weights of all neurons is made in such a way that each weight w_i has to be corrected by an amount Δw_i, which must be proportional to the gradient $\frac{\partial E}{\partial w_i}$ of the error function *E*(*W*) with respect to the weights [17]:

$$\Delta w_i^{jm} = -\alpha \frac{\partial E(W)}{\partial w_i^{jm}} = \begin{cases} \alpha \cdot \delta_{jm}^* \cdot y'_{jm}(z_{jm}) \cdot Y_{i(m-1)} & i = 1,\dots,\ N_{m-1} \\ \alpha \cdot \delta_{jm}^* \cdot y'_{jm}(z_{jm}) & i = 0, \end{cases} \tag{12}$$

where $\alpha > 0$ is a coefficient representing a learning rate.

The part of the rate of change of the square error *E*(*W*) with respect to the input weight of a neuron, which is independent of the value of the corresponding input signal to that neuron, will be called the local error (or simply the error) of that neuron. Accordingly, the local error of the *j*th neuron of the output layer, denoted by δ_{jm}, is given by [17]:

$$\delta_{jm} = y'_{jm}(z_{jm}) \cdot \delta_{jm}^*, \tag{13}$$

To propagate the error to the neurons of all hidden layers, a sequential error backpropagation through the network from the *m*th layer to the *m*-1st one, from the *m*-1st to the m-2nd one, ..., and from the 3rd to the 2nd one will be done. When the error is propagated from the layer *k*+1 to the layer *k*, the local error of each neuron of the *k*+1st layer is multiplied by the weight of the path connecting the corresponding input of this neuron at the *k*+1st layer with the corresponding output of the neuron at the *k*th layer. For example, the error $\delta_{j,k+1}$ of the *j*th neuron at the *k*+1st layer is propagated to the *l*th neuron at the *k*th layer, multiplying $\delta_{j,k+1}$ with $w_l^{j,k+1}$, namely the weight corresponding to the *l*th input of the *j*th neuron at the *k*+1st layer. This analysis leads to [17]:

$$\delta_{lk} = y'_{lk}(z_{lk}) \cdot \sum_{i=1}^{N_{k+1}} \delta_{i,k+1} w_l^{i,k+1} \text{ , } k=1, \ldots, m\text{-}1\text{- error for the } l^{\text{th}} \text{ neuron from the } k^{\text{th}} \text{ layer.} \qquad (14)$$

B. A backpropagation learning algorithm for the MLMVN

Let us now move back to the MLMVN. As it was mentioned from the beginning, the MVN activation function (5) is not differentiable. It means that the formulas (13)-(14) that determine the error calculation for MLF and (12) that determine the weights correction for MLF cannot be applied for the case of MLMVN because all of them contain the derivative of the activation function. However, for the MVN-based network this is not a problem!

As it was shown above for the single neuron, the differentiability of the MVN activation function is not required for learning. Since MVN learning is reduced to the movement along the unit circle, the correction of the weights is completely determined by the neuron's error. The same property is true not only for the single MVN, but for the network (MLMVN). The errors of all the neurons from MLMVN are completely determined by the global errors of the network (9). As well as MLF learning, MLMVN learning is based on the minimization of the error functional (11). Let us generalize all considerations that we made above for the single neuron for the case of MLMVN. As a result, we will obtain a backpropagation learning algorithm for the MLMVN, which will be based on the same idea that a backpropagation learning algorithm for MLF, but at the same time it will have some important and sharp distinctions.

To make things easier, let us start from the case, where a network contains one hidden layer and a single neuron in the output layer (see Fig. 4).

The error on the network output is equal to $\delta^* = T - Y_{12}$, where T is a desired output. We need to understand, how we can obtain the errors for each particular neuron, backpropagating the error δ^* from the right-hand side to the left-hand side. Let $(w_0^{12}, w_1^{12}, ..., w_n^{12})$ be an initial weighting vector of the neuron 12, Y_{i1} be an initial output of the neuron $i1$ from the hidden layer (i=1, …, n), Y_{12} be an initial output of the neuron 12, z_{12} be the weighted sum on the neuron 12 before the correction of the weights, δ_{i1} be the error of the neuron $i1$ from the hidden layer (i=1, …, n), and δ_{12} be the error of the neuron 12. Let us suppose that the errors for all the neurons of the network are already known. We will use the learning rule (6) for the correction of the weights. Let us suppose that all neurons from the 1st layer are already trained.

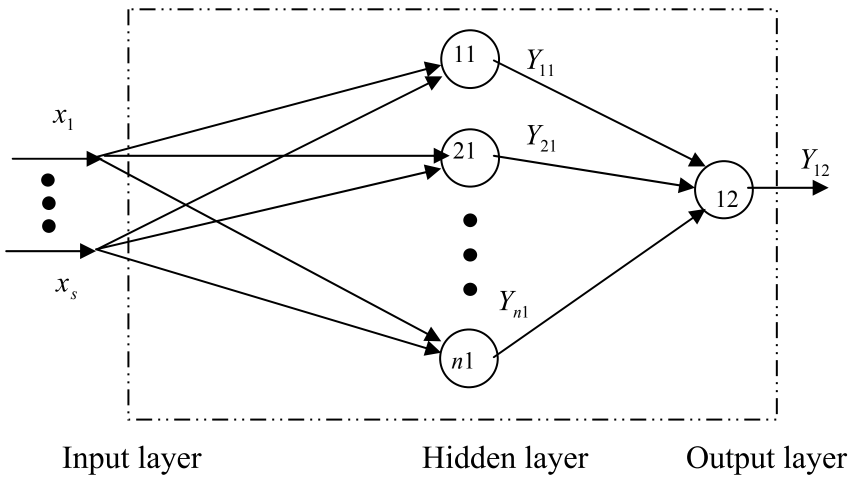


Fig. 4 A feedforward neural network with a single output neuron and with one hidden layer

Let us now correct the weights for the neuron 12 (the output neuron) and estimate its weighted sum (recall eq. (4)):

$$\tilde{z}_{12} = \left(w_0^{12} + \frac{1}{(n+1)}\delta_{12} \right) + \left(w_1^{12} + \frac{1}{(n+1)}\delta_{12}\overline{(Y_{11}+\delta_{11})} \right)\left(Y_{11}+\delta_{11}\right) + \ldots +$$
$$+ \left(w_n^{12} + \frac{1}{(n+1)}\delta_{12}\overline{(Y_{n1}+\delta_{n1})} \right)(Y_{n1}+\delta_{n1}) =$$
$$= \left(w_0^{12} + \frac{1}{(n+1)}\delta_{12} \right) + \sum_{i=1}^{n}\left(w_i^{12} + \frac{1}{(n+1)}\delta_{12}\overline{(Y_{i1}+\delta_{j1})} \right)(Y_{i1}+\delta_{i1}) =$$
$$= \left(w_0^{12} + \frac{1}{(n+1)}\delta_{12} \right) + \sum_{j=1}^{n}\left(w_i^{12}Y_{i1} + w_i^{12}\delta_{i1} + \frac{1}{(n+1)}\delta_{12}\overline{(Y_{i1}+\delta_{i1})}(Y_{i1}+\delta_{i1}) \right).$$

