Methods of Computational Intelligence

Bogdan M. Wilamowski, *Fellow Member, IEEE*
Auburn University , USA

Abstract **– Comparison of various methods of computational intelligence are presented and illustrated with examples. These methods include neural networks, fuzzy systems, and evolutionary computation. The presentation is focused on neural networks, their learning algorithms and special architectures. General learning rule as a function of the incoming signals is discussed. Other learning rules such as Hebbian learning, perceptron learning, LMS - Least Mean Square learning, delta learning, WTA – Winner Take All learning, and PCA - Principal Component Analysis are presented as a derivation of the general learning rule.**
Architecture specific learning algorithms for cascade correlation networks, Sarajedini and Hecht-Nielsen networks, functional link networks, polynomial networks, counterpropagation networks, RBF-Radial Basis Function networks are described.

I. INTRODUCTION

As a likely result of the on-going development of computer technology we may expect that massive parallel processing and soft computing will significantly enhance traditional computation methods. The methods of computational intelligence includes neural networks, fuzzy systems, and evolutionary computation. They provide a practical alternative for solving mathematically intractable and complex problems.

The mathematical power of computational intelligence is commonly attributed to the neural-like system architecture used and the fault tolerance arising from the massively interconnected structure. Such systems are characterized by heavy parallel processing. The last feature is unfortunately lost if algorithms are implemented using conventional microprocessors or digital computers.

Another aspect of soft computing systems is that instead of "zero" and "one" digital levels, they use fuzzy/continuous levels and in this way much more information is passed through the system. Conventional digital computers are not well suited for such signal processing.

A third feature of computational intelligence is their ability to find a close to optimum solutions for very complex cases which are difficult to handle by analytical methods.

II. NEURAL NETWORKS

The feedforward neural networks allow 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. 1.

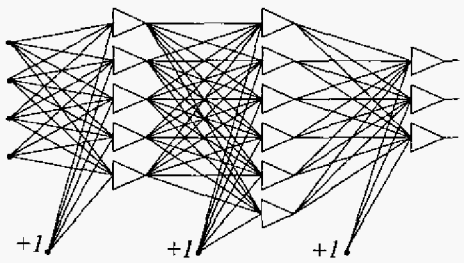

Fig. 1. Feedforward neural networks

A single 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. For example to separate a rectangular pattern 4 neurons are required as it is shown in Fig. 2. 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.

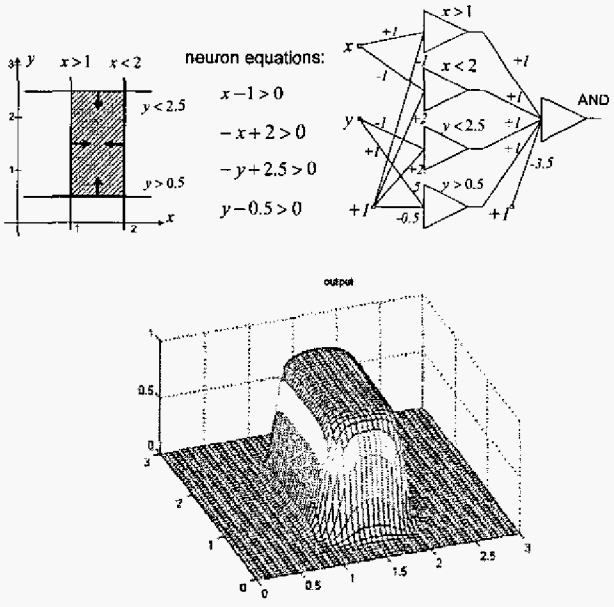

Fig. 2. Separation of input space by the set of four neurons.

The feedforward neural network can be used for nonlinear transformation (mapping) of a multidimensional input variable into another multidimensional variable in the output. Presently, there is no satisfactory method to define how many neurons should be used in hidden layers. Usually this is found by trial

0-7803-8662-0/04/$20.00 ©2004 IEEE

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.

Until now we have mastered only these feedforward neural networks. [1,2,3]. But biological neurons are connected in many loops. Unfortunately until now there are no good methods to analyze neural networks with multiple feedback. The power of these systems was already proven but more research is needed to analyze and design these recurrent networks.

