Neuro-fuzzy Systems and Their Applications

Bogdan M. Wilamowski
University of Wyoming
Department of Electrical Engineering
Laramie WY 82071
wilam@uwyo.edu

Abstract - **Computational intelligence combines neural networks, fuzzy systems, and evolutional computing. Neural networks and fuzzy systems, have already proved their usefulness and have been found useful for many practical applications. We are at the beginning of the third technological revolution. Now neural networks are being capable of replacing highly skilled people with all their experience.**

The concept of artificial neural networks is presented, underlining their unique features and limitations. A review and comparison of various supervised and unsupervised learning algorithms follows. Several special, easy to train, architectures are shown. The neural network presentation is illustrated with many practical applications such as speaker identification, sound recognition of various equipment as a diagnosis tool, written character recognition, data compression using pulse coupled neural networks, time series prediction, etc.

In the later part of the presentation the concept of fuzzy systems, including the conventional Zadeh approach and Takagi-Sugano architecture, is presented. The basic building blocks of fuzzy systems are discussed. Comparisons of fuzzy and neural systems, are given and illustrated with several applications. The fuzzy system presentation is concluded with a description of the fabricated VLSI fuzzy chip.

I. INTRODUCTION

Fascination about artificial neural networks started when McCulloch and Pitts in 1943 developed their model of an elementary computing neuron and when Hebb in 1949 introduced his learning rules. A decade latter Rosenblatt introduced the *perceptron* concept. In the early sixties Widrow and Holf developed intelligent systems such as ADALINE and MADALINE. Nilson in his book "Learning Machines" [10] summarized many developments of that time. The publication of the Mynsky and Paper in 1969 wrote the book with some discouraging results and this stopped for sometime a fascination of artificial neural networks, and achievements in the mathematical foundation of the *backpropagation* algorithm by Werbos in 1974 went unnoticed. The current rapid growth of the neural network area started in 1982 with Hopfield recurrent network and Kohonen unsupervised training algorithms. The *backpropagation* algorithm described by Rumelhard in 1986 stared rapid development of neural networks.

II. NEURON

Biological neuron has a complicated structure which receives trains of pulses on hundreds of *excitatory* and *inhibitory* inputs. Those incoming pulses are summed and averaged with different weights during the time period of *latent summation*. If the summed value is higher than a threshold then the neuron generates a pulse which is sent to neighboring neurons. If the value of the summed weighted inputs is higher, the neuron generates pulses more frequently.

An above simplified description of the neuron action leads to a very complex neuron model which is not practical. McCulloch and Pitts in 1943 show that even with a very simple neuron model it is possible to build logic and memory circuits. The McCulloch-Pitts neuron model assumes that incoming and outgoing signals may have only binary values *0* and *1*. If incoming signals summed through positive or negative weights have a value larger than threshold T, then the neuron output is set to *1*. Otherwise it is set to *0*.

Examples of McCulloch-Pitts neurons realizing *OR*, *AND*, *NOT* and *MEMORY* operations are shown in Fig. 1. Note, that the structure of *OR* and *AND* gates can be identical and only threshold is different. These simple neurons, known also as perceptrons, are usually more powerful than typical logic gates used in computers.

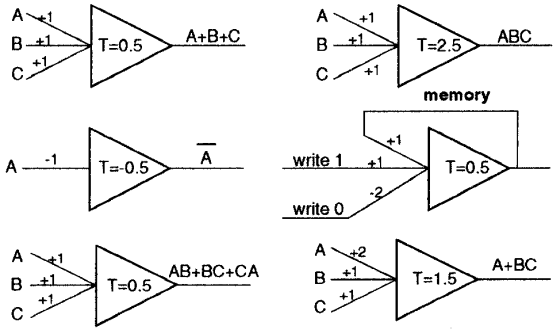

Fig. 1 Several logical operations using networks with McCulloch-Pitts neurons.

Multilayer neural networks usually use continuous activation functions, either unipolar

$$o = f(net) = \frac{1}{1 + \exp(-\lambda net)}$$

or bipolar

$$o = f(net) = \tanh(0.5\lambda net) = \frac{2}{1 + \exp(-\lambda net)} - 1$$

0-7803-4503-7/98/$10.00 1998 IEEE

These continuous activation functions allow for the gradient based training of multilayer networks. Typical activation functions are shown in Fig. 2.

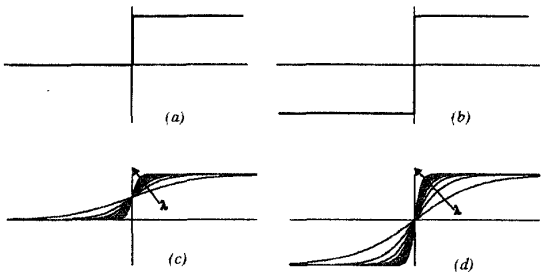

Fig.2. Typical activation functions: *(a)* hard threshold unipolar, *(b)* hard threshold bipolar, *(c)* continuous unipolar, *(d)* continuous bipolar.

III. FEEDFORWARD NEURAL NETWORKS

Simplest and most commonly used neural networks use only for one directional signal flow. Furthermore, most of feedforward neural networks are organized in layers. An example of the three layer feedforward neural network is shown in Fig. 3. This network consists of input nodes, two *hidden layers*, and an output layer.

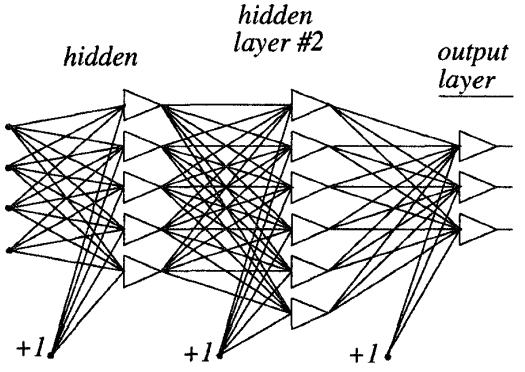

Fig. 3. An example of the three layer feedforward neural network, which is also known as the backpropagation network.

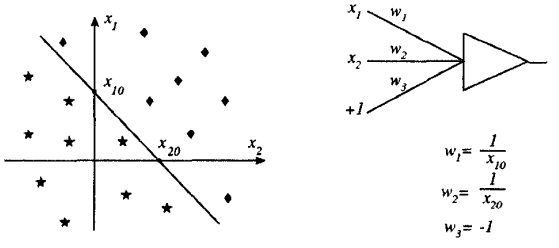

Fig. 4. Linear separation of patterns in the two-dimensional space by a single neuron.

A single neuron is capable of separating input patterns into two categories and this separation is linear. For example, the separation line shown in Fig. 4, which are crossing x_1 and x_2 axes at points x_{10} and x_{20}, can be achieved with a neuron having weights:

$$w_i = \frac{1}{x_{i0}} \quad \text{for} \quad i = 1 \text{ to } n; \qquad w_{n+1} = -1$$

One neuron can divide only linearly separated patterns. In order to select just one region in n-dimensional input space, more than $n+1$ neurons should be used. If more input clusters should be selected then the number of neurons in the input (hidden) layer should be properly multiplied. If the number of neurons in the input (hidden) layer is not limited, then all classification problems can be solved using the three layer network. The linear separation property of neurons makes some problems specially difficult for neural networks, such as exclusive *OR*, parity computation for several bits, or to separate patterns laying on two neighboring spirals.

The feedforward neural networks are used for nonlinear transformation (mapping) of a multidimensional input variable into another multidimensional output variable. In theory, any input-output mapping should be possible if neural network has enough neurons in hidden layers (size of output layer is set by the number of outputs required). Practically, it is not an easy task and presently, there is no satisfactory method to define how many neurons should be used in hidden layers. Usually this is found by try and error method. In general, it is known that if more neurons are used, more complicated shapes can be mapped. On the other side, networks with large number of neurons lose their ability for generalization, and it is more likely that such network will try to map noise supplied to the input also.

IV. LEARNING ALGORITHMS

Weights in artificial neurons are adjusted during a training procedure. Various learning algorithms were developed but only a few are suitable for multilayer neuron networks. Some use only local information about signals in the neurons others require information from outputs. Supervised algorithms require a supervisor who always knows what outputs should be unsupervised algorithms need no such information. Common learning rules are described below.

A. Hebbian Learning Rule

Hebb in 1949 developed unsupervised learning rule which was based on the assumption that if two neighbor neurons must be activated and deactivated at the same time, then the weight connecting these neurons should increase. For neurons operating in the opposite phase, the weight between them should decrease. If there is no correlation, the weight should remain unchanged. This assumption can be described by the formula

$$\Delta w_{ij} = c\, x_i\, o_j$$

where w_{ij} is the weight from *i-th* to *j-th* neuron, c is the learning constant, x_i is the signal on the *i-th* input and o_j is the output signal. The training process starts usually with values of all weights set to zero. This learning rule can be used for both soft and hard threshold neurons. The absolute values of the weights are usually proportional to the learning time, which is undesired.

B. Correlation learning rule

The correlation learning rule uses a similar principle as the Hebbian learning rule. It assumes that weights between simultaneously responding neurons should be largely positive, and weights between neurons with opposite reaction should be largely negative. Mathematically, this can be written that weights should be proportional to the product of states of connected neurons. In contrary to the Hebbian rule, the correlation rule is of the supervised type. Instead of actual response, the desired response is used for weight change calculation

$$\Delta w_{ij} = c\, x_i\, d_j$$

This training algorithm starts with initialization of weights to zero values.