Taking into account that $\forall i = 1,\ldots,n \ (Y_{i1}+\delta_{i1})$ is an output of the neuron, this complex number is always laying on the unit circle and therefore its absolute value is always equal to 1. This means that $\forall i = 1,\ldots,n \ \overline{(Y_{i1}+\delta_{i1})}(Y_{i1}+\delta_{i1}) = 1$. Then we obtain the following:

$$\tilde{z}_{12} = \left(w_0^{12} + \frac{1}{(n+1)}\delta_{12} \right) + \sum_{i=1}^{n}\left(w_i^{12}Y_{i1} + w_i^{12}\delta_{i1} + \frac{1}{(n+1)}\delta_{12}\overline{(Y_{i1}+\delta_{i1})}(Y_{i1}+\delta_{i1}) \right) =$$
$$= \left(w_0^{12} + \frac{1}{(n+1)}\delta_{12} \right) + \sum_{i=1}^{n}\left(w_i^{12}Y_{i1} + w_i^{12}\delta_{i1} + \frac{1}{(n+1)}\delta_{12} \right) =$$
$$= w_0^{12} + \frac{1}{(n+1)}\delta_{12} + \sum_{i=1}^{n} w_i^{12}Y_{i1} + \sum_{i=1}^{n} w_i^{12}\delta_{i1} + \frac{n}{(n+1)}\delta_{12} =$$
$$= \underbrace{w_0^{12} + \sum_{i=1}^{n} w_i^{12}Y_{i1}}_{z_{12}} + \delta_{12} + \sum_{i=1}^{n} w_i^{12}\delta_{i1} = z_{12} + \delta_{12} + \sum_{i=1}^{n} w_i^{12}\delta_{i1}.$$

Thus finally,

$$\tilde{z}_{12} = z_{12} + \delta_{12} + \sum_{i=1}^{n} w_i^{12}\delta_{i1} \qquad (15)$$

To ensure that after the correction procedure the output of the network will be exactly equal to $z_{12} + \delta^*$, it is clear from (15) that we need to satisfy the following:

$$\delta_{12} + \sum_{i=1}^{n} w_i^{12}\delta_{i1} = \delta^*. \qquad (16)$$

Of course, if we consider (16) as a formal equation, we will not get something useful. This equation has an infinite number of solutions, while we have to find among them a single solution, which will correctly represent the local errors of each neuron through the global error of the network and through the errors of the neurons of the preceding layer.

Let us come back to the learning rule (6) for the single MVN. In this rule $\Delta W = \frac{C_m}{(n+1)}\left(\varepsilon^q - \frac{z}{|z|}\right)\overline{X}$. It contains a factor $\frac{1}{(n+1)}$ in order to distribute the error $\varepsilon^q - \frac{z}{|z|}$ uniformly among all n+1 weights $w_0, w_1, ..., w_n$. It is easy to see that if we would omit this factor then the corrected weighted sum would not be equal to $z+\delta$ as expected, (see (4)), but to $z+(n+1)\delta$. On the other hand, since all the inputs are equitable, it will be correct and natural that during the correction procedure the error and therefore ΔW is distributed among the weights uniformly, so it must be shared among the weights. This makes clear the important role of the factor $\frac{1}{(n+1)}$ in (6).

If we have not a single neuron, but a feedforward network, we have to take into account the same property. This means that *the global error of the network consists not only of the output neuron error, but of the local errors of the output neuron and hidden neurons connected to the output neuron*. It means that to obtain the local errors for all neurons, the global error *must be shared among these neurons*. We can assume that *this sharing is uniform*: the contribution of each neuron to the global error is the same. This assumption is, of course heuristic, but it will be shown that it works, and it will be confirmed by the further considerations and the simulation results. Thus, we obtain from (16) the following:

$$\delta_{12} = \frac{1}{(n+1)}\delta^*, \tag{17}$$

$$w_i^{12}\delta_{i1} = \frac{1}{(n+1)}\delta^*; i = 1, ..., n. \tag{18}$$

But from (17) $\delta^* = (n+1)\delta_{12}$ and substituting this expression to (18), we obtain:

$$w_i^{12}\delta_{i1} = \delta_{12}; i = 1, ..., n .$$

Hence for the error δ_{i1} of the neuron $i1$ we obtain the following:

$$\delta_{i1} = \left(w_i^{12}\right)^{-1}\delta_{12}; i = 1, ..., n . \qquad (19)$$

Let us now substitute $\delta_{i1}; j = 1, ..., n$ and δ_{12} from (18) and (17), respectively, into (15):

$$\begin{aligned}\widetilde{z}_{12} &= z_{12} + \delta_{12} + \sum_{i=1}^{n} w_i^{12}\delta_{j1} = z_{12} + \delta_{12} + \sum_{i=1}^{n} w_i^{12}\left(w_i^{12}\right)^{-1}\delta_{12} = \\ &= z_{12} + \delta_{12} + n\delta_{12} = z_{12} + (n+1)\delta_{12} = z_{12} + \delta^*.\end{aligned} \qquad (20)$$

So, we obtain exactly the result that is our target: $\widetilde{z}_{12} = z_{12} + \delta^*$. It is important to mention that (20) corresponds with (4), where the same expression was obtained for the single MVN.

It follows from (17) that if in general the error of a neuron on the layer *j* is equal to $\widetilde{\delta}$, this $\widetilde{\delta}$ must contain a factor equal to $\frac{1}{s_j}$, where $s_j = N_{j-1} + 1$ is the number of neurons whose outputs are connected to the inputs of the considered neuron (let us remind that N_j is the number of neurons on the layer *j*, and all of these neurons are connected to the considered neuron) incremented by 1 (the considered neuron itself). This ensures sharing of the error among all the neurons on whose errors the error of the considered neuron depends. In other words, *the error of each neuron is distributed among the neurons connected to it and itself.* It should be mentioned that for the 1st hidden layer $s_1 = 1$ because there is no previous hidden layer, and there are no neurons, with which the error may be shared, respectively.

Eq. (19) is not only a formal expression for $\delta_{i1}; i = 1, ..., n$, but it leads us to the following important conclusion: *during a backpropagation procedure the backpropagated error must be multiplied by the inverse values of the corresponding weights.*[3]

Now it will not be difficult to generalize everything that we considered for the network with one hidden layer and a single output neuron (Fig. 4) to the network with an arbitrary number of layers and an arbitrary number of neurons in each layer. We will obtain the following. The global errors of the whole network are determined by (9). For the errors of the mth (output) layer neurons:

$$\delta_{km} = \frac{1}{s_m}\delta^*_{km}, \tag{21}$$

where *km* specifies a *k*th neuron of the *m*th layer; $s_m = N_{m-1} + 1$ (the number of all neurons on the previous layer (*m*-1, to which the error is backpropagated) incremented by 1).
For the errors of the hidden layers neurons:

$$\delta_{kj} = \frac{1}{s_j}\sum_{i=1}^{N_{j+1}} \delta_{i,j+1}(w_k^{i,j+1})^{-1}, \tag{22}$$

where *kj* specifies a *k*th neuron of the *j*th layer (*j*=1,…,*m*-1); $s_j = N_{j-1} + 1, \quad j = 2,...,m; \; s_1 = 1$ (the number of all neurons on the previous layer (previous to *j*, to which the error is backpropagated) incremented by 1). It should be mentioned that the backpropagation rule (22) is based on the same heuristic assumption as for the classical backpropagation. According to this assumption we suppose that the error of each neuron from the previous (*j*th) layer depends on the errors of all neurons from the following (*j*+1st) layer.