III. FUZZY SYSTEMS

In our digital world we are using Boolean algebra to perform basic AND, OR and NOT function. It is interesting that if the AND operator is replaced with MIN operator (returning minimum value of any input) and OR operator is replaced with MAX operator (returning maximum value of any input) then the same logic function will be performed. This approach however gives an ability to process not only zero and one signals but also any analog signals within a range between zero or one. Fig. 3 illustrates similarity and differences between Boolean operators (AND, OR) and fuzzy operators (MIN, MAX)

Boolean operators

A	B	A∪B
0	0	0
0	1	0
1	0	0
1	1	1

AND

A	B	A∩B
0	0	0
0	1	1
1	0	1
1	1	1

OR

Fuzzy operators

A	B	A∪B
0.1	0.3	0.1
0.1	0.8	0.1
0.7	0.3	0.3
0.7	0.8	0.7

MIN

A	B	A∩B
0.1	0.3	0.3
0.1	0.8	0.8
0.7	0.3	0.7
0.7	0.8	0.8

MAX

Fig. 3. Comparison Boolean and fuzzy operators

The principle of operation of the fuzzy systems significantly differs from neural networks. The block diagram of a fuzzy controller is shown in Fig. 4. In the first step, analog inputs are converted into a set of fuzzy variables. In this step, for each analog input, several fuzzy variables typically 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 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. 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.

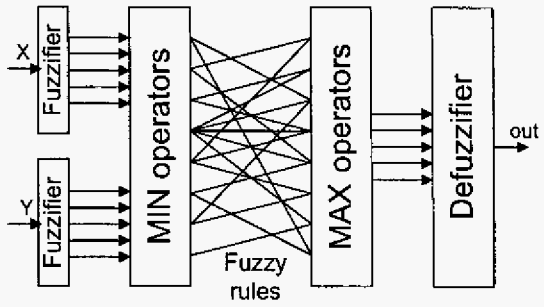

Fig. 4. An example of fuzzy system

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

- Each point of the input analog variable should belong to at least one and no more than two membership functions.
- 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).
- For higher accuracy, more membership functions should be used. However, very dense functions lead to frequent system reaction and sometimes to system instability.

IV. EVOLUTIONARY COMPUTATIONS

The success of the artificial neural networks encouraged researchers to search for other patterns in nature to follow. The power of genetics through evolution was able to create such sophisticated machines as the human being. Genetic algorithms follow the evolution process in nature to find better solutions to some complicated problems. The foundations of genetic algorithms are given by Holland [4] and Goldberg [5]. After initialization, the steps *selection, reproduction with a crossover*, and *mutation* are repeated for each generation. During this procedure, certain strings of symbols, known as chromosomes, evaluate toward a better solution. The genetic algorithm method begins with coding and an initialization. All significant steps of the genetic algorithm will be explained using a simple example of finding a maximum of the function $(\sin^2(x) - 0.5 * x)^2$ with the range of x from 0 to 1.6 (Fig. 5) Note that in this range, the function has a global maximum at x=1.309, and a local maximum at x=0.262.

At first, the variable x has to be represented as a string of symbols. With longer strings, the process usually converges faster, so the fewer symbols for one string field that are used, the better. Although this string may be the sequence of any symbols, the binary symbols 0 and 1 are usually used. In our example, six bit binary numbers are used for coding, having a decimal value of $40x$. The process starts with a random

generation of the initial population given in Table .1.

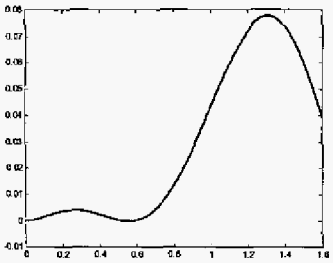

Fig. 5. Function $(\sin^2(x) - 0.5x)^2$ for which the minimum must be found

Table.1 Initial Population

string number	string	decimal value	variable value	function value	fraction of total
1	*101101*	*45*	*1.125*	*0.0633*	*0.2465*
2	*101000*	*40*	*1.000*	*0.0433*	*0.1686*
3	*010100*	*20*	*0.500*	*0.0004*	*0.0016*
4	*100101*	*37*	*0.925*	*0.0307*	*0.1197*
5	*001010*	*10*	*0.250*	*0.0041*	*0.0158*
6	*110001*	*49*	*1.225*	*0.0743*	*0.2895*
7	*100111*	*39*	*0.975*	*0.0390*	*0.1521*
8	*000100*	*4*	*0.100*	*0.0016*	*0.0062*
Total				*0.2568*	*1.0000*