C. Instar learning rule

If input vectors, and weights, are normalized, or they have only binary bipolar values (*-1* or *+1*), then the *net* value will have the largest positive value when the weights have the same values as the input signals. Therefore, weights should be changed only if they are different from the signals

$$\Delta w_i = c(x_i - w_i)$$

Note, that the information required for the weight is only taken from the input signals. This is a local and unsupervised learning algorithm.

D. WTA - Winner Takes All

The WTA is a modification of the instar algorithm where weights are modified only for the neuron with the highest *net* value. Weights of remaining neurons are left unchanged. This unsupervised algorithm (because we do not know what are desired outputs) has a global character. The WTA algorithm, developed by Kohonen in 1982, is often used for automatic clustering and for extracting statistical properties of input data.

E. Outstar learning rule

In the outstar learning rule it is required that weights connected to the certain node should be equal to the desired outputs for the neurons connected through those weights

$$\Delta w_{ij} = c(d_j - w_{ij})$$

where d_j is the desired neuron output and c is small learning constant which further decreases during the learning procedure. This is the supervised training procedure because desired outputs must be known. Both instar and outstar learning rules were developed by Grossberg in 1974.

F. Widrow-Hoff (LMS) learning rule

Widrow and Hoff in 1962 developed a supervised training algorithm which allows to train a neuron for the desired response. This rule was derived by minimizing the square of the difference between *net* and output value.

$$Error_j = \sum_{p=1}^{P} (net_{jp} - d_{jp})^2$$

where $Error_j$ is the error for *j-th* neuron, P is the number of applied patterns, d_{jp} is the desired output for *j-th* neuron when *p-th* pattern is applied. This rule is also known as the LMS (Least Mean Square) rule. By calculating a derivative of the error with respect to w_i, one can find a formula for the weight change.

$$\Delta w_{ij} = c\, x_{ij} \sum_{p=1}^{P} (d_{jp} - net_{jp})$$

Note, that weight change Δw_{ij} is a sum of the changes from each of the individual applied patterns. Therefore, it is possible to correct weight after each individual pattern was applied. If the learning constant c is chosen to be small, then both methods gives the same result. The LMS rule works well for all type of activation functions. This rule tries to enforce the *net* value to be equal to desired value. Sometimes, this is not what we are looking for. It is usually not important what the *net* value is, but it is important if the *net* value is positive or negative. For example, a very large *net* value with a proper sign will result in large error and this may be the preferred solution.

G. Linear regression

The LMS learning rule requires hundreds or thousands of iterations before it converges to the proper solution. Using the linear regression the same result can be obtained in only one step.

Considering one neuron and using vector notation for a set of the input patterns **X** applied through weights **w** the vector of *net* values **net** is calculated using

$$\mathbf{Xw = net}$$

where X is a rectangular array $(n+1)*p$, n is the number of inputs, and p is the number of patterns. Note that the size of the input patterns is always augmented by one, and this additional weight is responsible for the threshold. This method, similar to the LMS rule, assumes a linear activation function, so the *net* values ***net*** should be equal to desired output values ***d***

$$\mathbf{Xw = d}$$

Usually $p > n+1$, and the above equation can be solved only in the least mean square error sense

$$\mathbf{w = (X^T X)^{-1} X^T d}$$

or to convert the set of p equations with $n+1$ unknowns to the set of $n+1$ equations with $n+1$ unknowns. Weights are a solution of the equation

$$\mathbf{Yw = z}$$

where elements of the Y matrix and the z vector are given by

$$y_{ij} = \sum_{p=1}^{P} x_{ip}\, x_{jp} \qquad z_i = \sum_{p=1}^{P} x_{ip}\, d_p$$

H. Delta learning rule

The LMS method assumes linear activation function ***net*** = ***o***, and the obtained solution is sometimes far from optimum as it is shown in Fig. 5 for a simple two dimensional case, with four patterns belonging to two categories. In the solution obtained using the LMS algorithm one pattern is misclassified. If error is defined as

$$Error_j = \sum_{p=1}^{P} (o_{jp} - d_{jp})^2$$

Then the derivative of the error with respect to the weight w_{ij} is

$$\frac{d\,Error_j}{d\,w_{ij}} = 2 \sum_{p=1}^{P} (o_{jp} - d_{jp}) \frac{df(net_{jp})}{d\,net_{jp}} x_i$$

Note, that this derivative is proportional to the derivative of the activation function $f'(net)$. Thus, this type of approach is possible only for continuous activation functions and this method cannot be used with hard activation functions. In this respect the LMS method is more general. The derivatives most common continuous activation functions are

$f' = o(1 - o)$ for the unipolar and

$f' = 0.5(1 - o^2)$ for the bipolar.

Using the cumulative approach, the neuron weight w_{ij} should be changed with a direction of gradient

$$\Delta w_{ij} = c\, x_i \sum_{p=1}^{P} (d_{jp} - o_{jp})\, f_{j'p}$$

in case of the incremental training for each applied pattern

$$\Delta w_{ij} = c\, x_i\, f_{j'} (d_j - o_j)$$

the weight change should be proportional to input signal x_i, to the difference between desired and actual outputs d_{jp}-o_{jp}, and to the derivative of the activation function f_{jp}. Similar to the LMS rule, weights can be updated in both the incremental and the cumulative methods. In comparison to the LMS rule, the delta rule always leads to a solution close to the optimum. As it is illustrated in Fig 5, when the delta rule is used, all four patterns are classified correctly.

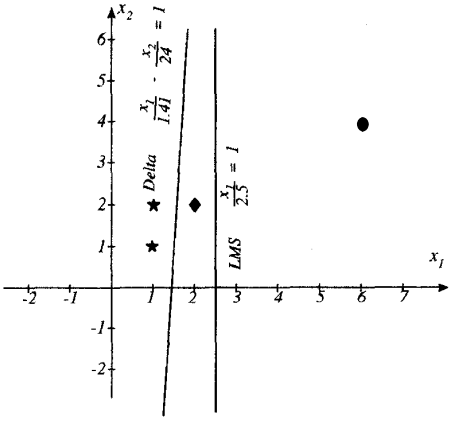

Fig. 5. An example with a comparison of results obtained using LMS and Delta training algorithms. Note that LMS is not able to find the proper solution.

I. Nonlinear regression method

The delta method is usually very slow. Solution can be very fast when nonlinear regression algorithm is adopted [1]. The total error for one neuron j and pattern p is now defined by a simple difference:

$$E_{jpo} = d_{jp} - o_{jp}(net)$$

where $net=w_1x_1+w_2x_2+.....w_nx_n$. The derivative of this error with respect to the i^{th} weight of the j^{th} neuron can be written as:

$$\frac{dE_{jp}}{dw_i} = \frac{do_{jp}}{dnet}\frac{dnet}{dw_i} = -f'_{jp}x_{ijp}$$

The error function can then be approximated by the first two terms of the linear approximation around a given point:

$$E_{jp} = E_{jpo} + \frac{dE_{jp}}{dw_1}\Delta w_1 + \frac{dE_{jp}}{dw_2}\Delta w_2 + ... \frac{dE_{jp}}{dw_n}\Delta w_n$$

therefore

$$\begin{bmatrix} x_{11} & x_{12} & x_{13} & \cdots & x_{1I} \\ x_{21} & x_{23} & x_{23} & \cdots & x_{2I} \\ \vdots & \vdots & \vdots & & \vdots \\ x_{P1} & x_{P2} & x_{P3} & \cdots & x_{PI} \\ \vdots & \vdots & \vdots & & \vdots \\ x_{P1} & x_{F1} & x_{F1} & \cdots & x_{PI} \end{bmatrix} \begin{bmatrix} \nabla w_1 \\ \nabla w_2 \\ \vdots \\ \nabla w_I \end{bmatrix} = \begin{bmatrix} \frac{d_1 - o_1}{f'_1} \\ \frac{d_2 - o_2}{f'_2} \\ \vdots \\ \frac{d_P - o_P}{f'_P} \\ \vdots \\ \frac{d_P - o_P}{f'_P} \end{bmatrix}$$

or

$$\mathbf{X}\Delta\mathbf{w} = \mathbf{v}$$

Matrix X is usually rectangular and above equation can be only solves using pseudo inversion technique.

$$\Delta\mathbf{w} = \left(\mathbf{X}^T\mathbf{X}\right)^{-1}\mathbf{X}^T\mathbf{v}$$

The $\left(\mathbf{X}^T\mathbf{X}\right)^{-1}\mathbf{X}^T$ matrix is composed of input patterns only, and it must be computed only once

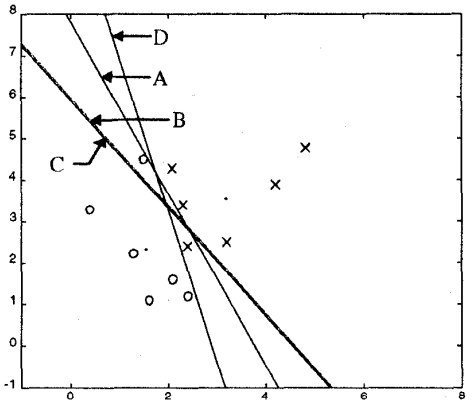

Fig. 6. Comparison of Algorithms where the algorithms can be identified by the labels A-regression, B-minimum distance, C-modified minimum distance, and D-modified regression and delta (back propagation) algorithm.

