Neural Network Architectures and Learning

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

***Abstract* - Various leaning method of neural networks including supervised and unsupervised methods are presented and illustrated with examples. 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. Dedicated learning algorithms for on chip neural network training are also evaluated. The tutorial focuses on various practical methods such as Quickprop, RPROP, Back Percolation, Delta-bar-Delta and others. Main reasons of convergence difficulties such as local minima or flat spot problems are analyzed. More advance gradient-based methods including pseudo inversion learning, conjugate gradient, Newton and LM - Levenberg-Marquardt Algorithm are illustrated with examples.

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. A natural consequence of this rapid growth is the emergence of the field of intelligent systems. The machine-intelligent behavior is determined by the flexibility of the architecture, the ability to realize machine incorporations of human expertise, laws of inference procedure and the speed of learning. All these titles are the main constituents of the research area named Computational Intelligence or Soft Computing. It is a practical alternative for solving mathematically intractable and complex problems. The main subdivisions of the area are artificial neural networks and fuzzy inference systems [1]-[5].

The mathematical power of machine 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 soft computing systems is their survivability in the presence of faults; this means they may work correctly if they are partially damaged. In contrast, a one-bit fault in traditional computers may lead to catastrophic results. Therefore there is a significant interest in development of special hardware for soft computing. Several special issues of various journals have been devoted to these topics and several reference books of collected articles on the subject have been published [6-8].

II. FEEDFORWARD 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.. This network consists of input nodes, two *hidden layers*, and an output layer. Typical activation functions are shown in Fig. 2. These continuous activation functions allow for the gradient based training of multilayer networks.

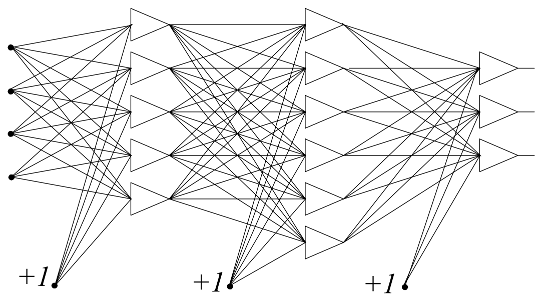


Fig. 1. Feedforward neural networks

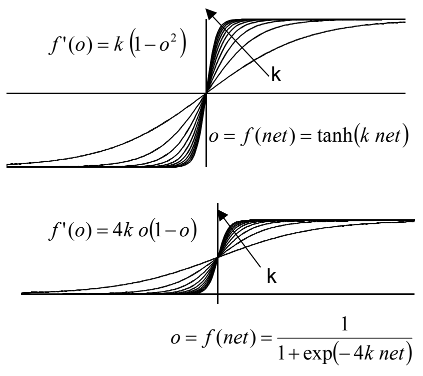


Fig. 2. Typical activation functions

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. 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.

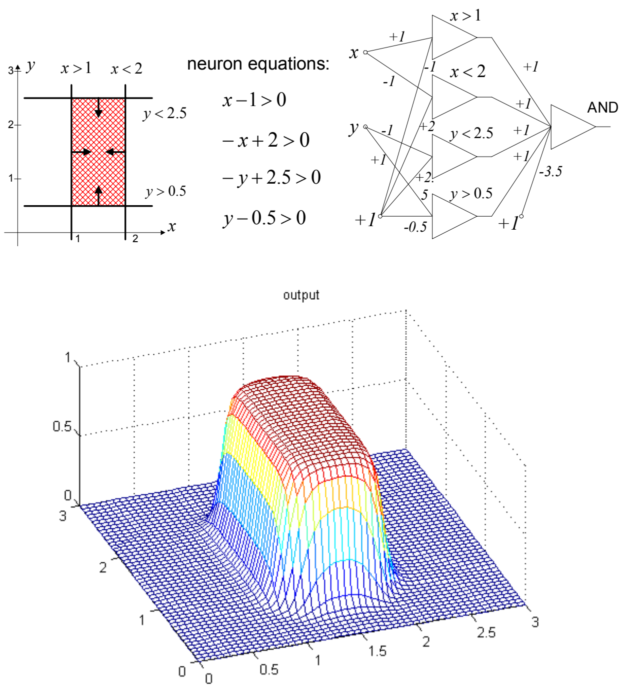


Fig. 3. Separtation of input space by the set of four neurons.

Neurons in the first hidden layer create the separation lines for input clusters. Neurons in the second hidden layer perform *AND* operation. Output neurons perform *OR* operation for each category. The linear separation property of neurons makes some problems especially 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 network is also used for nonlinear transformation (mapping) of a multidimensional input variable into another multidimensional variable in the output. 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. 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.

III. 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 need no such information. Common learning rules are described below.

A. Hebbian learning rule

The Hebb (1949) 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. 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. Since desired responses of neurons are not used in the learning procedure, this is the unsupervised learning rule. The absolute values of the weights are usually proportional to the learning time, which is an undesired effect.