Selection of the best members of the population is an important step in the genetic algorithm. Many different approaches can be used to rank individuals. In this example, the ranking function is given. Highest rank is member number 6, and lowest rank is member number 3. Members with higher rank should have higher chances to reproduce. The probability of reproduction for each member can be obtained as a fraction of the sum of all objective function values. This fraction is shown in the last column of Table 1. Note that to use this approach, our objective function should always be positive. If it is not, the proper normalization should be introduced at first.

The numbers in the last column of Table 1 show the probabilities of reproduction. Therefore, most likely members number 3 and 8 will not be reproduced, and members 1 and 6 may have two or more copies. Using a random reproduction process, the following population, arranged in pairs, could be generated:

101101 → 45 110001 → 49 100101 → 37 110001 → 49

100111 → 39 101101 → 45 110001 → 49 101000 → 40

If the size of the population from one generation to another is the same, two parents should generate two children. By combining two strings, two other strings should be generated. The simplest way to do this is to split in half each of the parent strings and exchange substrings between parents. For example, from parent strings, 010100 and 100111, the following child strings will be generated 010111 and 100100. This process is known as the *crossover*. The resultant children are

101111 → 47 110101 → 53 100001 → 33 110000 → 48

100101 → 37 101001 → 41 110101 → 53 101001 → 41

The second population is shown in Table 2.

TABLE 2 Population of Second Generation

string number	string	decimal value	variable value	function value	fraction of total
1	*010111*	*47*	*1.175*	*0.0696*	*0.1587*
2	*100100*	*37*	*0.925*	*0.0307*	*0.0701*
3	*110101*	*53*	*1.325*	*0.0774*	*0.1766*
4	*010001*	*41*	*1.025*	*0.0475*	*0.1084*
5	*100001*	*33*	*0.825*	*0.0161*	*0.0368*
6	*110101*	*53*	*1.325*	*0.0774*	*0.1766*
7	*110000*	*48*	*1.200*	*0.0722*	*0.1646*
8	*101001*	*41*	*1.025*	*0.0475*	*0.1084*
Total				*0.4387*	*1.0000*

In general, the string need not be split in half. It is usually enough if only selected bits are exchanged between parents. It is only important that bit positions are not changed.

Note that two identical highest ranking members of the second generation are very close to the solution x=1.309. The randomly chosen parents for the third generation are:

010111 → 47 110101 → 53 110000 → 48 101001 → 41

110101 → 53 110000 → 48 101001 → 41 110101 → 53

which produces the following children:

010101 → 21 110000 → 48 110001 → 49 101101 → 45

110111 → 55 110101 → 53 101000 → 40 110001 → 49

The best result in the third population is the same as in the second one. By careful inspection of all strings from the second or third generation, it may be concluded that using crossover, where strings are always split in half, the best solution 110100 → 52 will never be reached, regardless of how many generations are created. This is because none of the population in the second generation has a substring ending with 100. For such crossover, a better result can be only obtained due to the mutation process, which may require many generations. Better results in the future generation also can be obtained when strings are split in random places. Another possible solution is that only randomly chosen bits are exchanged between parents.

The genetic algorithm almost always leads to a good

solution, but sometimes many generations are required. This solution is usually close to global maximum, but not the best. In the case of a smooth function the gradinet metHods are converging much faster and to a better solution. GA are much slower, but more robust.

V. CORE LEARNING ALGORITHMS FOR NEURAL NETWORKS

Similarly to the biological neurons, the weights in artificial neurons are adjusted during a training procedure. Various learning algorithms were developed and only a few are suitable for multilayer neuron networks. Some use only local signals in the neurons, others require information from outputs, some require a supervisor who knows what outputs should be for the given patterns, other - unsupervised algorithms do not require such information.

A. Hebbian learning rule

The Hebb [6] learning rule is 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

$$\Delta w_{ij} = c\, x_i o_j \qquad (1)$$

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.