TABLE 1. Learning rules for single neuron

General formula

$$\Delta \mathbf{w}_i = \alpha \, \delta \, \mathbf{x}$$

Hebb rule (unsupervised):

$$\delta = o$$

correlation rule (supervised):

$$\delta = d$$

perceptron fixed rule:

$$\delta = d - o$$

perceptron adjustable rule:

$$\delta = (d-o)\frac{\mathbf{x}^T\mathbf{w}}{\mathbf{x}^T\mathbf{x}} = (d-o)\frac{net}{\|\mathbf{x}\|^2}$$

LMS (Widrow-Hoff) rule:

$$\delta = d - net$$

delta rule:

$$\delta = (d-o)f'$$

pseudoinverse rule (for linear system):

$$\mathbf{w} = \left(\mathbf{x}^T\mathbf{x}\right)^{-1}\mathbf{x}^T d$$

iterative pseudoinverse rule (for nonlinear system):

$$\mathbf{w} = \left(\mathbf{x}^T\mathbf{x}\right)^{-1}\mathbf{x}^T \frac{d-o}{f'}$$

J. Error Backpropagation learning

The delta learning rule can be generalized for multilayer networks. Using a similar approach, as it is described for the delta rule, the gradient of the global error can be computed in respect to each weight in the network. Interestingly

$$\Delta w_{ij} = c \; x_i \; f_{j'} \; E_j$$

where c is the learning constant, x_i is the signal on the *i-th* neuron input, and f_j' is the derivative of activation function. The cumulative error E_j on neuron output is given by

$$E_j = \frac{1}{f_{j'}} \sum_{k=1}^{K} (o_k - d_k) \; A_{jk}$$

where K is the number of network outputs, and A_{jk} is the small signal gain from the input of *j-th* neuron to the *k-th* network output as Fig. 7 shows. The calculation of the back propagating error is kind of artificial to the real nervous system. Also, the error backpropagation method is not practical from the point of view of hardware realization. Instead, it is simpler to find signal gains from the input of the *j-th* neuron to each of the network output (Fig. 7). In this case weights are corrected using

$$\Delta w_{ij} = c \; x_i \sum_{k=1}^{K} (o_k - d_k) \; A_{jk}$$

Note, that the above formula is general, no matter if neurons are arranged in layers or not. One way to find gains A_{jk} is to introduce an incremental change on the input of the *j-th* neuron and observe the change in the *k-th* network output. This procedure requires only forward signal propagation, and it is easy to implement in a hardware realization. Another possible way, is to calculate gains through each layer and then find the total gains as products of layer gains. This procedure is equally or less computational intensive than a calculation of cumulative errors in the error backpropagation algorithm.

The backpropagation algorithm has a tendency for oscillation. In order to smooth up the process, the weights increment Δw_{ij} can be modified according to Rumelhart, Hinton, and Wiliams [14]

$$w_{ij}(n+1) = w_{ij}(n) + \Delta w_{ij}(n) + \alpha \Delta w_{ij}(n-1)$$

or according to Sejnowski and Rosenberg (1987)

$$w_{ij}(n+1) = w_{ij}(n) + (1 - \alpha)\Delta w_{ij}(n) + \alpha \Delta w_{ij}(n-1)$$

where α is the momentum term.

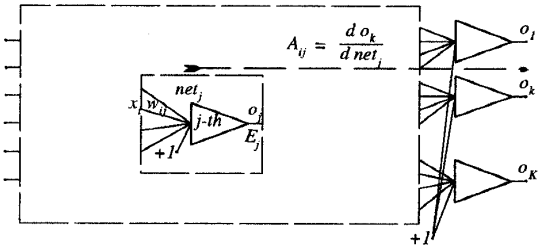

Fig. 7. Illustration of the concept of the gain computation in neural networks

The backpropagation algorithm can be significantly speedup, when after finding components of the gradient, weights are modified along the gradient direction until a minimum is reached. This process can be carried on without the necessity of computational intensive gradient calculation at each step. The new gradient components are calculated once a minimum on the direction of the previous gradient is obtained. This process is only possible for cumulative weight adjustment. One method to find a minimum along the gradient direction is the tree step process of finding error for three points along gradient direction and then, using a parabola approximation, jump directly to the minimum. The fast learning algorithm using the above approach was proposed by Fahlman [2] and it is known as the *quickprop*.

The backpropagation algorithm has many disadvantages which leads to very slow convergence. One of the most painful is this, that in the backpropagation algorithm the learning process almost perishes for neurons responding with the maximally wrong answer. For example if the value on the neuron output is close to *+1* and desired output should be close to *-1*, then the neuron gain $f'(net) \approx 0$ and the error signal cannot back propagate, so the learning procedure is not effective. To overcome this difficulty, a modified method for derivative calculation was introduced by Wilamowski and Torvik [24]. The derivative is calculated as the slope of a line connecting the point of the output value with the point of the desired value as shown in Fig. 8

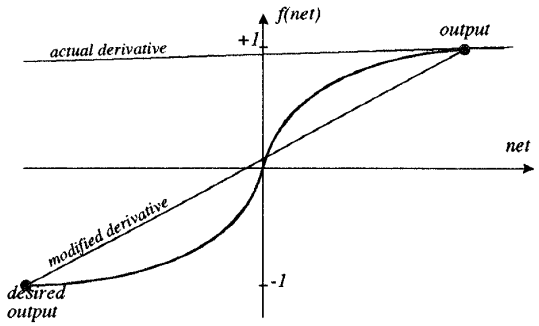

Fig. 8 Illustration of the modified derivative calculation for faster convergence of the error backpropagation algorithm.

$$f_{modif} = \frac{o_{desired} - o_{actual}}{net_{desired} - net_{actual}}$$

Note, that for small errors, equation converges to the derivative of activation function at the point of the output value. With an increase of the system dimensionality, a chance for local minima decrease. It is believed that the described above phenomenon, rather than a trapping in local minima, is responsible for convergence problems in the error backpropagation algorithm.

K. Lavenberg-Marquardt learning

The Lavenberg-Marquardt learning algorithm is the second order search method of a minimum. At each iteration step error surface is approximated by parabolic approximation and the minimum of the paraboloid is the solution for the step.

Simples approach require function approximation by first terms of Taylor series

$$F(\mathbf{w}_{k+1}) = F(\mathbf{w}_k + \Delta\mathbf{w}) + \mathbf{g}_k^T \Delta\mathbf{w}_k + \frac{1}{2}\Delta\mathbf{w}_k^T \mathbf{A}_k \Delta\mathbf{w}_k$$

where $\mathbf{g} = \nabla E$ is gradient and $\mathbf{A} = \nabla^2 E$ is Hessian of global error E.

$$Gradient \Rightarrow \frac{\partial E}{\partial w_1}, \frac{\partial E}{\partial w_2}, \frac{\partial E}{\partial w_3}, \cdots$$

$$Hessian \Rightarrow \begin{matrix} \frac{\partial^2 E}{\partial w_1^2} & \frac{\partial^2 E}{\partial w_1 \partial w_2} & \frac{\partial^2 E}{\partial w_1 \partial w_3} & \cdots \\ \frac{\partial^2 E}{\partial w_2 \partial w_1} & \frac{\partial^2 E}{\partial w_2^2} & \frac{\partial^2 E}{\partial w_2 \partial w_3} & \cdots \\ \frac{\partial^2 E}{\partial w_3 \partial w_1} & \frac{\partial^2 E}{\partial w_3 \partial w_2} & \frac{\partial^2 E}{\partial w_3^2} & \cdots \\ \vdots & \vdots & \vdots & \ddots \end{matrix}$$

Steepest decent (error backpropagation) method calculates weights using:

$$\mathbf{w}_{k+1} = \mathbf{w}_k - \alpha\,\mathbf{g}$$

while Newton method uses:

$$\mathbf{w}_{k+1} = \mathbf{w}_k - \mathbf{A}_k^{-1}\mathbf{g}$$

The Newton method is practical only for small networks where Hessian $\mathbf{A}_k$ can be calculated and inverted. In the Lavenberg-Marquardt method the Hessian $\mathbf{A}_k$ is approximated by product of Jacobians

$$\mathbf{A} \approx 2\mathbf{J}^T\mathbf{J}$$

and gradient as

$$\mathbf{g} \approx 2\mathbf{J}^T\mathbf{e}$$

where $\mathbf{e}$ is vector of output errors and Jacobian $\mathbf{J}$ is

$$Jacobian \Rightarrow \begin{matrix} \frac{\partial E_1}{\partial w_1} & \frac{\partial E_1}{\partial w_2} & \frac{\partial E_1}{\partial w_3} & \cdots \\ \frac{\partial E_2}{\partial w_1} & \frac{\partial E_2}{\partial w_2} & \frac{\partial E_2}{\partial w_3} & \cdots \\ \frac{\partial E_3}{\partial w_1} & \frac{\partial E_3}{\partial w_2} & \frac{\partial E_3}{\partial w_3} & \cdots \\ \vdots & \vdots & \vdots & \ddots \end{matrix}$$

It is much easier to calculate Jacobian than Hessian and also usually Jacobian is much smaller so less memory is required. Therefore weights can be calculated as

$$\mathbf{w}_{k+1} = \mathbf{w}_k - \left(2\mathbf{J}_k^T\mathbf{J}_k\right)^{-1} 2\mathbf{J}_k^T\mathbf{e}$$

or

$$\mathbf{w}_{k+1} = \mathbf{w}_k - \left(\mathbf{J}_k^T\mathbf{J}_k\right)^{-1} \mathbf{J}_k^T\mathbf{e}$$