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. In contrary to the Hebbian rule, the correlation rule is the supervised training. Instead of actual response, the desired response is used for weight change calculation

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

This training algorithm starts usually 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

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

Note, that the information required for the weight is only taken from the input signals. This is a very 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. Sometimes this algorithm is modified in such a way that a few neurons with the highest *net* values are modified at the same time. 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 (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

$$\Delta w_{ij} = c\left(d_j - w_{ij}\right) \tag{4}$$

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 (1974)

F. Perceptron learning rule

$$\Delta \mathbf{w}_i = c\,\delta\,\mathbf{x}_i \tag{5}$$

$$\delta = d - o \tag{6}$$

$$\Delta \mathbf{w}_i = \alpha\,\mathbf{x}_i\left(d - \operatorname{sign}(net)\right) \tag{7}$$

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

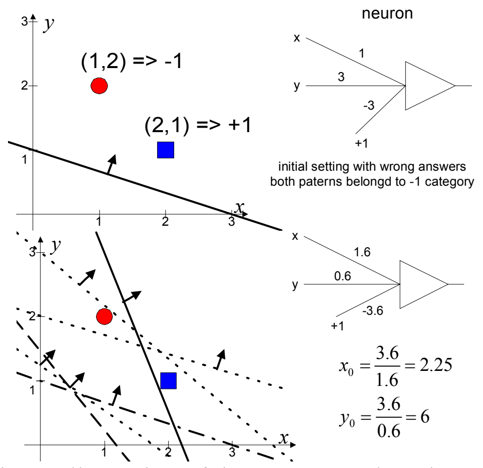

neuron
(1,2) => -1
(2,1) => +1
initial setting with wrong answers
both paterns belongd to -1 category
x
y
1
3
-3
+1
1.6
0.6
-3.6
$x_0 = \frac{3.6}{1.6} = 2.25$
$y_0 = \frac{3.6}{0.6} = 6$

Fig. 4. Illustration of the perceptron learning rule

G. Widrow-Hoff (LMS) learning rule

Widrow and Hoff (1962) developed a supervised training algorithm which allows to train a neuron for the desired response. This rule was derived so the square of the difference between *net* and output value is minimized.

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

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.

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

Note, that weight change $?w_{ij}$ 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. This is *incremental updating*, while *cumulative updating* is when weights are changed after all patterns were applied. The incremental updating usually leads to a solution faster, but it is sensitive to the order in which patterns are 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.

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

$$\mathbf{Xw} = \mathbf{net} \tag{11}$$

or

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

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**

$$\mathbf{Xw} = \mathbf{d} \tag{13}$$

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

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

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

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

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

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

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

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

in case of the incremental training for each applied pattern

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

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}. 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.

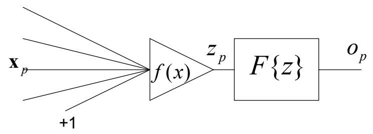
J. Error Backpropagation learning

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.

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

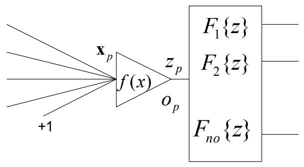
$$Total\_Error = \sum_{p=1}^{np} \left[d_p - o_p\right]^2 \tag{20}$$

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



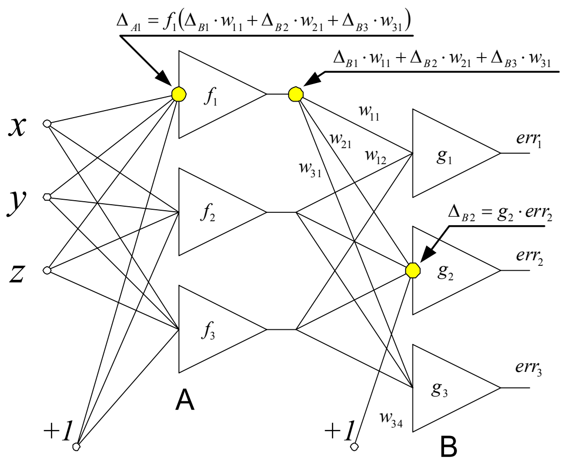
$$\Delta \mathbf{w}_p = \alpha \sum_{p=1}^{np} \left[\left(d_p - o_p\right) F'\left\{z_p\right\} f'\left(net_p\right) \mathbf{x}_p\right]$$

Fig. 5. Error backpropagation for neural networks with one output



$$\Delta \mathbf{w}_p = \alpha \sum_{o=1}^{no} \sum_{p=1}^{np} \left[\left(d_{op} - o_{op}\right) F'\left\{z_p\right\} f'\left(net_p\right) \mathbf{x}_p\right]$$

Fig. 6. Error backpropagation for neural networks with multiple output



$\Delta_{A1} = f_1\left(\Delta_{B1} \cdot w_{11} + \Delta_{B2} \cdot w_{21} + \Delta_{B3} \cdot w_{31}\right)$
$\Delta_{B1} \cdot w_{11} + \Delta_{B2} \cdot w_{21} + \Delta_{B3} \cdot w_{31}$
x
y
z
f_1
f_2
f_3
w_{11}
w_{21}
w_{12}
w_{31}
g_1
g_2
g_3
err_1
$\Delta_{B2} = g_2 \cdot err_2$
err_2
err_3
A
+1
+1
w_{34}
B

Fig. 7. Calculation errors in neural network using error backpropagation algorithm

III. HEURISTIC APPROACHES TO EBP

The backpropagation algorithm has many disadvantages which leads to very slow convergence. One of the most painful is that in the backpropagation algorithm the learning process almost perishes for neurons responding with the maximally wrong answer.