Thus, the equations (21)-(22) determine the error backpropagation for the MLMVN. It is important to mention the sharp distinctions of these equations from the ones (13)-(14) that describe the error backpropagation for a MLF. The equations (21)-(22) do not contain a derivative of the activation function. The next important distinction is that for the MLMVN the errors of the neurons

[3] Do not forget that our weights are complex numbers and therefore for any weight $w^{-1} = \frac{\overline{w}}{|w|^2}$

at a preceding layer are formed from the inversely weighted summations of the following layer errors, while for a MLF they are formed from the weighted summations of the following layer errors.

After calculation of the errors, the weights for all neurons of the network must be corrected. To do it, we can use the learning rule (6) applying it sequentially to all layers of the network from the first hidden layer to the output one. The rule (6) can be rewritten for this case in the following form: Correction rule for the neurons from the 2nd hidden layer till the *m*th (output) layer (*k*th neuron of the *j*th layer (*j*=2, …, *m*)):

$$\begin{aligned} \tilde{w}_i^{kj} &= w_i^{kj} + \frac{C_{kj}}{\left(N_{j-1}+1\right)}\delta_{kj}\bar{\tilde{Y}}_{i,j-1}, \quad i=1,\ldots,n \\ \tilde{w}_0^{kj} &= w_0^{kj} + \frac{C_{kj}}{\left(N_{j-1}+1\right)}\delta_{kj} \end{aligned} \tag{23}$$

Correction rule for the neurons from the 1st hidden layer:

$$\begin{aligned} \tilde{w}_i^{k1} &= w_i^{k1} + \frac{C_{k1}}{(n+1)}\delta_{k1}\bar{x}_i, \quad i=1,\ldots,n \\ \tilde{w}_0^{k1} &= w_0^{k1} + \frac{C_{k1}}{(n+1)}\delta_{k1} \end{aligned} \tag{24}$$

where C_{kj} is a learning rate for the *k*th neuron of *j*th layer and *n* is the number of the corresponding neuron inputs.

On the other hand, we have to take into account the following consideration. For the output layer neurons, we have the exact errors calculated according to (9), while for all the hidden neurons the errors are obtained according to the heuristic rule. This may cause a situation, where either the weighted sum for the hidden neurons (more exactly, the absolute value of the weighted sum) may become a not-smooth function with dramatically high jumps, or the hidden neuron output will be close to some constant with very small variations around it. In both cases thousands and even the hundreds of thousands of additional steps for the weights adjustment will be required. To avoid this situation, we can use for the hidden neurons the learning rule (7) instead of the rule (6) and therefore normalize ΔW for the hidden neurons by |*z*| every time, when the weights are being corrected. This will make the absolute value of the weighted sum for the hidden neurons (considered as a function of the weights) more smooth. This also can avoid the concentration of the hidden neurons output in some very narrow interval. On the other hand, the factor 1/|*z*| in (7) can be considered as a variable part of the learning rate. While used, it provides the adaptation of ΔW on each step of learning. At the same time, it is not reasonable to use the rule (7) for the output layer. The exact errors and the exact desired outputs for the output neurons are known. On the other hand, since these errors are shared among all neurons of the network according to (21)-(22), there is no reason to normalize by 1/|*z*| the errors of the output neurons. The absolute value of the output neurons weighted sums belongs to the narrow ring, which includes the unit circle (see Fig. 3), without this normalization. We will return to these features of the learning algorithm in the Section IV.A, where the corresponding example will be considered in detail. So the weights of the output neurons may be corrected by the rule (6). In this way we are coming to the following modification of the correction rule (23)-(24). Correction rule for the neurons from the *m*th (output) layer (*k*th neuron of *m*th layer):

$$\begin{aligned}\tilde{w}_i^{km} &= w_i^{km} + \frac{C_{km}}{(N_{m-1}+1)}\delta_{km}\bar{\tilde{Y}}_{i,m-1}, \quad i=1,\dots,n\\ \tilde{w}_0^{km} &= w_0^{km} + \frac{C_{km}}{(N_{m-1}+1)}\delta_{km}\end{aligned} \tag{25}$$

Correction rule for the neurons from the 2nd till *m*-1st layer (*k*th neuron of the *j*th layer (*j*=2, …, *m*-1):

$$\begin{aligned}\tilde{w}_i^{kj} &= w_i^{kj} + \frac{C_{kj}}{(N_{j-1}+1)\,|z_{kj}|}\delta_{kj}\bar{\tilde{Y}}_{i,j-1}, \quad i=1,\dots,n\\ \tilde{w}_0^{kj} &= w_0^{kj} + \frac{C_{kj}}{(N_{j-1}+1)\,|z_{kj}|}\delta_{kj}\end{aligned} \tag{26}$$

Correction rule for the neurons from the 1st hidden layer:

$$\tilde{w}_i^{k1} = w_i^{k1} + \frac{C_{k1}}{(n+1)\,|\,z_{kj}\,|}\delta_{k1}\bar{x}_i, \quad i = 1,...,n$$
$$\tilde{w}_0^{k1} = w_0^{k1} + \frac{C_{k1}}{(n+1)\,|\,z_{kj}\,|}\delta_{k1} \tag{27}$$

All the considerations above lead us to the conclusion that the errors of the output layer neurons and therefore the global errors of the network will descent after correction of the weights according to the rules (25)-(27). It means that the square error (10) and the error functional (11) will also descent step by step. In general, the learning process should continue until the following condition is satisfied

$$\frac{1}{N}\sum_{s=1}^{N}\sum_{k}(\delta_{km\,s}^{*})^2(W) = \frac{1}{N}\sum_{s=1}^{N}E_s \le \lambda\,, \tag{28}$$

where λ determines the precision of learning. The convergence of the presented training algorithm follows from the same considerations as the convergence of the learning algorithm with the rule (6) for the single neuron. As it was shown above, a problem of this convergence can be reduced to the convergence of the learning algorithm for the discrete-valued single MVN based on the rule (3), which is proven in [1].