To secure convergence the Lavenberg-Marquardt introduces μ parameter

$$\mathbf{w}_{k+1} = \mathbf{w}_k - \left(\mathbf{J}_k^T\mathbf{J}_k + \mu\mathbf{I}\right)^{-1} \mathbf{J}_k^T\mathbf{e}$$

when $\mu = 0$ this method is similar to the second order Newton method. For larger values of μ parameter the Lavenberg-Marquardt works as the steepest decent method with small time steps. The m parameter is automatically adjusted during computation process so good convergence is secured. The Lavenberg-Marquardt recently becomes very popular because it will usually converge in 5 to 10 iteration steps. Main disadvantage of this method is a large memory requirement and therefore it cannot be used for

V. SPECIAL FEEDFORWARD NETWORKS

The multilayer backpropagation network, as shown in Fig. 3, is a commonly used feedforward network. This network consists of neurons with the sigmoid type continuous activation function presented in Figures 2(c) and 2(d). In most cases, only the one hidden layer is required, and the number of neurons in the hidden layer are chosen to be proportional to the problem complexity. The number of neurons in the hidden layer is usually found by a try and error process. The training process starts with all weights randomized to small values, and then the error backpropagation algorithm is used to find a solution. When the learning process does not converge, the training is repeated with a new set of randomly chosen weights. Nguyen and Widrow [16] proposed an experimental approach for the two layer network weight initialization. In the second layer, weights are randomly chosen in the range from *-0.5* to *+0.5*. In the first layer, initial weights are calculated from

$$w_{ij} = \frac{\beta z_{ij}}{\|z_j\|} \; ; \qquad w_{(n+1)j} = random(-\beta, +\beta)$$

where z_{ij} is the random number from *-0.5* to *+0.5* and the scaling factor ß is given by

$$\beta = 0.7P^{\frac{1}{N}}$$

where n is the number of inputs, and N is the number of hidden neurons in the first layer. This type of weight initialization usually leads to faster solutions.

For adequate solutions with backpropagation networks, many tries are typically required with different network structures and different initial random weights. This encouraged researchers to develop feedforward networks which can be more reliable. Some of those networks are described below.

A. Functional link network

One layer neural networks are relatively easy to train, but these networks can solve only linearly separated problems. The concept of functional link networks was developed by Nilson book [10] and late elaborated by Pao [13] using the functional link network shown in Fig. 9.

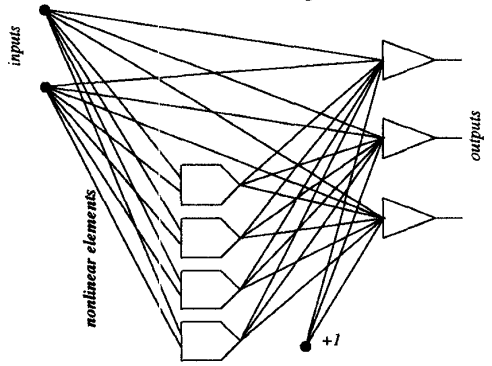

Fig. 9. The functional link network

Using nonlinear terms with initially determined functions, the actual number of inputs supplied to the one layer neural network is increased. In the simplest case nonlinear elements are higher order terms of input patterns. Note that the functional link network can be treated as a one layer network, where additional input data are generated off line using nonlinear transformations. The learning procedure for one layer is easy and fast. Fig. 10 shows an *XOR* problem solved using functional link networks. Note, that when the functional link approach is used, this difficult problem becomes a trivial one. The problem with the functional link network is that proper selection of nonlinear elements is not an easy task. However, in many practical cases it is not difficult to predict what kind of transformation of input data may linearize the problem, so the functional link approach can be used.

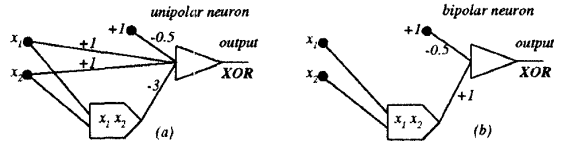

Fig. 10 Functional link networks for solution of the *XOR* problem: (a) using unipolar signals, (b) using bipolar signals.

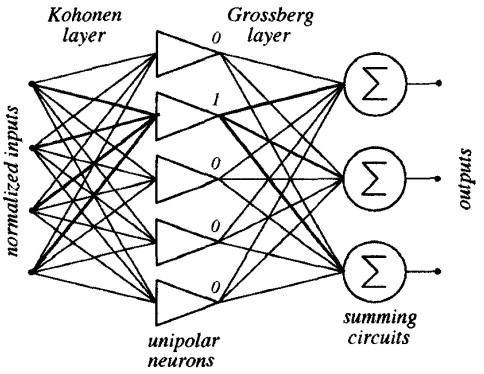

Fig. 11 The counterpropagation network.

B. Feedforward version of the counterpropagation network

The counterpropagation network was originally proposed by Hecht-Nilsen [3] and modified feedforward version described by Zurada [26]. This network, which is shown in Fig. 11, requires numbers of hidden neurons equal to the number of input patterns, or more exactly, to the number of input clusters. The first layer is known as the Kohonen layer with unipolar neurons. In this layer only one neuron, the winner, can be active. The second is the Grossberg outstar layer. The Kohonen layer can be trained in the unsupervised mode, but that need not be the case. When binary input patterns are considered, then the input weights must be exactly equal to the input patterns. In this case

$$net = \mathbf{x}^t\mathbf{w} = (n - 2HD(\mathbf{x},\mathbf{w}))$$

where n is the number of inputs, w are weights, x is the input vector, and $HD(w,x)$ is the Hamming distance between input pattern and weights.

Since for a given input pattern, only one neuron in the first layer may have the value of one and remaining neurons have zero values, the weights in the output layer are equal to the required output pattern.

The feedforward counterpropagation network may also use analog inputs, but in this case all input data should be normalized

$$\mathbf{w}_i = \hat{\mathbf{x}}_i = \frac{\mathbf{x}_i}{\|\mathbf{x}_i\|}$$

The counterpropagation network is very easy to design. The number of neurons in the hidden layer should be equal to the number of patterns (clusters). The weights in the input layer should be equal to the input patterns and, the weights in the output layer should be equal to the output patterns. This simple network can be used for rapid prototyping.

C. WTA architecture

The winner take all WTA network was proposed by Kohonen in 1988 [12]. This is basically a one layer network used in the unsupervised training algorithm to extract a statistical property of the input data (Fig. 12). At the first step all input data is normalized so the length of each input vector is the same, and usually equal to unity. The activation

functions of neurons are unipolar and continuous. The learning process starts with a weight initialization to small random values. During the learning process the weights are changed only for the neuron with highest value on the output - the winner

$$\Delta \mathbf{w}_w = c\,(\mathbf{x} - \mathbf{w}_w)$$

where w_w are weights of the winning neuron, x is the input vector, and c is the learning constant.

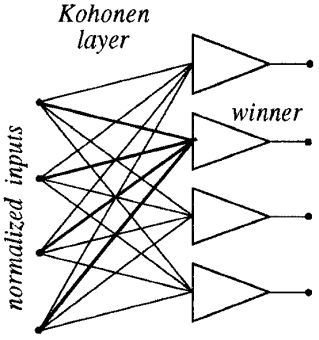

Fig. 12 A winner take all - WTA architecture for cluster extracting in the unsupervised training mode: *(a)* network connections, *(b)* single layer network arranged into a hexagonal shape.

The algorithm is modified in such a way that not only the winning neuron, but also neighboring neurons are allowed for the weight change. At the same time, the learning constant c decreases with the distance from the winning neuron. After such a unsupervised training procedure, the Kohonen layer is able to organize data into clusters.

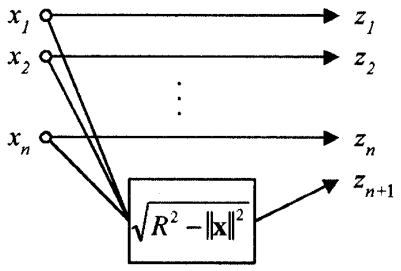

Fig. 13. Input pattern transformation on a sphere

D. Projection on sphere

There are various method to transform input space onto hypersphere without necessity of the information lost [11][21]. In every case the dimensionality of the problem must be increased. The simples way of transforming input space into hypersphere is to use all input variables untouched $z_i = x_i$ and add one additional input z_{n+1}

$$z_i = \begin{cases} x_i & (i = 1,2,\ldots,n) \\ \sqrt{R^2 - \|\mathbf{x}\|^2} & (i = n+1) \end{cases}$$

where $\|\mathbf{x}\|^2 = \mathbf{x}^T\mathbf{x}$ is the norm (length) of the input vector. Note that $\mathbf{z}^T\mathbf{z}$ = *const* therefore all patterns in new transformed z-space are located on a hypersphere with radius R.

$$\mathbf{z}^T\mathbf{z} = x_1^2 + x_2^2 + \cdots + x_n^2 + \left(\sqrt{R^2 - x_1^2 + x_2^2 + \cdots + x_n^2}\right)^2 = R^2$$

The patterns of the transformed z space have the same magnitudes and lie on the $n+1$ hypersphere. Each cluster can now be cut out by a single hyperplane in the $n+1$ dimensional space. The separation hyperplanes should be normal to the vectors specified by thetclusters' centers of gravity zc_k. Equations for the separation plane of the *k-th* cluster can be easily found using the point and normal vector formula:

$$\mathbf{zc}_k^T\left(\mathbf{z} - \mathbf{ze}_k\right) = 0$$

where $\mathbf{zc}_k$ is the center of gravity of the vector and $\mathbf{ze}_k$ is a point transformed from the edge of the cluster. To visualize the problem let us consider a simple two dimensional example with three clusters shown in Fig. 14.