B. Correlation learning rule

The correlation learning rule is based on 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.

$$\Delta w_{ij} = c\, x_i d_j \qquad (2)$$

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

$$\Delta w_i = c(x_i - w_i) \qquad (3)$$

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 *net* values for all neurons in the network should be compared in each training step. The WTA algorithm, developed by Kohonen [7] 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

$$\Delta w_{ij} = c(d_j - w_{ij}) \qquad (4)$$

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 [8]

F. Perceptron learning rule

$$\Delta \mathbf{w}_i = c\,\delta\,\mathbf{x}_i \qquad (5)$$

$$\delta = d - o \qquad (6)$$

$$\Delta \mathbf{w}_i = \alpha\,\mathbf{x}_i\,(d - \mathrm{sign}(net)) \qquad (7)$$

$$net = \sum_{i=1}^{n} w_i \mathbf{x}_i \qquad (8)$$

G. Widrow-Hoff (LMS) learning rule

Widrow and Hoff developed a supervised training algorithm which allows training a neuron for the desired response. This rule was derived so the square of the difference between *net* and output value is minimized.

$$Error_j = \sum_{p=1}^{P} (net_{jp} - d_{jp})^2 \qquad (9)$$

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, and *net* is given by equation **(7)**. This rule is also known as the LMS (Least Mean Square) rule. By calculating a derivative of **(8)** with respect to w_i, one can find a formula for the weight change.

$$\Delta w_{ij} = c\, x_{ij} \sum_{p=1}^{P} (d_{jp} - net_{jp}) \qquad (10)$$

H. 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

$$\mathbf{Xw} = \mathbf{net} \qquad (11)$$

or

$$\begin{bmatrix} \sum_{p=1}^{P} x_{p1}x_{p1} & \sum_{p=1}^{P} x_{p1}x_{p2} & \cdots & \sum_{p=1}^{P} x_{p1}x_{pN} \\ \sum_{p=1}^{P} x_{p2}x_{p1} & \sum_{p=1}^{P} x_{p2}x_{p2} & \cdots & \sum_{p=1}^{P} x_{p2}x_{pN} \\ \cdots & \cdots & \cdots & \cdots \\ \sum_{p=1}^{P} x_{pN}x_{p1} & \sum_{p=1}^{P} x_{pN}x_{p2} & \cdots & \sum_{p=1}^{P} x_{pN}x_{pN} \end{bmatrix} * \begin{bmatrix} w_1 \\ w_2 \\ \cdots \\ w_N \end{bmatrix} = \begin{bmatrix} \sum_{p=1}^{P} d_p x_{p1} \\ \sum_{p=1}^{P} d_p x_{p2} \\ \cdots \\ \sum_{p=1}^{P} d_p x_{pN} \end{bmatrix} \quad (12)$$

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**

$$\mathbf{Xw} = \mathbf{d} \quad (13)$$

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

$$\mathbf{W} = \left(\mathbf{X}^{\mathrm{T}}\mathbf{X}\right)^{-1} \mathbf{X}^{\mathrm{T}} \mathbf{d} \quad (14)$$

I. Delta learning rule

The LMS method assumes linear activation function ***net*** = ***o***, and the obtained solution is sometimes far from optimum. If error is defined as

$$Error_j = \sum_{p=1}^{P} \left(o_{jp} - d_{jp}\right)^2 \quad (15)$$

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

$$\frac{d\,Error_j}{d\,w_{ij}} = 2 \sum_{p=1}^{P} \left(o_{jp} - d_{jp}\right) \frac{df(net_{jp})}{d\,net_{jp}} x_i \quad (16)$$

Note, that this derivative is proportional to the derivative of the activation function $f'(net)$.

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

$$\Delta w_{ij} = c\, x_i \sum_{p=1}^{P} \left(d_{jp} - o_{jp}\right) f_{j'p} \quad (17)$$

in case of the incremental training for each applied pattern

$$\Delta w_{ij} = c\, x_i\, f_{j'} \left(d_j - o_j\right) \quad (18)$$

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}.