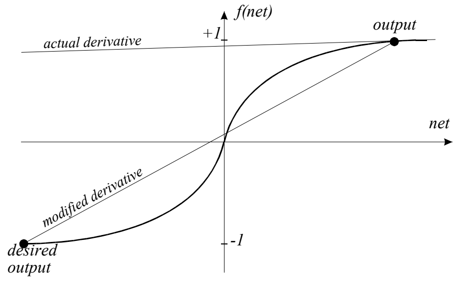


Fig. 8. Lack of error backpropagation for very large errors.

$$f'(net) = k\left[1 - o^2\right] \tag{22}$$

$$f'(net) = k\left[1 - o^2\left(1 - \left(\frac{err}{2}\right)^2\right)\right] \tag{23}$$

For small errors: $f'(net) = k\left[1 - o^2\right]$ (24)

For large errors: $f'(net) = k$ (25)

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)?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 (1993). 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

$$f_{modif} = \frac{o_{desired} - o_{actual}}{net_{desired} - net_{actual}} \tag{26}$$

Note, that for small errors, equation **(26)** 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.

A. Momentum term

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 (1986)

$$w_{ij}(n+1) = w_{ij}(n) + \Delta w_{ij}(n) + \alpha \Delta w_{ij}(n-1) \tag{27}$$

or according to Sejnowski and Rosenberg (1987)

$$w_{ij}(n+1) = w_{ij}(n) + (1-\alpha)\,\Delta w_{ij}(n) + \alpha \Delta w_{ij}(n-1) \tag{28}$$

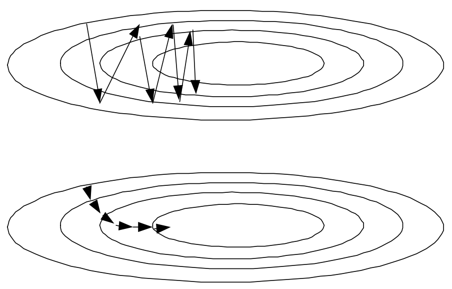


Fig. 9. Solution proces without and with momentum term.

B. Gradient direction search

The backpropagation algorithm can be significantly sped up, 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.

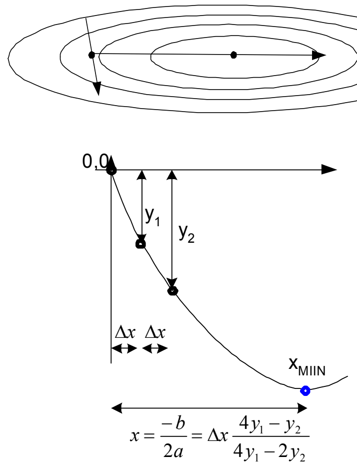

0,0
y1
y2
Δx Δx
xMIIN
$x = \frac{-b}{2a} = \Delta x \frac{4y_1 - y_2}{4y_1 - 2y_2}$

Fig. 10. Search on the gradien direction befor a new calculation of gradient components.

C. Quickprop algorithm by Fahlman

The fast learning algorithm using the above approach was proposed by Fahlman (1988) and it is known as the *quickprop*.

$$\Delta w_{ij}(t) = -\alpha\, S_{ij}(t) + \gamma_{ij} \Delta w_{ij}(t-1) \tag{29}$$

$$S_{ij}(t) = \frac{\partial E(\mathbf{w}(t))}{\partial w_{ij}} + \eta w_{ij}(t) \tag{30}$$

α learning constant
γ memory constant (small 0.0001 range) leads to reduction of weights and limits growth of weights
η momentum term selected individually for each weight

$$0.01 < \alpha < 0.6 \qquad \text{when} \qquad \Delta \mathrm{w_{ij}} = 0 \text{ or sign of } \Delta \mathrm{w_{ij}}$$
$$\alpha = 0 \qquad \text{otherwise}$$
$$S_{ij}(t)\Delta w_{ij}(t) > 0 \tag{31}$$

$$\Delta w_{ij}(t) = -\alpha\, S_{ij}(t) + \gamma_{ij} \Delta w_{ij}(t-1) \tag{32}$$

momentum term selected individually for each weight is very important part of this algorithm. Quickprop algorithm sometimes reduces computation time a hundreds times
Later this algorithm was simplified:

$$\beta_{ij}(t) = \frac{S_{ij}(t)}{S_{ij}(t-1) - S_{ij}(t)} \tag{33}$$

Modified Quickprop algorithm even is simpler often gives better results than the original one.