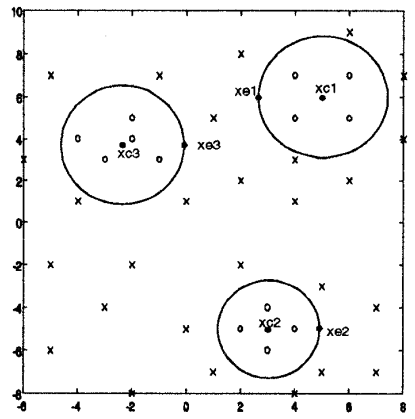

Fig. 14. Separation of three clusters in input space where centers of clusters and cluster edges are marked.

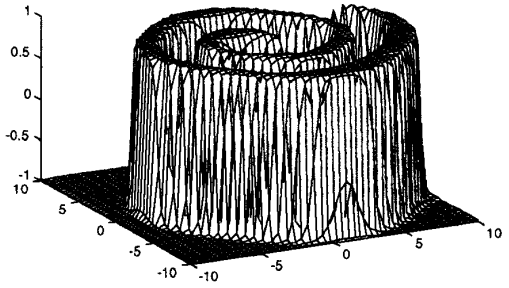

Fig. 15. Spiral problem solved with sigmoidal type neurons

E. Sarajedini and Hecht-Nielsen network

Let us consider stored vector **w** and input pattern **x**. Both input and stored patterns have the same dimension n. The square Euclidean distance between **x** and **w** is:

$$\|\mathbf{x} - \mathbf{w}\|^2 = (x_1 - w_1)^2 + (x_2 - w_2)^2 + \cdots + (x_n - w_n)^2$$

After defactorization

$$\|\mathbf{x}-\mathbf{w}\|^2 = x_1^2 + x_2^2 + \cdots + x_n^2 + w_1^2 + w_2^2 + \cdots + w_n^2 - 2(x_1 w_1 + x_2 w_2 + \cdots + x_n w_n)$$

finally

$$\|\mathbf{x}-\mathbf{w}\|^2 = \mathbf{x}^T\mathbf{x} + \mathbf{w}^T\mathbf{w} - 2\mathbf{x}^T\mathbf{w} = \|\mathbf{x}\|^2 + \|\mathbf{w}\|^2 - 2net$$

where $net = \mathbf{x}^T\mathbf{w}$ is the weighted sum of input signals. Fig. 16 shows the network which is capable of calculating the square of Euclidean distance between input vector **x** and stored pattern **w**.

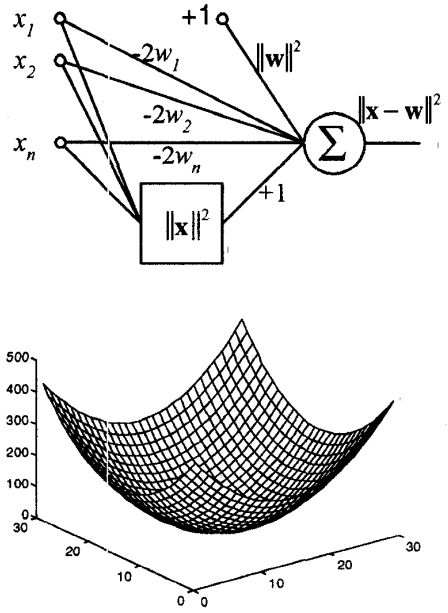

Fig. 16. Network capable of calculating square of Euclidean distance between input pattern**x** and stored pattern**w**. (a) network; (b) Euclidean distance calculation form vector (15,15)

In order to calculate the square of Euclidean distance the following modifications are required: (i) bias equal to $\|\mathbf{w}\|^2$, (ii) additional input with square of input vector magnitude $\|\mathbf{w}\|^2$, and (iii) weights equal to components of stored vector multiplied by -2 factor. Note that when additional input with the square of magnitude is added than simple weighted sum type of network is capable of calculating square of Euclidean distance between **x** and **w**. This approach can be directly incorporated into RBF and LVQ networks.

With the approach presented on Fig. 2 several neural network techniques such as RBF, LVQ, and GR can be used with classical weighted sum type neurons without necessity of computing the Euclidean distance in a traditional way. All what is required is to add additional input with the magnitude of the input pattern.

E. Cascade correlation architecture

The cascade correlation architecture was proposed by Fahlman and Lebiere in 1990. The process of network building starts with a one layer neural network and hidden neurons are added as needed. The network architecture is shown in Fig. 17.

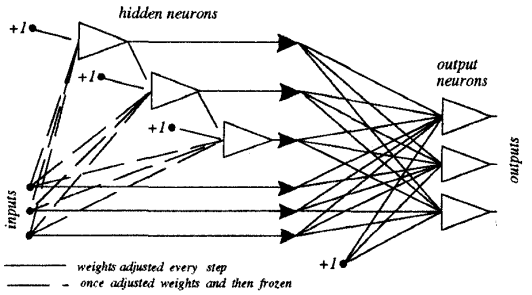

Fig. 17. The cascade correlation architecture.

In each training step, the new hidden neuron is added and its weights are adjusted to maximize the magnitude of the correlation between the new hidden neuron output and the residual error signal on the network output that we are trying to eliminate. The correlation parameter *S* defined below must be maximized

$$S = \sum_{o=1}^{O} \left| \sum_{p=1}^{P} \left(V_p - \overline{V}\right)\left(E_{po} - \overline{E_o}\right) \right|$$

where *O* is the number of network outputs, *P* is the number of training patterns, V_p is output on the new hidden neuron, and E_{po} is the error on the network output. By finding the gradient, $\delta S/\delta w_i$, the weight adjustment for the new neuron can be found as

$$\Delta w_i = \sum_{o=1}^{O} \sum_{p=1}^{P} \sigma_o \left(E_{po} - \overline{E_o}\right) f_{p'} \, x_{ip}$$

where σ_o is the sign of the correlation between the new neuron output value and network output, f_p' is the derivative of activation function for pattern *p*, and x_{ip} is the input signal. The output neurons are trained using the delta (backpropagation) algorithm. Each hidden neuron is trained just once and then its weights are frozen. The network learning and building process is completed when satisfied results are obtained.

G. Radial basis function networks

The structure of the radial basis network is shown in Fig. 16. This type of network usually has only one hidden layer with special "neurons". Each of these "neurons" responds only to the inputs signals close to the stored pattern. The output signal h_i of the *i-th* hidden "neuron" is computed using formula

$$h_i = \exp\left(-\frac{\|\mathbf{x}-\mathbf{s}_i\|^2}{2\sigma^2}\right)$$

where $\boldsymbol{x}$ is the input vector, $\boldsymbol{s_i}$ is the stored pattern representing the center of the *i* cluster, and σ_i is the radius of this cluster. Note, that the behavior of this "neuron" significantly differs form the biological neuron. In this "neuron", excitation is not a function of the weighted sum of the input signals. Instead, the distance between the input and stored pattern is computed. If this distance is zero then the "neuron" responds with a maximum output magnitude equal

to one. This "neuron" is capable of recognizing certain patterns and generating output signals being functions of a similarity. Features of this "neuron" are much more powerful than a neuron used in the backpropagation networks. As a consequence, a network made of such "neurons" is also more powerful.

If the input signal is the same as a pattern stored in a neuron , then this "neuron" responds with *1* and remaining "neurons" have *0* on the output, as it is illustrated in Fig. 16. Thus, output signals are exactly equal to the weights coming out from the active "neuron". This way, if the number of "neurons" in the hidden layer is large, then any input output mapping can be obtained. Unfortunately, it may also happen that for some patterns several "neurons" in the first layer will respond with a non-zero signal. For a proper approximation the sum of all signals from hidden layer should be equal to one. In order to meet this requirement, output signals are often normalized as shown in Fig. 18.

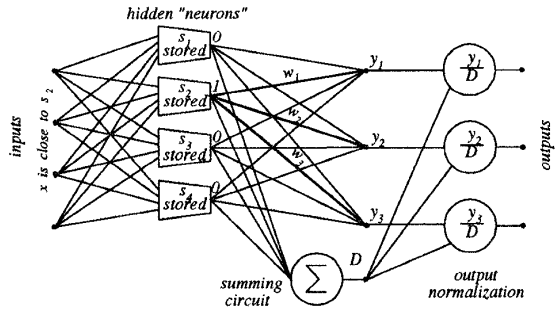

Fig. 18 A typical structure of the radial basis function network.

The radial based networks could be designed, or trained. Training is usually carried on in two steps. In the first step the hidden layer is usually trained in the unsupervised mode for choosing best patterns for cluster representation. A similar approach, as used in the WTA architecture, can be used. Also in this step, radiuses σ_i must be found for a proper overlapping of clusters. The second step of training is the error backpropagation algorithm carried on only for the output layer. Since this is a supervised algorithm for one layer only, the training is very rapid, 100 or 1000 times faster than in the backpropagation multilayer network. This makes the radial basis-function network very attractive. Also, this network can be easily modeled using digital computers, however, its hardware implementation would be very difficult.

VI. RECURRENT NEURAL NETWORKS

In contrast to feedforward neural networks, *recurrent networks* neuron outputs could be connected with their inputs. Thus, signals in the network can continuously circulated. Until now only a limited number of recurrent neural networks were described.