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.

$$o_p = F\left\{f\left(w_1 x_{p1} + w_2 x_{p2} + \cdots + w_n x_n\right)\right\} \quad (19)$$

$$Total\_Error = \sum_{p=1}^{np} \left[d_p - o_p\right]^2 \quad (20)$$

$$\frac{d(TE)}{dw_i} = -2 \sum_{p=1}^{np} \left[\left(d_p - o_p\right) F'\left\{z_p\right\} f'\left(net_p\right) x_{pi}\right] \quad (21)$$

K. Levenberg-Marquardt Algorithm (LM)

Steepest descent method (error backpropagation)

$$\mathbf{w}_{k+1} = \mathbf{w}_k - \alpha\, \mathbf{g} \quad (22)$$

where **g** is gradient vector

$$gradient \quad \mathbf{g} = \begin{matrix} \frac{\partial E}{\partial w_1} \\ \frac{\partial E}{\partial w_2} \\ \vdots \\ \frac{\partial E}{\partial w_n} \end{matrix} \quad (23)$$

Newton method

$$\mathbf{w}_{k+1} = \mathbf{w}_k - \mathbf{A}_k^{-1}\mathbf{g} \quad (24)$$

where $\mathbf{A}_k$ is Hessian

$$Hessian \quad \mathbf{A} = \begin{matrix} \frac{\partial^2 E}{\partial w_1^2} & \frac{\partial^2 E}{\partial w_2 \partial w_1} & \cdots & \frac{\partial^2 E}{\partial w_n \partial w_1} \\ \frac{\partial^2 E}{\partial w_1 \partial w_2} & \frac{\partial^2 E}{\partial w_2^2} & \cdots & \frac{\partial^2 E}{\partial w_n \partial w_2} \\ \vdots & \vdots & \ddots & \vdots \\ \frac{\partial^2 E}{\partial w_1 \partial w_n} & \frac{\partial^2 E}{\partial w_2 \partial w_n} & \cdots & \frac{\partial^2 E}{\partial w_n^2} \end{matrix} \quad (25)$$

$$\mathbf{J} = \begin{bmatrix} \frac{\partial e_{11}}{\partial w_1} & \frac{\partial e_{11}}{\partial w_2} & \cdots & \frac{\partial e_{11}}{\partial w_n} \\ \frac{\partial e_{21}}{\partial w_1} & \frac{\partial e_{21}}{\partial w_2} & \cdots & \frac{\partial e_{21}}{\partial w_n} \\ \vdots & \vdots & & \vdots \\ \frac{\partial e_{M1}}{\partial w_1} & \frac{\partial e_{M1}}{\partial w_2} & \cdots & \frac{\partial e_{M1}}{\partial w_N} \\ \vdots & \vdots & & \vdots \\ \frac{\partial e_{1P}}{\partial w_1} & \frac{\partial e_{1P}}{\partial w_2} & \cdots & \frac{\partial e_{1P}}{\partial w_N} \\ \frac{\partial e_{2P}}{\partial w_1} & \frac{\partial e_{2P}}{\partial w_2} & \cdots & \frac{\partial e_{2P}}{\partial w_N} \\ \vdots & \vdots & & \vdots \\ \frac{\partial e_{MP}}{\partial w_1} & \frac{\partial e_{MP}}{\partial w_2} & \cdots & \frac{\partial e_{MP}}{\partial w_N} \end{bmatrix} \quad (26)$$

$$\mathbf{A} = 2\mathbf{J}^T\mathbf{J} \quad \text{and} \quad \mathbf{g} = 2\mathbf{J}^T\mathbf{e} \quad (27)$$

Gauss-Newton method:

$$\mathbf{w}_{k+1} = \mathbf{w}_k - \left(\mathbf{J}_k^T\mathbf{J}_k\right)^{-1}\mathbf{J}_k^T\mathbf{e} \quad (28)$$

Levenberg - Marquardt method:

$$\mathbf{w}_{k+1} = \mathbf{w}_k - \left(\mathbf{J}_k^T\mathbf{J}_k + \mu\mathbf{I}\right)^{-1}\mathbf{J}_k^T\mathbf{e} \quad (29)$$

The LM algorithm requires computation of the Jacobian J matrix at each iteration step and the inversion of J^TJ square matrix. Note that in the LM algorithm an N by N matrix must be inverted in every iterations. This is the reason why for large size neural networks the LM algorithm is not practical.