IV. Simulation Results

When simulating the MLMVN we try to operate with a network with a minimal possible number of the hidden layers and with a minimal possible number of the neurons. So the simulation results that we will consider now, were obtained using the simplest possible network structure $n \rightarrow S \rightarrow 1$ (n inputs, S neurons on the 1st hidden layer and 1 neuron on the output layer). It should be mentioned that by including additional hidden layers we did not get any significant improvement. So the efficiency of the backpropagation algorithm and of the learning algorithm based on the rules (25)-(27) has been tested by experiments with four standard and popular benchmarks: parity *n*, two spirals, "sonar" and the Mackey-Glass time series. Data for the two spirals and "sonar" benchmarks was downloaded from the CMU benchmark collection[4]

The neural network was implemented using a software simulator that was run on a PC with a Pentium III 600 MHz processor. The software was developed on the Borland/Inprise Delphi 5.0 platform. The real and imaginary parts of the starting weights were taken as random numbers from the interval [0, 1] for all experiments. For the continuous-valued benchmarks (inputs for two spirals and sonar benchmarks and inputs/outputs for the Mackey Glass time series) the corresponding data were rescaled to the range $[0, 2\pi]$ in order to represent all inputs/outputs as points on the unit circle.

A. Parity N function

The parity *n* function is a mod 2 sum of *n* Boolean variables. The "parity *n* problem" is a problem of the implementation of the parity *n* function, and it is commonly and widely used for testing of the feedforward neural networks. The parity *n* functions $(3 \le n \le 9)$ were trained completely (see Table I) using the network $n \rightarrow S \rightarrow 1$, where $S < n$. It should be mentioned that parity 9 and parity 8 functions were implemented using only 7 and 4 hidden neurons, respectively. These results show advantages of MLMVN in comparison with the traditional solutions. Indeed, it was reported in [22] that the most optimistic estimation for the number of the hidden neurons for the implementation of the *n* bit parity function using one hidden layer is $\sqrt{n}$, (which is true for n ≤ 4), meanwhile the realistic estimation is Ό(*n*). In [14] it was shown up to $n = 4$, that the minimum size of the hidden layer required to solve the *n*-bit parity is *n*. It was theoretically estimated in [31] that the parity *n* function could be implemented using MLF with only $\frac{n+1}{2}$ hidden neurons for *n* odd. However, this estimation was experimentally confirmed in [32] and [31] for maximum *n*=7. The parity 7 function was implemented using a 7-4-1 MLF in [32] and [31] using a special learning algorithm based on the careful choice of learning rates on each epoch with special attention to hidden node saturation. The corresponding learning algorithm is computationally complicated because it requires

[4] http://www-2.cs.cmu.edu/afs/cs/project/ai-repository/ai/areas/neural/bench/cmu (a number of references to the different experimental results and the summary of these results can be also found at the same directory)

the computation of a 128 x 37 Jacobian matrix on each of about 18000-19000 epochs required for the training process [32]. The parity 7 function implementation on the 7-4-1 MLMVN is computationally much more effective. It formally requires about 20000-30000 epochs, but an amount of computations for the MLMVN backpropagation according to (21)-(22) and (25)-(27) is incomparably smaller, and as a result the MLMVN implementation requires a very short time (see Table I). Using modular network architecture, the parity 8 function was implemented in [12]. The corresponding modular network consists of 7 hidden layers and totally of 28 neurons with 42 synaptic weights. The 8-4-1 MLMVN, on which the parity 8 function is implemented, consists only of 5 neurons with 41 synaptic weights. Notice that the modular network [12] was specially developed only for solving the parity problem, while the MLMVN has a standard feedforward architecture, which does not depend on problem to be solved. The parity 9 function was implemented on the 9-7-1 MLMVN, and we did not find any competitive solution, where this function was implemented using a MLF with less than 9 hidden neurons. Solving the parity *n* problem we did use neither some special architecture nor some specific learning strategy.

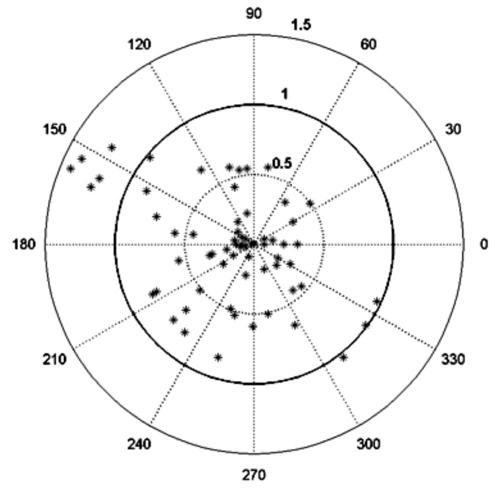
Table I Implementation of the Parity *n* function ($n = 2,3,...,9$) on the MLMVN $S \rightarrow 1$ ($S < n$)

Function	Configuration of the network	Number of epochs (the median value of 5 independent experiments is taken)	Processing time on P-III 600 MHz
Parity 3	3→2→1	57	2 seconds
Parity 4	4→2→1	109	3 seconds
Parity 5	5→3→1	2536	15 seconds
Parity 6	6→4→1	7235	30 seconds
Parity 6	6→3→1	159199	5 min. 20 sec.
Parity 7	7→4→1	26243	2 min. 50 sec.
Parity 8	8→4→1	164747	15 min.
Parity 9	9→7→1	24234	20 min.

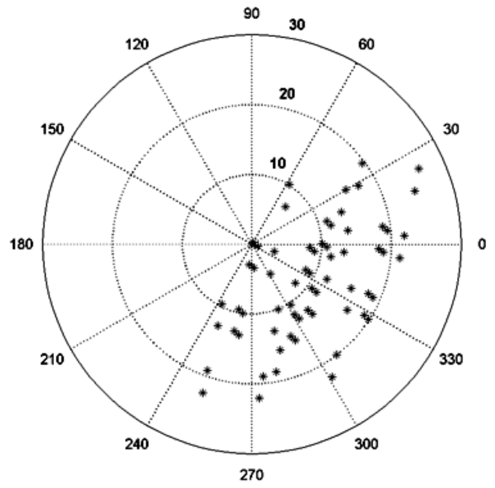
Moreover, no adaptation of the constant part of the learning rate was used in all our experiments not only with the parity functions, but with all the benchmarks, i.e. all $C_{kj} = 1$ in (25)-(27). It is important to mention that our experiments show that at least up to *n*=8 the parity *n* problem may be solved on the MLMVN with a single hidden layer and at most $(n+1)/2$ hidden neurons in this layer. It means that the MLMVN to the best knowledge of the authors overcomes all other known solutions for $n \le 9$.