A. Hopfield network

The single layer recurrent network was analyzed by Hopfield in 1982. This network shown in Fig. 19 has unipolar hard threshold neurons with outputs equal to *0* or *1*. Weights are given by a symmetrical square matrix W with zero elements ($w_{ij} = 0$ *for* $i=j$) on the main diagonal. The stability of the system is usually analyzed by means of the *energy function*

$$E = -\frac{1}{2}\sum_{i=1}^{N}\sum_{j=1}^{N} w_{ij} v_i v_j$$

$$\Delta w_{ij} = \Delta w_{ji} = (2v_i - 1)(2v_j - 1)$$

It was proved that during signal circulation the energy E of the network decreases and system converges to the stable points. This is especially true when values of system outputs are updated in the asynchronous mode. This means that at the given cycle, only one random output can be changed to the required value. Hopfield also proved that those stable points to which the system converges can by programmed by adjusting the weights using a modified Hebbian rule. Such memory has limited storage capacity. Based on experiments, Hopfield estimated that the maximum number of stored patterns is *0.15N*, where N is the number of neurons.

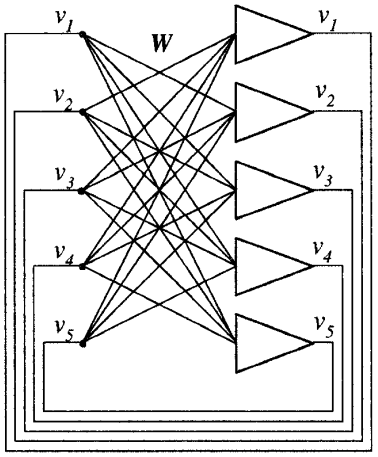

Fig. 19 A Hopfield network or autoassociative memory

B. Autoassociative memory

Hopfield in 1984 extended the concept of his network to autoassociative memories. In the same network structure as shown in Fig. 19, the bipolar neurons were used with outputs equal to *-1* of *+1*. In this network pattern s_m are stored into the weight matrix W using autocorrelation algorithm

$$\mathbf{W} = \sum_{m=1}^{M} \mathbf{s_m} \mathbf{s_m^T} - M\,\mathbf{I}$$

where M is the number of stored pattern, and I is the unity matrix. Note, that W is the square symmetrical matrix with elements on the main diagonal equal to zero (w_{ji} *for* $i=j$). Using a modified formula, new patterns can be added or subtracted from memory. When such memory is exposed to a binary bipolar pattern by enforcing the initial network states,

then after signal circulation the network will converge to the closest (most similar) stored pattern or to its complement. This stable point will be at

$$E(\mathrm{v}) = -\frac{1}{2}\mathbf{v}^{\mathrm{T}}\mathbf{W}\mathbf{v}$$

the closest minimum of the energy functionLike the Hopfield network, the autoassociative memory has limited storage capacity, which is estimated to be about $M_{max}=0.15N$. When the number of stored patterns is large and close to the memory capacity, the network has a tendency to converge to spurious states which were not stored. These spurious states are additional minima of the energy function.

C. BAM

The concept of the autoassociative memory was extended to bi-directional associative memories - BAM by Kosko [7][8]. This memory shown in Fig. 20 is able to associate pairs of the patterns $\boldsymbol{a}$ and $\boldsymbol{b}$. This is the two layer network with the output of the second layer connected directly to the input of the first layer. The weight matrix of the second layer is $\boldsymbol{W}^T$ and it is $\boldsymbol{W}$ for the first layer. The rectangular weight matrix $\boldsymbol{W}$ is obtained as a sum of the cross correlation matrixes

$$\mathbf{W} = \sum_{m=1}^{M} \mathbf{a}_m \mathbf{b}_m$$

where M is the number of stored pairs, and $\boldsymbol{a}_m$ and $\boldsymbol{b}_m$ are the stored vector pairs. If the nodes $\boldsymbol{a}$ or $\boldsymbol{b}$ are initialized with a vector similar to the stored one, then after signal circulation, both stored patterns $\boldsymbol{a}_m$ and $\boldsymbol{b}_m$ should be recovered. The BAM has similar limited memory capacity and memory corruption problems as the autoassociative memory. The BAM concept can be extended for association of three or more vectors.

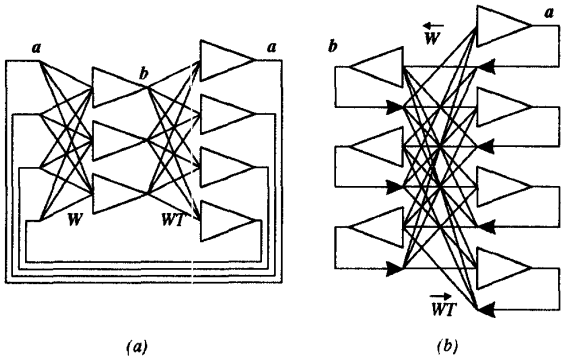

Fig. 20. An example of the bi-directional autoassociative memory - BAM: *(a)* drawn as a two layer network with circulating signals *(b)* drawn as two layer network with bi-directional signal flow.

VII. FUZZY SYSTEMS

A. Fuzzy variables and basic operations

In contrary to the Boolean logic where variables can have only binary states, in fuzzy logic all variables may have any values between zero and one. The fuzzy logic consists of the same basic $\wedge$ - *AND*, $\vee$ - *OR*, and *NOT* operators:

$A \wedge B \wedge C$ ==> $min\{A,B,C\}$ - smallest value of A or B or C

$A \vee B \vee C$ ==> $max\{A,B,C\}$- largest value of A or B or C

$\overline{A}$ ==> $1 - A$ - one minus value of A

For example $0.1 \wedge 0.8 \wedge 0.4 = 0.1$, $0.1 \vee 0.8 \vee 0.4 = 0.8$, and $\overline{0.3} = 0.7$. The above rules are also known as Zadeh [25] *AND*, *OR*, and *NOT* operators. One can note that these rules are true also for classical binary logic.

B. Fuzzy controllers and basic blocks

The principle of operation of the fuzzy controller significantly differs from neural networks. The block diagram of a fuzzy controller is shown in Fig. 21.(a) In the first step, analog inputs are converted into a set of fuzzy variables. In this step usually for each analog input *3* to *9* fuzzy variables are generated. Each fuzzy variable has an analog value between zero and one. In the next step a fuzzy logic is applied to the input fuzzy variables and a resulting set of the output variables is generated. In the last step, known as defuzzification, from a set of output fuzzy variables, one or more output analog variables are generated, which are to be used as control variables.

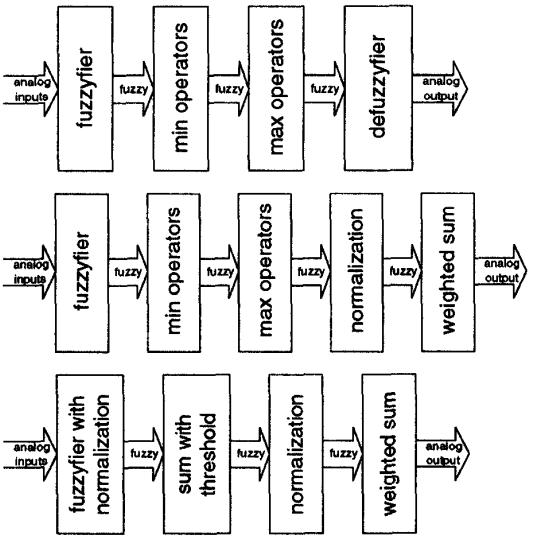

Fig. 21. Typical fuzzy systems (a) proposed by Zadeh, (b) proposed by Takagi-Sugano, and (c) suitable for VLSI implementation

C. Fuzzification

The purpose of fuzzification is to convert an analog variable input into a set of fuzzy variables. For higher accuracy more fuzzy variables will be chosen. To illustrate the fuzzification process let us consider that the input variable is the temperature, and this is coded into five fuzzy variables: cold, cool, normal, warm, and hot. Each fuzzy variable should obtain a value between zero and one, which describes a degree of association of the analog input (temperature) within the given fuzzy variable. Sometimes, instead of term of degree of association, the degree of membership is used. The process of

fuzzification is illustrated in Fig. 22. For example, for a temperature of *57°F*, the following set of fuzzy variables is obtained: *[0, 0.5, 0.3, 0, 0]*, and for *T=80°F* it is *[0, 0, 0.2, 0.7, 0]*. Usually only one or two fuzzy variables have a value different than zero. In the presented example, trapezoidal function are used for calculation of the degree of association. Various different functions as the triangular, or the gaussian can also be used, as long as the computed value is in the range from zero to one. Always each membership function is described by only three or four parameters, which have to be stored in memory.

For proper design of the fuzzification stage, certain practical rules should be used:

1. Each point of the input analog variable should belong to at least one and no more then two membership functions.
2. For overlapping functions, the sum of two membership functions must not be larger than one. This also means that overlaps must not cross the points of maximum values (ones).
3. For higher accuracy, more membership function should be used. However, very dense functions lead to a frequent system reaction, and sometimes to a system instability.