D. RPROP Resilient Error Back Propagation

Very similar to EBP, but weights adjusted without using values of the propagated errors, but only its sign. Learning constants are selected individually to each weight based on the history

$$\Delta w_{ij}(t) = -\alpha_{ij}\,\mathrm{sgn}\left(\frac{\partial E(\mathbf{w}(t))}{\partial w_{ij}(t)}\right) \tag{34}$$

$$S_{ij}(t) = \frac{\partial E(\mathbf{w}(t))}{\partial w_{ij}} + \eta w_{ij}(t) \tag{35}$$

$$\alpha_{ij}(t) = \begin{cases} \min\left(a\cdot\alpha_{ij}(t-1), \alpha_{\max}\right) & \text{for} \quad \mathrm{S_{ij}}(t)\cdot \mathrm{S_{ij}}(t-1) > 0 \\ \max\left(b\cdot\alpha_{ij}(t-1), \alpha_{\min}\right) & \text{for} \quad \mathrm{S_{ij}}(t)\cdot \mathrm{S_{ij}}(t-1) < 0 \\ \alpha_{ij}(t-1) & \text{otherwise} \end{cases}$$

E. Back Percolation

Error is propagated as in EBP and than each neuron is “trained” using and algorithm to train one neuron such as pseudo inversion. Unfortunately pseudo inversion may lead to errors, which are sometimes larger than 2 for bipolar or larger then 1 for unipolar

G. Delta-bar-Delta

For each weight the learning coefficient is selected individually. It was developed for quadratic error functions

$$\Delta\alpha_{ij}(t) = \begin{cases} a & \text{for} \quad S_{ij}(t-1)D_{ij}(t) > 0 \\ -b\cdot\alpha_{ij}(t-1) & \text{for} \quad S_{ij}(t-1)D_{ij}(t) < 0 \\ 0 & \text{otherwise} \end{cases} \tag{36}$$

$$D_{ij}(t) = \frac{\partial E(t)}{\partial w_{ij}(t)} \tag{37}$$

$$S_{ij}(t) = (1-\xi)D_{ij}(t) + \xi\, S_{ij}(t-1) \tag{38}$$

IV. ADVANCED LEARNIG ALGORITHMS

A. Levenberg-Marquardt Algorithm (LM)

Steepest descent method (error backpropagation)

$$\mathbf{w}_{k+1} = \mathbf{w}_k - \alpha\,\mathbf{g} \tag{39}$$

where **g** is gradient vector

$$\textit{gradient} \quad \mathbf{g} = \begin{matrix} \dfrac{\partial E}{\partial w_1} \\ \dfrac{\partial E}{\partial w_2} \\ \vdots \\ \dfrac{\partial E}{\partial w_n} \end{matrix} \tag{40}$$

Newton method

$$\mathbf{w}_{k+1} = \mathbf{w}_k - \mathbf{A}_k^{-1}\mathbf{g} \tag{41}$$

where **Ak** is Hessian

$$\textit{Hessian} \quad \mathbf{A} = \begin{matrix} \dfrac{\partial^2 E}{\partial w_1^2} & \dfrac{\partial^2 E}{\partial w_2 \partial w_1} & \cdots & \dfrac{\partial^2 E}{\partial w_n \partial w_1} \\ \dfrac{\partial^2 E}{\partial w_1 \partial w_2} & \dfrac{\partial^2 E}{\partial w_2^2} & \cdots & \dfrac{\partial^2 E}{\partial w_n \partial w_2} \\ \vdots & \vdots & \ddots & \vdots \\ \dfrac{\partial^2 E}{\partial w_1 \partial w_n} & \dfrac{\partial^2 E}{\partial w_2 \partial w_n} & \cdots & \dfrac{\partial^2 E}{\partial w_n^2} \end{matrix} \tag{42}$$

$$\mathbf{J} = \begin{bmatrix} \dfrac{\partial \mathrm{e}_{11}}{\partial \mathrm{w}_1} & \dfrac{\partial \mathrm{e}_{11}}{\partial \mathrm{w}_2} & \cdots & \dfrac{\partial \mathrm{e}_{11}}{\partial \mathrm{w}_n} \\ \dfrac{\partial \mathrm{e}_{21}}{\partial \mathrm{w}_1} & \dfrac{\partial \mathrm{e}_{21}}{\partial \mathrm{w}_2} & \cdots & \dfrac{\partial \mathrm{e}_{21}}{\partial \mathrm{w}_n} \\ \vdots & \vdots & & \vdots \\ \dfrac{\partial \mathrm{e}_{M1}}{\partial \mathrm{w}_1} & \dfrac{\partial \mathrm{e}_{M1}}{\partial \mathrm{w}_2} & \cdots & \dfrac{\partial \mathrm{e}_{M1}}{\partial \mathrm{w}_N} \\ \vdots & \vdots & & \vdots \\ \dfrac{\partial \mathrm{e}_{1P}}{\partial \mathrm{w}_1} & \dfrac{\partial \mathrm{e}_{1P}}{\partial \mathrm{w}_2} & \cdots & \dfrac{\partial \mathrm{e}_{1P}}{\partial \mathrm{w}_N} \\ \dfrac{\partial \mathrm{e}_{2P}}{\partial \mathrm{w}_1} & \dfrac{\partial \mathrm{e}_{2P}}{\partial \mathrm{w}_2} & \cdots & \dfrac{\partial \mathrm{e}_{2P}}{\partial \mathrm{w}_N} \\ \vdots & \vdots & & \vdots \\ \dfrac{\partial \mathrm{e}_{MP}}{\partial \mathrm{w}_1} & \dfrac{\partial \mathrm{e}_{MP}}{\partial \mathrm{w}_2} & \cdots & \dfrac{\partial \mathrm{e}_{MP}}{\partial \mathrm{w}_N} \end{bmatrix} \tag{43}$$