Let us return to the role of the factor $\frac{1}{z_{kj}}$ in the correction rule (26)-(27) for the hidden neurons and to the absence of this factor in the correction rule (25) for the output neurons. Without loss of generality we can consider the example of parity 6 function. Let us consider its implementation using the network 6→4→1 (four hidden neurons and one output neuron, see Table I). Training this network, we applied the rule (23)-(24) and (25)-(27) starting from the same randomly chosen weighting vector. With the rule (25)-(27) we get the convergence after 5874 epochs. The distribution of the weighted sums after the convergence is shown for the output neuron and one of the hidden neurons on Fig. 5a and Fig. 5b, respectively. It is clearly visible from Fig. 5a that the weighted sums for the output neuron belong to the narrow ring, whose longer border is limited by the circle of a radius 1.5. For the first hidden neuron (Fig. 5b) the weighted sums are distributed in such a way that the neuron's output belongs to the wide interval approximately equal to the half of the unit circle. Actually, exactly this is our target: we introduce the hidden layer in order to replace a Boolean nonlinearly separable function (parity *n*) by the multiple-valued function with a binary output, however with informally multiple-valued inputs. With the rule (23)-(24), which does not contain the factor $\frac{1}{z_{kj}}$ as a variable part of the learning rate for the hidden neurons, we cannot get the convergence even after 20000 epochs with the same starting weighting vector. Fig. 5c shows that after 10000 epochs the weighted sums for the first hidden neuron are concentrated within a narrow domain, which looks as a stripe. As a result, the neuron's output belongs to a very limited domain. It is not binary, but it is not enough multiple-valued. Projections of too many weighted sums on the unit circle coincide with each other. There are practically no changes after additional

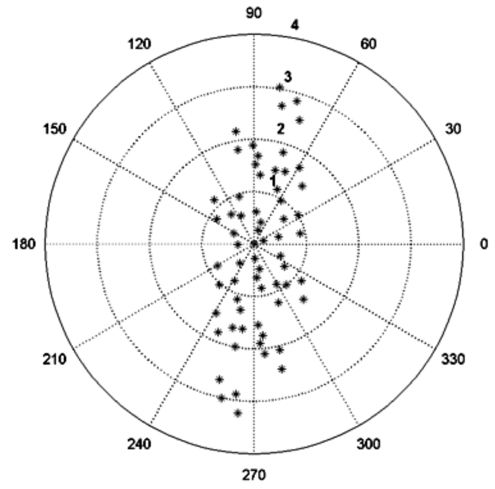
10000 epochs (Fig. 5d). This example is typical. It shows that the use of the factor $\frac{1}{z_{kj}}$ as a variable part of the learning rate is very important for the hidden neurons. It makes the learning rate self-adaptive to the particular case.



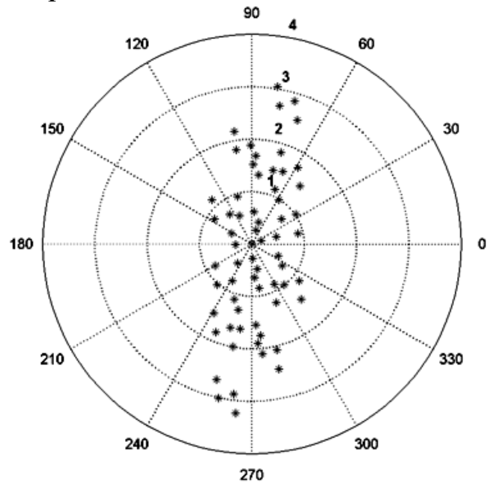
(a) output neuron after convergence of the learning algorithm. The weighted sums belong to the ring, whose longer border is limited by the circle of a radius 1.5



(b) the first neuron from the hidden layer after convergence of the learning algorithm (the rule (27) was used, 5874 epochs). The weighted sums are distributed in such a way that the neuron's output belongs to the wide interval approximately equal to the half of the unit circle



(c) the first neuron from the hidden layer after 10000 epochs without convergence using the rule (24). The weighted sums are distributed within a narrow domain and the neuron's output also belongs to a quite limited domain



(d) the first neuron from the hidden layer after 20000 epochs without convergence using the rule (24). Almost no changes in comparison with the picture (b) after additional 10000 epochs.

Fig. 5 Distribution of the weighted sums for the output neuron and one of the hidden neurons for the network 6→4→1 implementing the parity 6 function.
64 weighted sums are shown as points on the complex plane

B. Two spirals problem

The two spirals problem is a well known classification problem, where the two spirals points (see Fig. 6) must be classified as belonging to the 1st or to the 2nd spiral. The two spirals data also was trained completely without errors using the networks 2→40→1 (800123 epochs is a median of 11 experiments) and 2→30→1 (1590005 epochs is a median of 11 experiments). For example, for MLF with an adapted learning algorithm there is the result with about 4% errors for the network 2→40→1 and with about 14% errors for the network 2→30→1 after about 150000 learning epochs [32]. It should be mentioned that after the same number of learning epochs the MLMVN shows not more than 2% of errors for the network 2→40→1 and not more than 6% errors for the network 2→30→1.

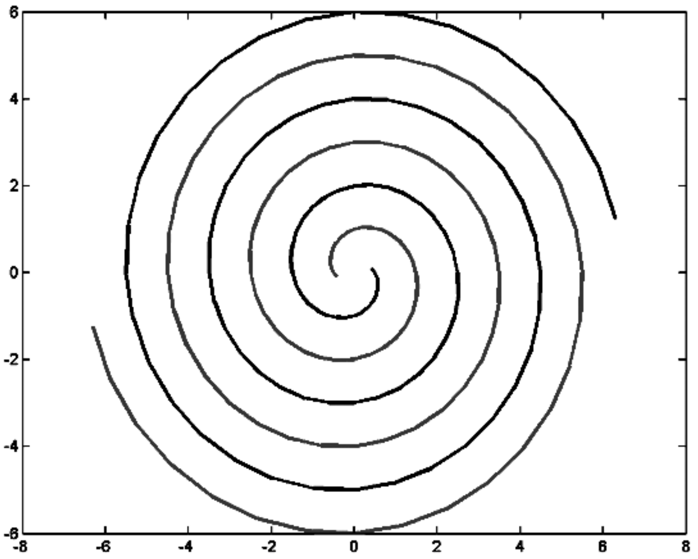


Fig. 6 Two spirals

Table II Correspondence between the number of hidden neurons and the number of training epochs

# of epochs (median for 9 independent experiments)	1298406	1015315	516807	218004	98732	84338	31145	19781
# of epochs (min and max for 9 independent experiments)	1010877-1412355	881017-1414385	375289-1467516	157278-587619	60115-159784	51914-138137	18385-130853	3866-129291
# of neurons on the hidden layer	26	28	30	32	34	36	38	40

In order to compare our results with others available in the literature, we also evaluated the "prediction" capability of the network using a cross-validation by taking alternative pairs of consecutive points as training and test data. The two spirals data set containing 194 points was separated into training (98 points) and testing (96 points) subsets (the first two points were assigned to the training subset, while the next two points were assigned to the testing subset, etc). After training using the first subset, the prediction capability was tested using the second one. For this experiment we used the networks containing 26, 28, …, 40 hidden neurons on the single hidden layer. The networks with the larger number of the hidden neurons are trained much faster, but the prediction results are approximately the same for all the networks. Table II shows, how the number of training epochs decreases with increasing of the number of the hidden neurons. The median, minimal and maximal numbers of the epochs for nine experiments with each network are shown. The prediction results for the two spiral benchmark are stable for all considered networks from $2\rightarrow 26\rightarrow 1$ till $2\rightarrow 40\rightarrow 1$. The predictions rate for the two spirals testing set obtained as a result of nine independent experiments with each network is 68-72%. These results that were obtained using a smaller and simpler network are comparative with the best known results (70-74.5%) obtained using the fuzzy kernel perceptron (FKP) [9] whose training is computationally more complicated than the one for the MLMVN. The FKP is oriented to solving the "2-classes" problems, while the MLMVN can solve both multiple-valued and continuous-valued problems. It should also be mentioned that some additional series of the experiments show that it is possible to obtain even a better prediction rate (74-75%) for the MLMVN. This result appeared very randomly (2 times per

100 independent experiments with the 2→40→1 network), so it can not be recognized as a stable result, but it is an additional argument for the high potential capabilities of MLMVN.