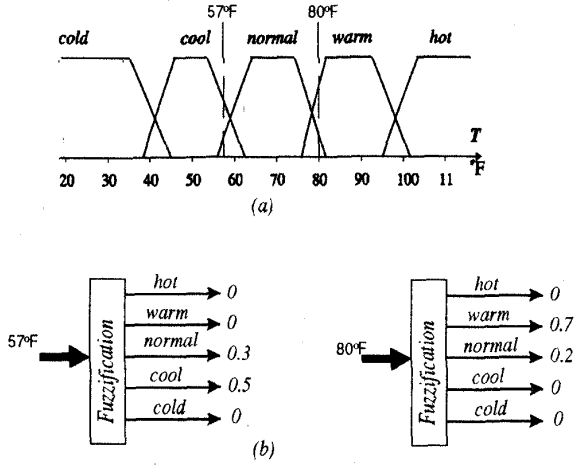

Fig. 20. Fuzzification process: *(a)* a typical membership functions for the fuzzification and the defuzzification processes, *(b)* examples of converting a temperature into fuzzy variables.

	y_1	y_2	y_3
x_1	z_1	z_1	z_2
x_2	z_1	z_2	z_3
x_3	z_1	z_3	z_4
x_4	z_2	z_3	z_4
x_5	z_2	z_3	z_4

(a)

	y_1	y_2	y_3
x_1	t_{11}	t_{12}	t_{13}
x_2	t_{21}	t_{22}	t_{23}
x_3	t_{31}	t_{32}	t_{33}
x_4	t_{41}	t_{42}	t_{43}
x_5	t_{51}	t_{52}	t_{53}

(b)

Fig. 21. Fuzzy tables: *(a)* a table with fuzzy rules, *(b)* table with the intermediate variables t_{ij}.

D. Rule Evaluation

Fuzzy rules are specified in the fuzzy table as it is shown for a given system. Let us consider a simple system with two analog input variables *x* and *y*, and one output variable z. We have to design a fuzzy system generating *z* as *f(x,y)*. Let us also assume that after fuzzification the analog variable *x* is represented by five fuzzy variables: x_1, x_2, x_3, x_4, x_5 and an analog variable *y* is represented by three fuzzy variables: y_1, y_2, y_3. Let us also assume that an analog output variable is represented by four fuzzy variables: z_1, z_2, z_3, z_4. The key issue of the design process is to set proper output fuzzy variables z_k for all combination of input fuzzy variables, as it is illustrated in the table *(a)* shown in Fig. 23. The designer has to specify many rules such as: if inputs are represented by fuzzy variables x_i and y_j then the output should be represented by fuzzy variable z_k. It is only required to specify what membership functions of the output variable are associated with a combination of input fuzzy variables. Once the fuzzy table is specified, the fuzzy logic computation is proceeded with two steps. At first each field of the fuzzy table is filled with intermediate fuzzy variables t_{ij}, obtained from *AND* operator $t_{ij}=min\{x_i,y_j\}$, as shown in Fig. 23*(b)*. This step is independent of the required rules for a given system. In the second step the *OR* (*max*) operator is used to compute each output fuzzy variable z_k. In the given example in Fig. 22 $z_1=max\{t_{11},t_{12},t_{21},t_{31}\}$, $z_2=max\{t_{13},t_{22},t_{41},t_{51}\}$, $z_3=max\{t_{23},t_{32},t_{42},t_{52}\}$, $z_4=max\{t_{33},t_{43},t_{45}\}$. Note that the formulas depend on the specifications given in the fuzzy table shown in Fig 23*(a)*.

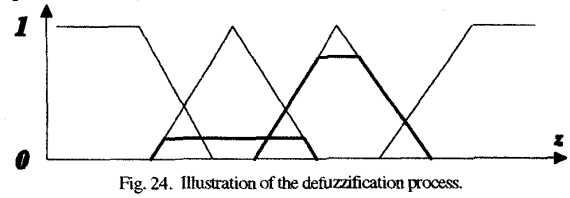

Fig. 24. Illustration of the defuzzification process.

E. Defuzzification

As a result of fuzzy rule evaluation, each analog output variable is represented by several fuzzy variables. The purpose of defuzzification is to obtain analog outputs. This can be done by using the similar membership function as shown in Fig. 22. In the first step fuzzy variables obtained from rule evaluations are used to modify the membership function employing the formula

$$\mu_k^*(z) = \min\{\mu_k(z), z_k\}$$

For example, if output fuzzy variables are: *0, 0.3, 0.7, 0.0*, then the modified membership functions have shapes shown by the thick line in Fig. 24. The analog value of the *z* variable is found as a "center of gravity" of modified membership functions $\mu_k^*(z)$

$$z_{analog} = \frac{\sum_{k=1}^{n} \int_{-\infty}^{+\infty} \mu_k^*(z) z dz}{\sum_{k=1}^{n} \int_{-\infty}^{+\infty} \mu_k^*(z) dz}$$

In the case when shapes of the output membership functions $\mu_k(z)$ are the same, the above equation can simplified to

$$z_{analog} = \frac{\sum_{k=1}^{n} z_k zc_k}{\sum_{k=1}^{n} z_k}$$

where n is the number of membership function of z_{analog} output variable, z_k is fuzzy output variables obtained from rule evaluation, and zc_k are analog values corresponding to the center of k-th membership function.

VIII. VLSI FUZZY CHIP

The classical approach to fuzzy systems presented by Zadeh [25] is difficult to implement in analog hardware. Especially difficult is the defuzzifier where signal division must be implemented. Division can be avoided through use of feedback loops, but this approach can lead to limited accuracy and stability problems. Also, in the case of the classical fuzzy system shown in Fig. 21, the information about the required control surface is encoded in three places: in the fuzzifier, in the defuzzifier, and in the prewired connections between MIN and MAX operators. Although the architecture is relatively simple, it is not suitable for custom programming.

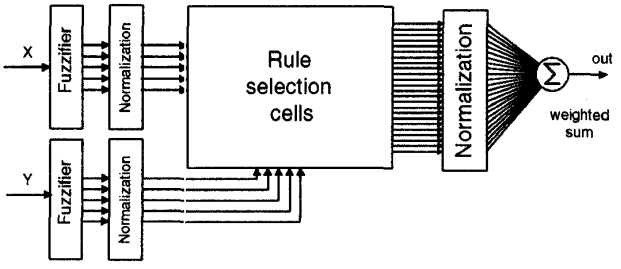

Fig. 25. Architecture of VLSI fuzzy controller

The concept of the proposed circuit is shown in Fig. 25. Fuzzification is done in a traditional manner with additional normalization which leads to a linear interpolation of the output between stored points. The second stage is an array of cluster cells with fuzzy "AND" operators. Instead of classical defuzzification, simplified Takagi-Sugeno singleton inference rules [18] with normalization are used. The output is then normalized and calculated as a weighted sum of the signals approaching from all selected areas.

A. Fuzzifier

Various fuzzifier circuits that can be implemented in bipolar or MOS technology have already been proposed. Most approaches use two source- or emitter-coupled differential pairs for a single membership function. The approach proposed here differs from the previous techniques in two ways (i) it is simpler - only one differential pair is required per membership function and (ii) the fuzzy outputs are automatically normalized; therefore the sum of all the signals representing the fuzzy variables of a single input is constant.

The fuzzifier circuit is presented in Fig. 26. This design requires only one differential pair for each membership function, in contrast to earlier designs where at least two differential pairs were required. Also the output currents are automatically normalized because the sum of I_1 through I_6 is always equal to I_O. Thus the normalization circuit is integrated within the fuzzifier.

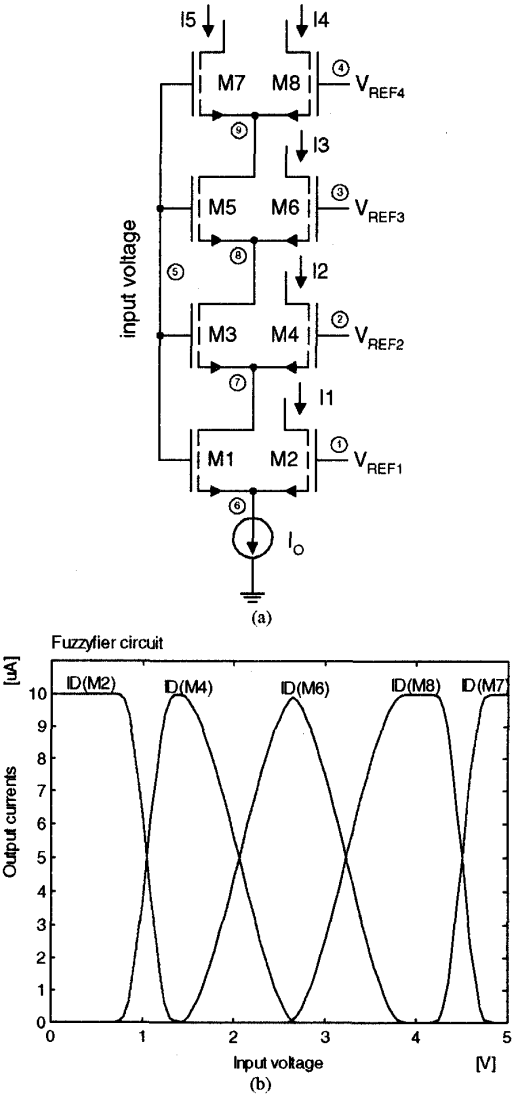

Fig. 26. Fuzzyfier circuit with four differential pairs creating five membership functions: three Gaussian/trapezoidal-like in the center and two sigmoidal types at the ends of the input range: (a) circuit diagram and (b) fuzzyfier characteristics generated by SPICE program.