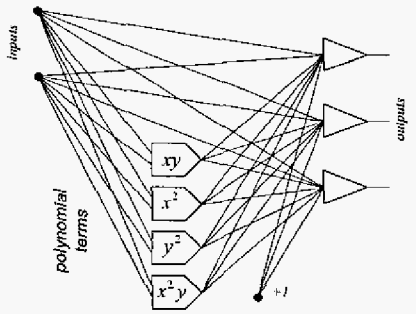

Fig. 6. One layer neural network with nonlinear polynomial terms.

VI. SPECIAL FEEDFORWARD ARCHITECTURES

A. Polynomial Networks

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 polynomial terms of input patterns. Fig. 6 shows an example of functional link networks.

B. Functional link networks

One layer neural networks are relatively easy to train, but these networks can solve only linearly separated problems. One possible solution for nonlinear problems was elaborated by Pao [9] using the functional link network shown in Fig. 7. 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.

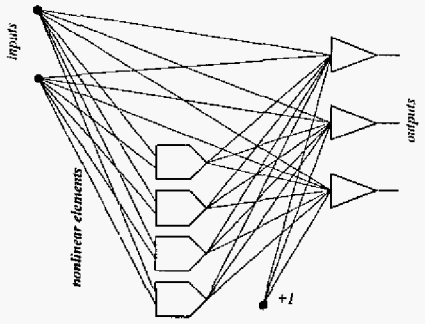

Fig. 7. One layer neural network with arbitrary nonlinear terms.

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

C. Feedforward version of the counterpropagation network

The counterpropagation network was originally proposed by Hecht-Nilsen [10]. This network, which is shown in Fig. 8, requires numbers of hidden neurons equal to the number of input patterns, or more exactly, to the number of input clusters.

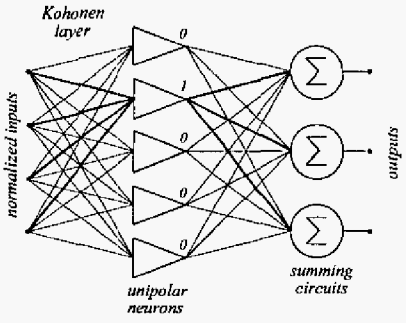

Fig. 8. Counterpropagation network

When binary input patterns are considered, then the input weights must be exactly equal to the input patterns. In this case,

$$net = \mathrm{x}^{\mathrm{t}}\mathrm{w} = \left(n - 2HD(\mathrm{x},\mathrm{w})\right) \qquad (30)$$

where n is the number of inputs, $\boldsymbol{w}$ are weights, $\boldsymbol{x}$ is the input vector, and $HD(\boldsymbol{w},\boldsymbol{x})$ 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 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.

D. LVQ Learning Vector Quantization

At LVQ network the first layer detects subclasses. The second layer combines subclasses into a single class (Fig. 9). First layer computes Euclidean distances between input pattern and stored patterns. Winning "neuron" is with the minimum distance

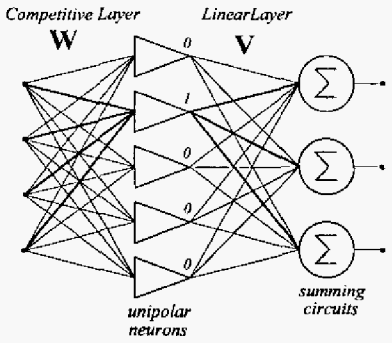

Fig. 9. LVQ Learning Vector Quantization

E. WTA architecture

The winner takes all WTA network was proposed by Kohonen [7]. This is basically a one layer network used in the unsupervised training algorithm to extract a statistical property of the input data. 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.

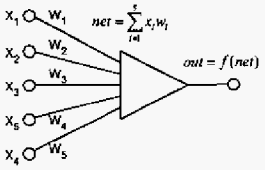

Fig. 10. Neuron as the Hamming distance classifier

If inputs of the neuron of Fig. 10 are binaries, for example $\mathbf{X}$=[1, -1, 1, -1, -1] then the maximum value of *net*

$$net = \sum_{i=1}^{5} x_i w_i = \mathbf{X}\mathbf{W}^T \qquad (31)$$

is when weights are identical to the input pattern **W**=[1, -1, 1, -1, -1] In this case *net* = 5. For binary weights and patterns *net* value can be found using equation:

$$net = \sum_{i=1}^{n} x_i w_i = \mathbf{X}\mathbf{W}^T = n - 2HD \qquad (32)$$

where n is the number of inputs and HD is the Hamming distance between input vector **X** and weight vector **W**. This concept can be extended to weights and patterns with analog values as long as both lengths of the weight vector and input pattern vectors are the same.