C. *Sonar benchmark*

The same experiment was performed for the "sonar" data set, but using the simplest possible network 60→2→1 (the "sonar" problem initially depends on 60 input parameters) with only two neurons in the hidden layer. The "sonar" data set contains 208 samples. 104 of them are recommended to be used for training and the other 104 for testing, respectively. The training process requires 400-2400 epochs and a few seconds, respectively. This statistics is based on 50 independent experiments. The prediction results are also very stable. The prediction rate from the same experiments is 88-93%. This result is comparative with the best known result for the fuzzy kernel perceptron (FKP) (94%) [9] and SVM (89.5%) [9]. However, our result is obtained using the smallest possible network containing only 3 neurons, while for solving the sonar problem FKP and SVM require in average 167 and 82.6 supporting vectors, respectively [9]. On the other hand the whole "sonar" data set was trained completely without errors using the same simplest network 60→2→1. This training process requires from 817 till 3700 epochs according to the results of 50 independent runs.

The best prediction rate reported for the classical MLF with two hidden neurons is 85.7% [16] To increase it up to 89.2%, a MLF must contain 24 hidden neurons [16].

D. *Mackey-Glass time series prediction*

To test the MLMVN capabilities in time series prediction, we used the well known Mackey-Glass time series. This time series is generated by the chaotic Mackey-Glass differential delay equation defined as follows [30]:

$$\frac{dx(t)}{dt}=\frac{0.2x(t-\tau)}{1+x^{10}(t-\tau)}-0.1x(t)+n(t), \tag{29}$$

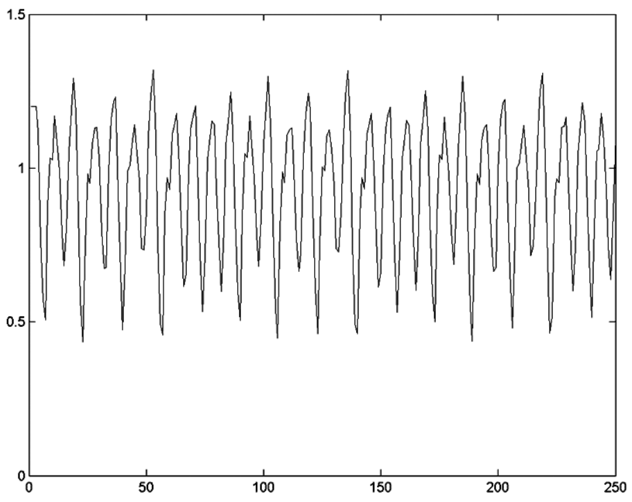
where $n(t)$ is a uniform noise (it is possible that $n(t)=0$). $x(t)$ is quasi-periodic, and chosing $\tau=17$ it becomes chaotic [30], [11]. This means that only short term forecasts are feasible. Exactly $\tau=17$

was used in our experiment. To integrate the equation (29) and to generate the data, we used an initial condition $x(0) = 1.2$ and a time step $\Delta t = 1$. The Runge-Kutta method was used for the integration of the equation (29). The data was sampled every 6 points, as it is usually recommended for the Mackey Glass time-series (see e.g., [36], [23]). Thus, we use the Mackey-Glass time series generated with the same parameters and in the same way as in the recently published papers [36], [23], [26]. The task of prediction is to predict $x(t+6)$ from $x(t), x(t-6), x(t-12), x(t-18)$. We generated 1000 points data set. The first 500 points were used for training (a fragment of the first 250 points is shown in Fig. 7a) and the next 500 points were used for testing (a fragment of the last 250 points is shown in Fig. 7b). The true values of $x(t+6)$ were used as the target values during training.

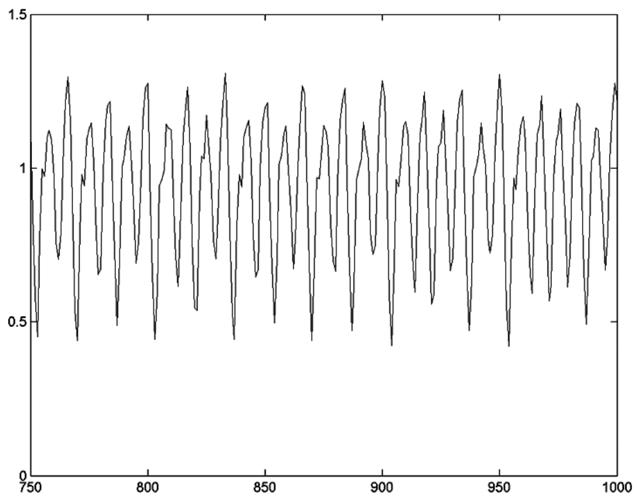
Since the *root mean square error* (RMSE) is a usual estimation of the quality for the Mackey-Glass time series prediction [11], [23], [24], [26], [36] we also used it here. We did not require a convergence of the training algorithm to the zero error. Since RMSE is a usual estimator for the prediction quality, it also was used for the training control. Thus instead of the MSE criterion (28), we used the following RMSE criterion for the convergence of the training algorithm:

$$\sqrt{\frac{1}{N}\sum_{s=1}^{N}\sum_{k}(\delta^{*}_{km_s})^2(W)} = \sqrt{\frac{1}{N}\sum_{s=1}^{N}E_s} \le \lambda\,, \tag{30}$$

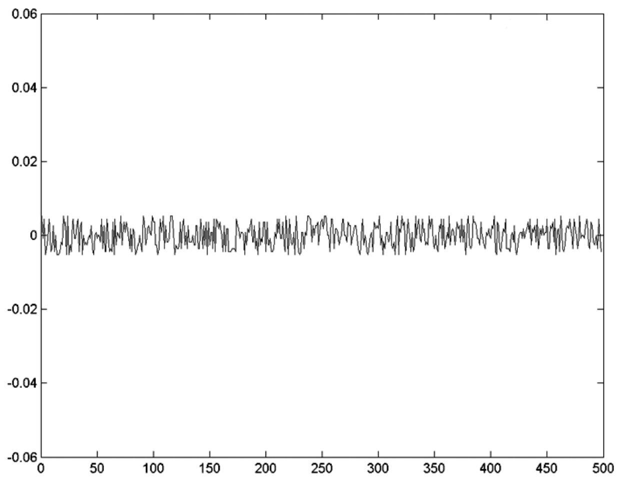
where λ determines a maximum possible RMSE for the training data.