B. Array of Rule Selection Circuits

Each rule selection circuit is connected to one fuzzy variable from each fuzzifier. Therefore the number of these circuits is equal to $n_1 * n_2$, where n_1 and n_2 are numbers of fuzzy variables for each fuzzifier. The rule selection circuit cell is activated only if both fuzzy inputs have non-zero

values. Due to the specific shapes of the fuzzifier membership functions, where only two membership functions can overlap, a maximum of four cluster cells are active at a time. Although current mode MIN and MAX operators are possible, it is much easier to convert currents from the fuzzifiers into voltages and use the simple rule selection circuits with the fuzzy conjunction (AND) or fuzzy MIN operator.

The voltage on the common node of all sources always follows the highest potential of any of the transistor gates, so it operates as a MAX/OR circuit. However using the negative signal convention (lower voltages for higher signals) this circuit performs the MIN/AND function. This means that the output signal is low only when all inputs are low. A cluster is selected when all fuzzy signals are significantly lower than the positive battery voltage. Selectivity of the circuit increases with larger *W/L* ratios. Transistor M3 would be required only if three fuzzifier circuits were used with three inputs.

C. Normalization circuit

In order to obtain proper outputs it is essential that normalization occurs before weights are applied to the summed currents. The normalization circuit can be implemented using the same concept as the rule selection circuit. For the negative signal convention, PMOS transistors supplied by a common current source are required. The normalization circuit is shown in Fig. 27. The voltage on the common node **A** follows the lowest input potential. The normalization circuit consists of transistors M1, M2, ...M_N connected to a single common current source. This means that the sum of all drain currents of transistors M1, M2, ...M_N is always constant and equal to I_D. The *W/L* ratios in current mirrors can determine the output value for each cluster. Currents from all cluster cells are summed to form the output current.

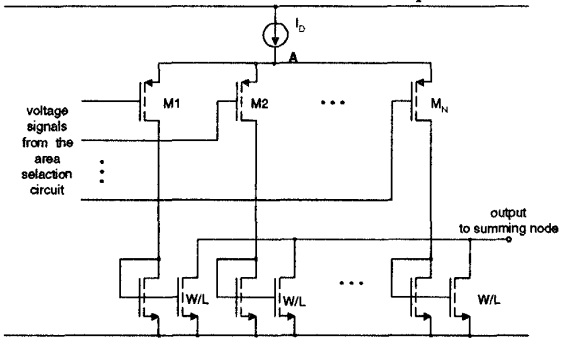

Fig. 27. Normalization circuit

D. Weight circuit

The weights for different clusters can be assigned by setting proper *W/L* ratios for current sources. This task can be also accomplished by introducing a digitally programmable current mirrors.

E. VLSI implementation

A universal fuzzy approximator has been designed and fabricated. In order to make the chip universal, each fuzzifier consists of seven differential pairs with seven equally spaced reference voltages. This results in eight membership functions for each input and 8*8 = 64 cluster cells. Sixty-four adjustable current mirrors for setting weights of output signals are programmed with 6 bit accuracy. For an arbitrary two-dimensional function only 6*64=384 bits are required for programming. A test chip has been implemented in the 2 μm n-well MOSIS process using more than 2000 transistors to perform the analog signal processing. To simplify the test chip implementation, current sources were programmed at the contact mask level. Fig. 28 shows a comparison between the desired and the actually measured control surface from the fuzzy chip.

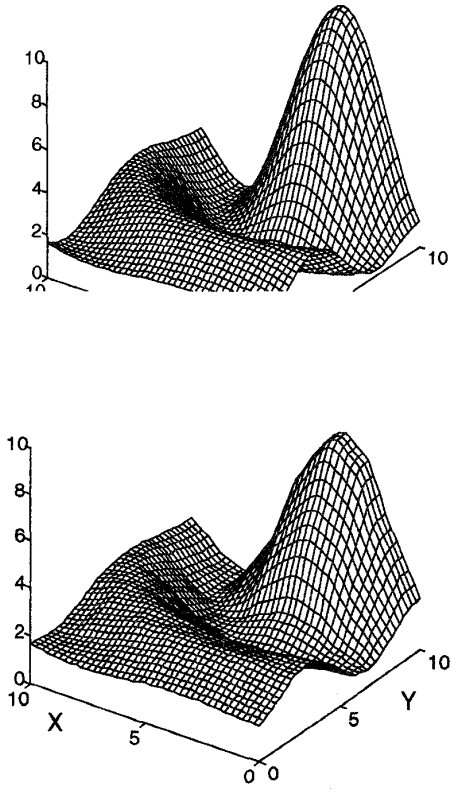

Fig. 28. Control surfaces: (a) desired control surface, and (b) example of a control surface obtained from fuzzy system using weights

V. REFERENCES

[1] Andersen, Thomas J. and B.M. Wilamowski, "A. Modified Regression Algorithm for Fast One Layer Neural Network Training", World Congress of Neural Networks,

vol. 1, pp. 687-690, Washington DC, USA, July 17-21, 1995.

[2] Fahlman, S. E. "Faster-learning variations on backpropagation: An empirical study." *Proceedings of the Connectionist Models Summer School*, eds. D. Touretzky, G. Hinton, and T. Sejnowski, San Mateo, CA: Morgan Kaufmann, 1988.

[3] Hecht-Nielsen, R. "Counterpropagation Networks," *Appl. Opt.*, vol. 26(23) pp. 4979-4984, 1987.

[4] Hopfield, J. J. "Neural networks and physical systems with emergent collective computation abilities. *Proceedings of the National Academy of Science*, vol 79, pp. 2554-2558, 1982.

[5] Hopfield, J. J. "Neurons with graded response have collective computational properties like those of two-state neurons." *Proceedings of the National Academy of Science*, vol 81, pp. 3088-3092, 1984.

[6] Kohonen, T. "The self-organized map," *Proc. IEEE*, vol 78(9), pp. 1464-1480, 1990

[7] Kosko, B. "Adaptive Bidirectional Associative Memories," *App. Opt.* vol 26, pp 4947-4959, 1987

[8] Kosko, B. "Bidirectional Associative Memories," *IEEE Transaction on System Man, and Cybernnetics* vol 18, pp. 49-60, 1988.

[9] Nguyen, D. and B. Widrow, "Improving the learning speed of 2-layer neural networks, by choosing initial values of the adaptive weights." Proc. Intl. Join Conf. on Neural Networks, San Diego Ca, June 1990.

[10] Nilson N. J., (1965) Learning Machines: Foundations of Trainable Pattern Classifiers, New York: McGraw Hill.

[11] Ota Y. and B. Wilamowski, (1994) "Input data transformation for better pattern classification with less neurons," *Proc. of World Congress on Neural Networks*, San Diego, California, vol. 3, pp 667-672.

[12] Ota Y., B.M.Wilamowski, Analog Hardware Implementation of a Voltage-Mode Fuzzy Min-Max Controller, *Journal of Circuits, Systems, and Computers*, Vol. **6**, No.2, pp. 171-184, 1996.

[13] Pao, Y. H. , *Adaptive Pattern Recognition and Neural Networks*, REading, Mass.: Addison-Wesley Publishing Co., 1989.

[14] Rumelhart, D. E., G. E. Hinton, and R. J. Williams, "Learning internal representation by error propagation," *Parallel Distributed Processing*, vol 1, pp. 318-362, Cambrige, MA: MIT Press 1986

[15] Sarajedini A., R. Hecht-Nielsen, (1992) The best of both worlds: Casasent networks integrate multilayer perceptrons and radial basis functions, *International Joint Conference on Neural Networks*, III, 905-910.

[16] Specht, D. F. "General regression neural network." *IEEE Transactions on Neural Networks*, vol 2, pp. 568-576, 1992.

[17] Specht, D. F. "Probalistic neural networks", *Neural Networks*, vol. 3, pp. 109-118.

[18] Takagi and M. Sugeno, Derivation of Fuzzy Control Rules from Human Operator's Control Action. *Proc. of the IFAC Symp. on Fuzzy Inf. Knowledge Representation and Decision Analysis*, pp. 55-60, July 1989.

[19] Tapkan, Baskin I. and Bogdan M. Wilamowski, "Trainable Functional Link Neural Network Architecture", presented at ANNIE'95 - Artificial Neural Networks in

[20] Wilamowski B. M. and R. C. Jaeger, "Neuro-Fuzzy Architecture for CMOS Implementation" accepted for IEEE Transaction on Industrial Electronics

[21] Wilamowski B. M., " Modified EBP Algorithm with Instant Training of the Hidden Layer", Proceedings of Industrial Electronic Conference (IECON'97), New Orleans, November 9-14, 1997, pp. 1097-1101.

[22] Wilamowski, B. M. and Richard C. Jaeger, "VLSI Implementation of a Universal Fuzzy Controller," ANNIE'96 - Artificial Neural Networks in Engineering, St. Louis, Missouri, USA, November 11-14, 1996.

[23] Wilamowski, B. M., "Neural Networks and Fuzzy Systems" chapters 124.1 to 124.8 in The Electronic Handbook. CRC Press 1996, pp. 1893-1914.

[24] Wilamowski, B.M. and L. Torvik, "Modification of Gradient Computation in the Back-Propagation Algorithm", presented at ANNIE'93 - Artificial Neural Networks in Engineering, St. Louis, Missouri, November 14-17, 1993;

[25] Zadeh, L. A. "Fuzzy sets."*Information and Control*, vol 8, 338-353, 1965.

[26] Zurada, J. *Introduction to Artificial Neural Systems*, West Publishing 1992.