The Euclidean distance between weight vector **W** and input vector **X** is

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

$$\|\mathbf{W}-\mathbf{X}\| = \sqrt{\sum_{i=1}^{n}(w_i-x_i)^2} \qquad (33)$$

$$\|\mathbf{W}-\mathbf{X}\| = \sqrt{\mathbf{W}\mathbf{W}^T - 2\mathbf{W}\mathbf{X}^T + \mathbf{X}\mathbf{X}^T} \qquad (34)$$

When the lengths of both the weight and input vectors are normalized to value of one

$$\|\mathbf{X}\| = 1 \quad \text{and} \quad \|\mathbf{W}\| = 1 \qquad (35)$$

Then the equation simplifies to

$$\|\mathbf{W}-\mathbf{X}\| = \sqrt{2-2\mathbf{W}\mathbf{X}^T} \qquad (36)$$

Please notice that the maximum value of net value *net=1* is when **W** and **X** are identical

F. Cascade correlation architecture

The cascade correlation architecture was proposed by Fahlman and Lebiere (fig. 11) The process of network building starts with a one layer neural network and hidden neurons are added as needed.

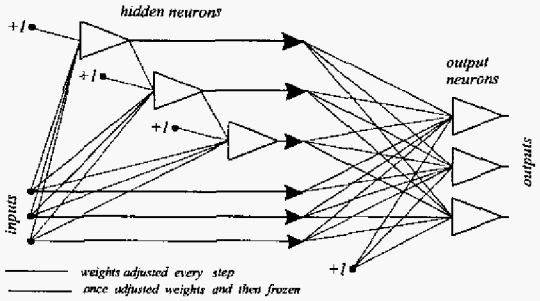

Fig. 11. Cascade correlation architecture

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 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. RBF - Radial basis function networks

The structure of the radial basis network is shown in Fig. 12. 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.

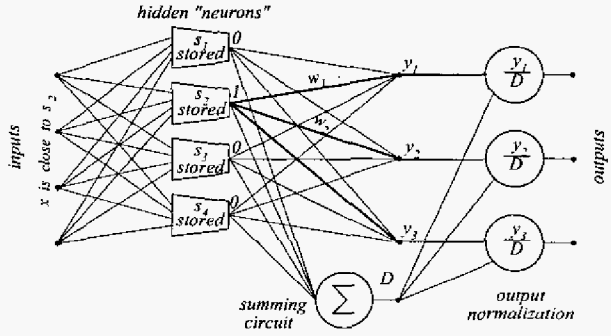

Fig. 12. *Radial basis function networks*

The output signal h_i of the *i-th* hidden "neuron" is computed using formula

$$h_i = \exp\left(-\frac{\|\mathbf{x}-\mathbf{s}_i\|^2}{2\sigma^2}\right) \qquad (37)$$

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.

H. Input pattern transformation

The network shown in Fig. 13 has similar property (and power) like RBF networks, but it uses only traditional neurons with sigmoidal activation functions.

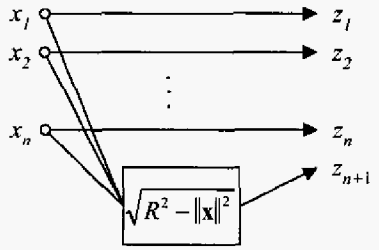

Fig. 13. Transformation, which are required to give a traditional neurons the RBF properties.