$$\mathbf{A} = 2\mathbf{J}^T\mathbf{J} \quad \text{and} \quad \mathbf{g} = 2\mathbf{J}^T\mathbf{e} \tag{44}$$

Gauss-Newton method:

$$\mathbf{w}_{k+1} = \mathbf{w}_k - \left(\mathbf{J}_k^T\mathbf{J}_k\right)^{-1}\mathbf{J}_k^T\mathbf{e} \tag{45}$$

Levenberg - Marquardt method:

$$\mathbf{w}_{k+1} = \mathbf{w}_k - \left(\mathbf{J}_k^T\mathbf{J}_k + \mu\mathbf{I}\right)^{-1}\mathbf{J}_k^T\mathbf{e} \tag{46}$$

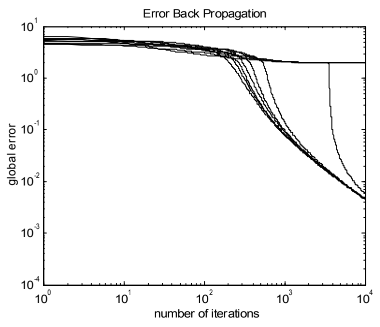


Fig. 11. Sum of squared errors as a function of number of iterations for the “XOR” problem using EBP algorithm

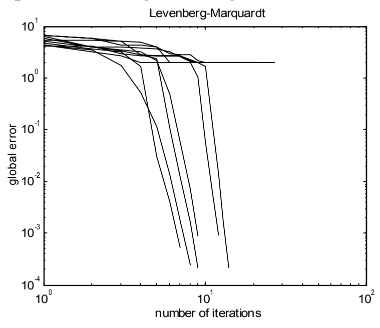


Fig. 12. Sum of squared errors as a function of number of iterations for the “XOR” problem using LM algorithm

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.

V. SPECIAL FEEDFORWARD ARCHITECTURES

For adequate solutions with backpropagation networks, many tries are typically required with different network structures and different initial random weights. It is important, that the trained network gains a generalization property. This means that the trained network should be able to handle correctly also patterns which were not used for training. Therefore, in the training procedure, often some data are removed from training patterns and then these patterns are used for verification. The results with backpropagation networks often depends on luck. This encouraged researchers to develop feedforward networks which can be more reliable. Some of those networks are described below.

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. The learning procedure for one layer is easy and fast. Fig. 13 shows an *XOR* problem solved using functional link networks.

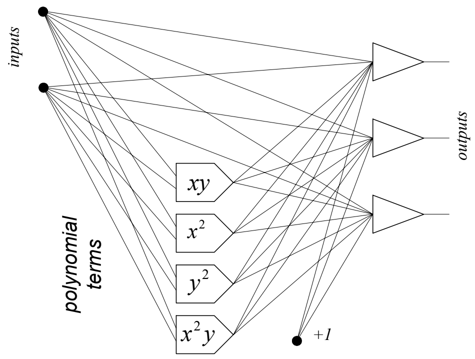


Fig. 13. One layer neural network with nonlinear polynomial terms.

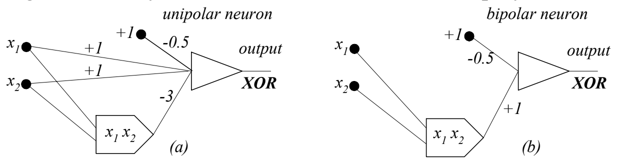


Fig. 14. Polynomial networks for solution of the *XOR* problem: (a) using unipolar signals, (b) using bipolar signals.

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 (1989) using the functional link network shown in Fig. 15. 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.

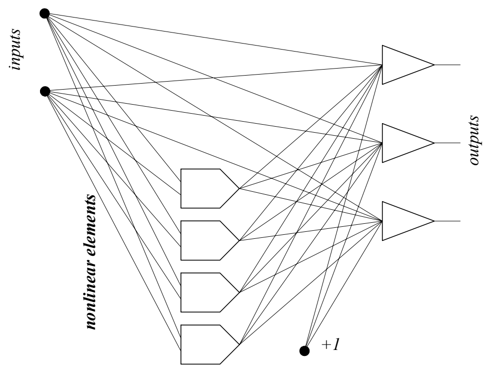
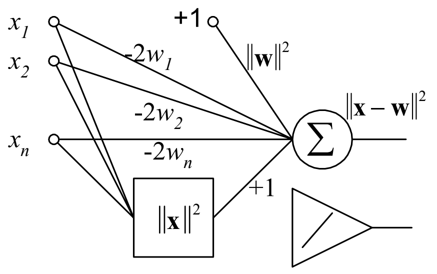


Fig. 15. One layer neural network with arbitrary nonlinear terms.