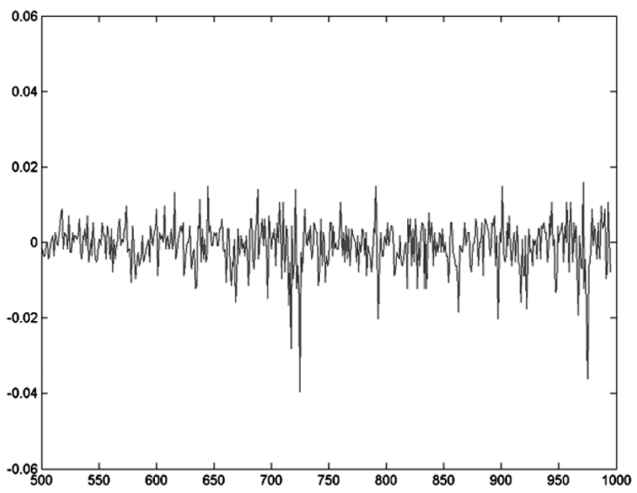
(a) Mackey-Glass time series: a fragment of the first 250 points of the training data



(b) Mackey-Glass time series: a fragment of the last 250 points of the testing data



(c) training error (RMSE=0.0032)



(d) testing error (RMSE=0.0063)

Fig. 7 Mackey-Glass time series prediction

The results of our experiments are summarized in Table III. For each of the three series of experiments we made 30 independent runs of training and prediction, like it was done in [23]. Our experiments show that choosing a smaller λ in (30) it is possible to decrease the RMSE for the testing data significantly. The results of training and prediction are illustrated in Fig. 7c and Fig. 7d, respectively. Since both the testing and prediction errors are very small and it is practically impossible to show the difference among the actual and predicted values at the same graph, Fig. 7c and Fig. 7d show not the actual and predicted data, but the error among them. Fig. 7c presents the error on the training set after the convergence of the training algorithm for the network containing 50 hidden neurons. $\varepsilon = 0.0035$ was used as a maximum possible RMSE in (30). An actual RMSE on the training set at the moment, when training was stopped, was equal to 0.0032. Fig. 7d presents the error on the testing set (RMSE=0.0063, which is a median value of the 30 independent experiments). To estimate a training time, one can base on the following data for the networks containing 50 and 40 hidden neurons, respectively. 100000 epochs require 50 minutes for the first network and 40 minutes for the second one on a PC with a Pentium-III 600 MHz CPU.

Table III The results of Mackey-Glass time series prediction using MLMVN

# of neurons on the hidden layer		**50**	**50**	**40**
λ - a maximum possible **RMSE** in (30)		0.0035	0.0056	0.0056
Actual RMSE for the training set (min - max)		0.0032 - 0.0035	0.0053 – 0.0056	0.0053 – 0.0056
RMSE for the **testing set**	Min	**0.0056**	0.0083	0.0086
	Max	**0.0083**	0.0101	0.0125
	Median	**0.0063**	0.0089	0.0097
	Average	**0.0066**	0.0089	0.0098
	SD	0.0009	0.0005	0.0011
Number of training epochs	Min	95381	24754	34406
	Max	272660	116690	137860
	Median	145137	56295	62056
	Average	162180	58903	70051

Comparing the results of the Mackey-Glass time series prediction using MLMVN to the results that were obtained during the last years using different solutions, we have to conclude that MLMVN outperforms all of them (see Table IV). The results comparative with the ones obtained using MLMVN are obtained only using GEFREX [38] and ANFIS [24]. On the other hand, it is not mentioned in [38] and [24] whether the reported RMSE is the result of averaging over the series of experiments or it is the best result of this series.

Table IV Comparison of Mackey-Glass time series prediction using MLMVN with other models

MLMVN min	**MLMVN average**	GEFREX [38]	EPNet [41]	ANFIS [24]	CNNE [23]	SuPFuNIS [36]	Classical Backprop. NN (taken from [28])
0.0056	**0.0066**	0.0061	0.02	0.0074	0.009	0.014	0.02

In any case MLMVN with the 50 hidden neurons steadily outperforms ANFIS and shows at least a comparative result with GEFREX. However, the implementation of MLMVN is incompatibly simpler than the one of GEFREX. (for example, referring to the difficulty of the GEFREX implementation, the SuPFuNIS model was proposed in [36]). It is important to mention that the classical feedforward neural network with a backpropagation learning [28] loses to MLMVN dramatically. It is interesting that the best results of our predecessors were obtained using different fuzzy techniques [36], [38], [24]. The only pure neural solution, which shows a comparative result (which is not so good as the one obtained using MLMVN) is the recently proposed CNNE [23]. This is an ensemble of several feedforward neural networks. Its average result obtained with a greater number of the hidden neurons (56 in average) is worse than the one obtained using MLMVN with 50 hidden neurons. It should also be mentioned that a training of the networks ensemble (CNNE) is more complicated than the one of the single network. The additional important advantage of MLMVN is an opportunity to control a level of the prediction error by choosing an appropriate value of the maximum training error λ in (30).

V. Conclusions

In this paper a new feedforward neural network was proposed. A multilayer neural network based on multi-valued neurons (MLMVN) is a network with a traditional feedforward architecture and a multi-valued neuron (MVN) as a basic one. A single MVN has the higher functionality than the traditional neurons. Its training, which is reduced to the movement along the unit circle, does not require the differentiability of the activation function. The learning process is completely determined by a simple linear rule. These properties make MLMVN more powerful than traditional feedforward networks. The backpropagation learning algorithm for MLMVN that was developed in the paper also does not require the differentiability of the activation function. Being similar to the classical backpropagation algorithm, it has some important differences that make it more stable and less heuristic. The main differences are: sharing of the network error among all the neurons of the network and a self adaptation of the learning rate for the hidden neurons.

These advantages of the MLMVN make it possible to solve different prediction, recognition and classification problems using a smaller network and a simpler learning algorithm than ones traditionally used. MLMVN shows a good performance in convergence speed. Its testing using several traditional benchmarks shows better or comparative performance compared to the best known results.

ACKNOWLEDGEMENT

The first author gladly acknowledges the support of the Collaborative Research Center 531 of the University of Dortmund, where he was an Invited Researcher, January through April 2004.