VII. Networks for solution of Parity N problems

For the parity-N problem with layered neural networks containing one hidden layer, the weight calculations for the hidden neurons are:

$$w_{i,j} = 1 \quad for \quad i,j = 1,2,\cdots N \tag{38}$$

$$w_{N+1,j} = 2j - N - 1 \quad for \quad j = 1,2,\cdots N \tag{39}$$

While weights for the output neurons are:

$$v_1 = 2\,\mathrm{mod}_2(N) - 1 \tag{40}$$

$$v_j = -v_{j-1} \quad for \quad j = 2,3,\cdots N \tag{41}$$

$$v_{N+1} = \mathrm{mod}_2(N) - 1 \tag{42}$$

For example, the architecture for the parity-8 problem with bipolar neurons is shown in Fig. 15

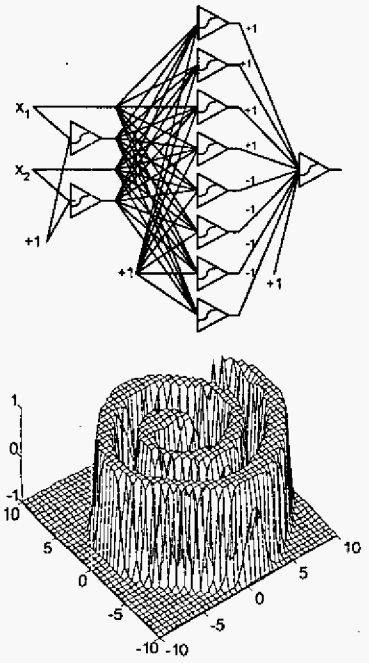

Fig. 14. Solution of two spiral problem using transformation from Fig. 14.

VII. CONCLUSIONS

Neural Networks

- ✓ Any nonlinear transfer function in multi dimensions can be implemented
- ✓ Neural networks could perform better than teacher (trainer)
- ✓ Trained neural networks will work, but humans are not able to trace and understand details of their operation.

Fuzzy Systems

- ✓ Any nonlinear transfer function can be implemented, but limited to several dimensions.
- ✓ Relatively simple and transparent design process

Genetic Algorithms

- ✓ Suitable for any nonlinear optimization process
- ✓ System converges always to good solution, but not optimal one

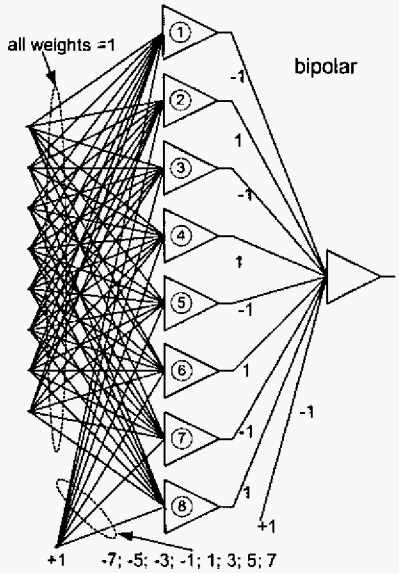

Fig. 15. Layered bipolar neural network with one hidden layer for the parity-8 problem.

REFERENCES

[1] Wasserman, P.D. 1989. *Neural Computing Theory and Practice*. Van Nostrand Reinhold, New York.

[2] Wilamowski, B.M. Neural network architectures and learning 2003 *IEEE International Conference on Industrial Technology* 10-12 Dec. 2003 pp. TU1 - T12

[3] Wilamowski, B.M, Neural Networks and Fuzzy Systems, chapter 32 in Mechatronics Handbook edited by Robert R. Bishop, CRC Press, pp. 33-1 to 32-26, 2002.

[4] Holland, J.H. 1975. *Adaptation in Natural and Artificial Systems*. Univ. of Michigan Press, Ann Arbor, MI.

[5] Goldberg, D.E. 1989. *Genetic Algorithm in Search, Optimization and Machine Learning*. Addison-Wesley, Reading Mass.

[6] Hebb, D.O. 1949. *The Organization of Behivior, a Neuropsychological Theory*. John Wiley, New York

[7] Kohonen, T. 1990. The self-organized map. *Proc. IEEE* 78(9):1464-1480.

[8] Grossberg, S. 1969. Embedding fields: a theory of learning with physiological implications. *Journal of Mathematical Psychology* 6:209-239.

[9] Y. H. Pao, Adaptive Pattern Recognition and Neural Networks, Reading, Mass. Addison-Wesley Publishing Co. 1989

[10] Hecht-Nielsen, R. 1987. Counterpropagation networks. *Appl. Opt.* 26(23):4979-4984.