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. 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.

C. Sarajedini and Hecht-Nielsen network



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

Fig. 15. Sarajedini and Hecht-Nielsen neural network.

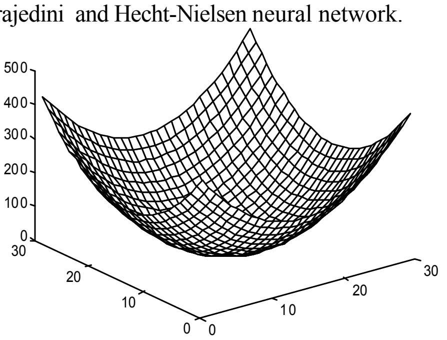


Fig. 16. Ouput of the Sarajedini and Hecht-Nielsen network is proportional to the square of Euclidean distance.

D. Feedforward version of the counterpropagation network

The counterpropagation network was originally proposed by Hecht-Nilsen (1987). In this chapter a modified feedforward version as described by Zurada (1992) is discussed. This network, which is shown in Fig. 17, requires numbers of hidden neurons equal to the number of input patterns, or more exactly, to the number of input clusters.

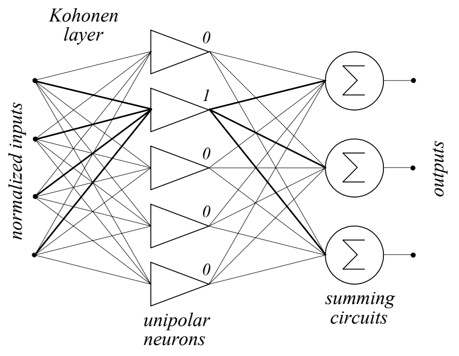


Fig. 17. Counterpropagation network

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

$$net = \mathrm{x}^t \mathrm{w} = \left(n - 2HD(\mathrm{x},\mathrm{w})\right) \tag{47}$$

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. In order that a neuron in the input layer is reacting just for the stored pattern, the threshold value for this neuron should be

$$w_{n+1} = -\left(n - 1\right) \tag{48}$$

If it is required that the neuron must react also for similar patterns then the threshold should be set to $w_{n+1} = -(n-(1+HD))$, where HD is the Hamming distance defining the range of similarity. 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 network, with unipolar activation functions in the first layer, works as a look up table. When the linear activation function (or no activation function at all) is used in the second layer, then the network also can be considered as an analog memory.

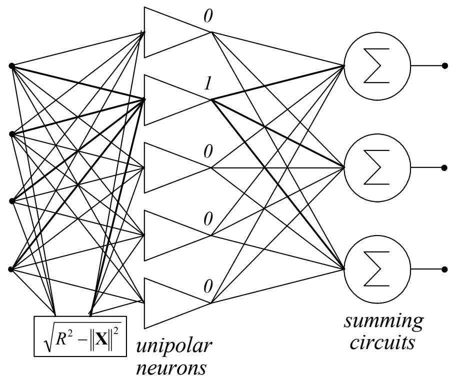


Fig. 18. Counterpropagation network used as analog memory with analog address

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. The counterpropagation network usually has more hidden neurons than required.

E. LVQ Learning Vector Quantization

At LVQ network the first layer detect subclasses. The second layer combines subclasses into a single class. First layer computes Euclidean distances between input pattern and stored patterns. Wining "neuron" is with the minimum distance

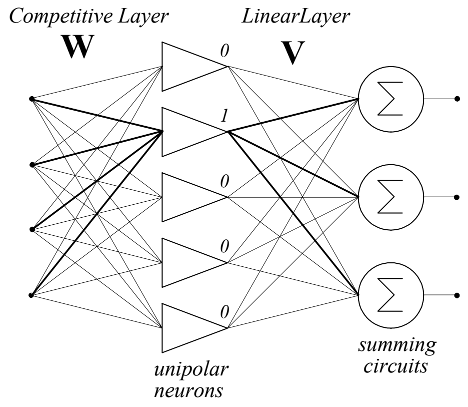


Fig. 19. LVQ Learning Vector Quantization

F. WTA architecture

The winner take all WTA network was proposed by Kohonen (1988). 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.