REFERENCES

[1] Aizenberg I., Aizenberg N. and Vandewalle J. (2000) Multi-valued and universal binary neurons: theory, learning, applications. Kluwer Academic Publishers, Boston Dordrecht London.
[2] Aizenberg I., Bregin T., Butakoff C., Karnaukhov V., Merzlyakov N. and Milukova O. (2002) Type of Blur and Blur Parameters Identification Using Neural Network and Its Application to Image Restoration. In Dorronsoro JR (ed) Lecture Notes in Computer Science, 2415, Springer, Berlin Heidelberg New York, pp 1231-1236,.
[3] Aizenberg I., Myasnikova E., Samsonova M. and Reinitz J. (2002) Temporal Classification of Drosophila Segmentation Gene Expression Patterns by the Multi-Valued Neural Recognition Method. Journal of Mathematical Biosciences 176 (1): 145-159.
[4] Aizenberg N.N., Ivaskiv Yu. L. and Pospelov D.A. (1971) About one Generalization of the Threshold Function Doklady Akademii Nauk SSSR (The Reports of the Academy of Sciences of the USSR), 196: 1287-1290, (in Russian).
[5] Aizenberg N.N. and Ivaskiv Yu.L. (1977) Multiple-Valued Threshold Logic. Naukova Dumka Publisher House, Kiev (in Russian).
[6] Aizenberg N.N. and Aizenberg I.N. (1992) CNN Based on Multi-Valued Neuron as a Model of Associative Memory for Gray-Scale Images, In Proceedings of the Second IEEE Int. Workshop on Cellular Neural Networks and their Applications, Technical University Munich, Germany October 14-16, 1992, pp.36-41.
[7] Aoki H. and Kosugi Y. (2000) An Image Storage System Using Complex-Valued Associative Memory, In Proceedings. of the 15th International Conference on Pattern Recognition. IEEE Computer Society Press, vol. 2, pp. 626-629.
[8] Aoki H., Watanabe E., Nagata A. and Kosugi Y. (2001) Rotation-Invariant Image Association for Endoscopic Positional Identification Using Complex-Valued Associative Memories. In Mira J., Prieto A.(eds) Bio-inspired Applications of Connectionism, Lecture Notes in Computer Science 2085 Springer, Berlin Heidelberg New York, pp. 369-374.
[9] Chen J.-H. and Chen C.-S. (2002) Fuzzy Kernel Perceptron. IEEE Transactions on Neural Networks 13: 1364-1373
[10] Cover T.M. (1965) Geometrical and Statistical Properties of systems of Linear Inequalities with application in pattern recognition. IEEE Trans. Electron. Comput. 14: 326-334.

[11] Fahlman J.D. and Lebiere C. (1987) Predicting the Mackey-Glass time series. Physical Review Letters 59: 845-847.
[12] Franco L. and Cannas S.A. (2001) Generalization Properties of Modular Networks: Implementing the Parity Function, IEEE Transactions on Neural Networks 12: 1306-1313.
[13] Funahashi K.I. (1989) On the Approximate Realization of Continuous Mappings by Neural Networks. Neural Networks 2: 183-192.
[14] Fung H. and Li L. K. (2001) Minimal feedforward parity networks using threshold gates. Neural Computing 13: 319–326.
[15] Georgiou G.M. and Koutsougeras C. (1992) Complex Domain Backpropagation IEEE Transactions on Circuits and Systems CAS-II 39: 330-334.
[16] Gorman R. P., , and Sejnowski T. J. (1988) Analysis of Hidden Units in a Layered Network Trained to Classify Sonar Targets, Neural Networks 1: 75-89.
[17] Haykin S. (1999) Neural Networks: A Comprehensive Foundation, 2nd edn. Prentice Hall, New Jersey.
[18] Hecht-Nielsen R. (1988) Kolmogorov Mapping Neural Network Existence Theorem. In Proceedings of the 1st IEEE International Conference on Neural Networks, IEEE Computer Society Press, vol. 3, pp. 11-13.
[19] Hecht-Nielsen R. (1990) Neurocomputing. Adison Wesley, New York.
[20] Hirose A. (ed) (2003) Complex Valued Neural Networks. Theories and Applications. World Scientific, Singapore.
[21] Hornik K., Stinchcombe M., White H.. (1989) "Multilayer Feedforward Neural Networks are Universal Approximators", Neural Networks 2: 259-366.
[22] Impagliazzo R., Paturi R., and Saks M. E. (1997) Size-depth tradeoffs for threshold circuits. SIAM Journal of Computing 26: 693–707.
[23] Islam M.M., Xin Yao and Murase K. (2003) A Constructive Algorithm for Training Cooperative Neural Networks Ensembles. IEEE Transactions on Neural Networks 14: 820-834.
[24] Jang J.-S. R. (1993) ANFIS: Adaptive-Network-Based Fuzzy Inference System. IEEE Transactions on Systems, Man and Cybernetics 23: 665–685.
[25] Jankowski S., Lozowski A. and Zurada J.M. (1996) Complex-Valued Multistate Neural Associative Memory. IEEE Transactions on Neural Networks 7: 1491-1496.
[26] Kim D. and Kim C. (1997) Forecasting Time Series with Genetic Fuzzy Predictor Ensemble. IEEE Transactions on Neural Networks 5: 523-535.
[27] Kolmogorov A.N. (1957) On the Representation of Continuous Functions of many Variables by Superposition of Continuous Functions and Addition. Doklady Akademii Nauk SSSR (The Reports of the Academy of Sciences of the USSR) 114: 953-956 (in Russian).
[28] Lee S.-H. and Kim I. (1994) Time Series Analysis using Fuzzy Learning. In Proceedings of the International Conference on Neural Information Processing, Seoul, Korea. Vol. 6, pp. 1577–1582.
[29] Leung H. and Haykin S. (1991) The Complex Backpropagation Algorithm. IEEE Transactions on Signal Processing 39: 2101-2104.
[30] Mackey M.C. and Glass L. (1977) Oscillation and chaos in physiological control systems. Science 197: 287-289.
[31] Mizutani E., Dreyfus S.E., and Jang J.-S. R. (2000) On dynamic programming-like recursive gradient formula for alleviating hidden-node saturation in the parity problem. In Proceedings of the International Workshop on Intelligent Systems Resolutions – the 8th Bellman Continuum, Hsinchu, Taiwan, pp. 100-104.
[32] Mizutani E. and Dreyfus S.E. (2002) MLP's hidden-node saturations and insensitivity to initial weights in two classification benchmark problems: parity and two-spirals. In Proceedings of the 2002 International Joint Conference on Neural Networks (IJCNN'02), pp. 2831-2836.
[33] Muezzinoglu M. K., Guzelis C. and Zurada J. M. (2003) A New Design Method for the Complex-Valued Multistate Hopfield Associative Memory. IEEE Transactions on Neural Networks 14: 891-899.

[34] Müller K.-R., Mika S., Rätsch G., Tsuda K. and Shölkopf B. (2001) An Introduction to Kernel-based Learning Algorithms. IEEE Transactions on Neural Networks 12: 181-201.
[35] Nitta T. (1997) An extension of the Backpropagation Algorithm to Complex Numbers. Neural Networks 10: 1391-1415.
[36] Paul S. and Kumar S. (2002) Subsethood-Product Fuzzy Neural Inference System (SuPFuNIS), IEEE Transactions on Neural Networks 13: 578-599.
[37] Rumelhart D.E. and McClelland J.L. (1986) Parallel Distributed Processing: Explorations in the Microstructure of Cognition. MIT Press, Cambridge.
[38] Russo M. (2000) Genetic Fuzzy Learning. IEEE Transactions on Evolutionary Computation 4: 259-273.
[39] Siegelman H., Sontag E.. Neural Nets are Universal Computing Devices. *Research* Report SYCON-91-08. Rutgers Center for Systems and Control. Rutgers University, 1991.
[40] Vapnik V. (1995) The Nature of Statistical Learning Theory. Springer, Berlin Heidelberg New York,.
[41] Yao X. and Liu Y. (1997) A new Evolutionary System for Evolving Artificial Neural Networks. IEEE Transactions on Neural Networks 8: 694–713.