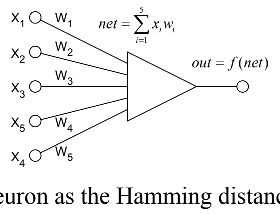


Fig. 20. Neuron as the Hamming distance clasifier

If inputs are binaries, for example **X**=[1, -1, 1, -1, -1] then the maximum value of *net*

$$net = \sum_{i=1}^{5} x_i w_i = \mathbf{X}\mathbf{W}^T \tag{49}$$

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:

$$net = \sum_{i=1}^{n} x_i w_i = \mathbf{X}\mathbf{W}^T = n - 2HD \qquad (50)$$

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

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

$$\|\mathbf{W}-\mathbf{X}\| = \sqrt{\sum_{i=1}^{n}(w_i-x_i)^2} \qquad (51)$$

$$\|\mathbf{W}-\mathbf{X}\| = \sqrt{\mathbf{W}\mathbf{W}^T - 2\mathbf{W}\mathbf{X}^T + \mathbf{X}\mathbf{X}^T} \qquad (52)$$

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

$$\|\mathbf{X}\| = 1 \quad \text{and} \quad \|\mathbf{W}\| = 1 \qquad (53)$$

Then the equation simplifies to

$$\|\mathbf{W}-\mathbf{X}\| = \sqrt{2 - 2\mathbf{W}\mathbf{X}^T} \qquad (54)$$

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

Kohonen WTA networks have some problems:

1. Important information about length of the vector is lost during the normalization process
2. Clustering depends from:
   a) Order patterns are applied
   b) Number of initial neurons
   c) Initial weights

G. Cascade correlation architecture

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

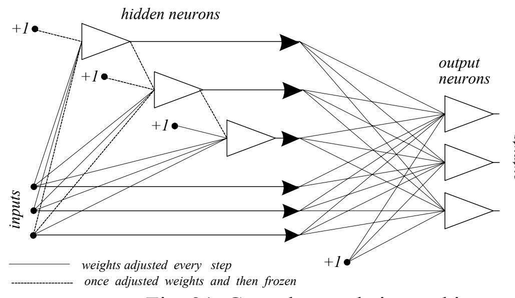

hidden neurons
+1
+1
+1
+1
output neurons
outputs
inputs
weights adjusted every step
once adjusted weights and then frozen

Fig. 21. 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

$$S = \sum_{o=1}^{O} \left| \sum_{p=1}^{P} \left(V_p - \overline{V}\right)\left(E_{po} - \overline{E_o}\right) \right| \qquad (55)$$

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

$$\Delta w_i = \sum_{o=1}^{O} \sum_{p=1}^{P} \sigma_o \left(E_{po} - \overline{E_o}\right) f_{p'}\ x_{ip} \qquad (56)$$

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.

H. RBF - Radial basis function networks

The structure of the radial basis network is shown in Fig. 22. 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.

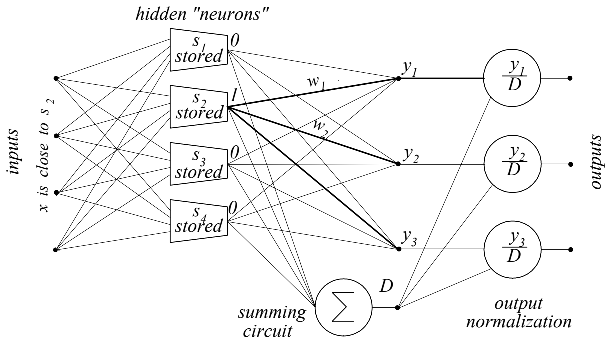


Fig. 22. *Radial basis function networks*

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

$$h_i = \exp\left(-\frac{\|\mathbf{x}-\mathbf{s_i}\|^2}{2\sigma^2}\right) \qquad (57)$$

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.

I. Input pattern transformation

The network shown in Fig. 23 has similar property (and power) like RBF networks, but it uses only traditional neurons with sigmoidal activation functions.

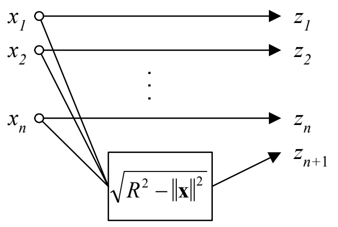
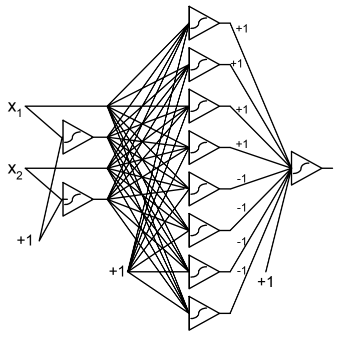


Fig. 23. Transfromation, which are required to give a traditional neurond the RBF properties.



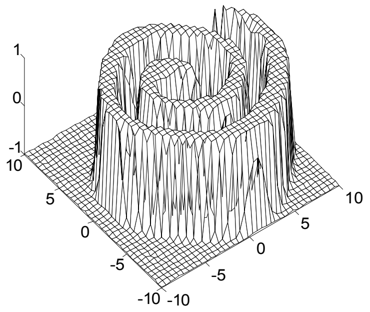


Fig. 24. Solution of two spirale problem using transformation from Fig. 23.

J. 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:

$$w_{i,j} = 1 \quad for \quad i,j = 1,2,\cdots N \tag{58}$$

$$w_{N+1,j} = 2j - N - 1 \quad for \quad j = 1,2,\cdots N \tag{59}$$

While weights for the output neurons are:

$$v_1 = 2\,\mathrm{mod}_2(N) - 1 \tag{60}$$

$$v_j = -v_{j-1} \quad for \quad j = 2,3,\cdots N \tag{61}$$

$$v_{N+1} = \mathrm{mod}_2(N) - 1 \tag{62}$$

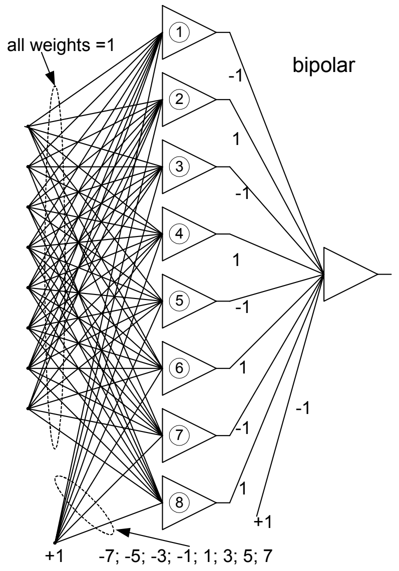


Fig. 25. Layered bipolar neural network with one hidden layer for the parity-8 problem.

For example, the architecture for the parity-8 problem with bipolar neurons is shown in Fig. 5. The same architecture can be used for a unipolar network. In this case:

$$w_{N+1,j} = 0.5 - j \quad for \quad j = 1,2,\cdots N$$
$$\overset{for\ N=8}{\Rightarrow} \left(-0.5, -1.5, -2.5, -3.5, -4.5, -5.5, -6.5, -7.5\right) \tag{63}$$

$$v_{N+1} = 0.5 - \mathrm{mod}_2(N) \quad \overset{for\ N=8}{\Rightarrow} \quad 0.5 \tag{64}$$

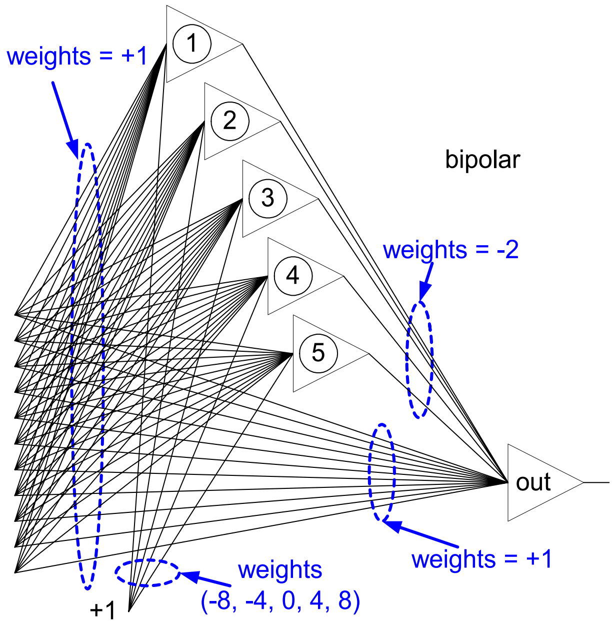


Fig. 26. Parity-11 implemented in fully connected bipolar neural networks with five neurons in the hidden layer.

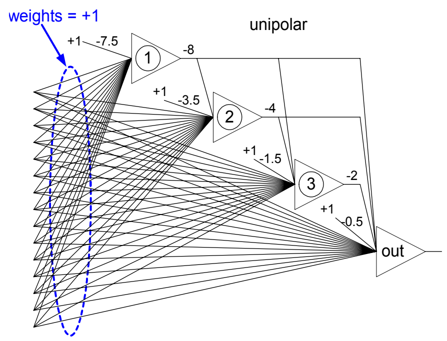


Fig. 27. Parity-15 implemented with 4 neurons in one cascade

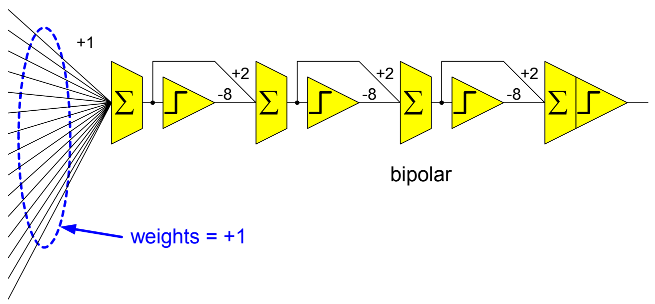


Fig. 28 Parity-15 implemented with 4 bipolar neurons in pipeline hybrid architecture with identical neurons and weights.

K. How to find proper neural network architecture?

1. Predict theoretically bye reducing number of inputs using statistical methods such as PCA
2. Use evolutionary computation technique
3. Use try and error approach
4. Start with large system and prune it, by combining (eliminate) neurons with the same similar (complement) responses for all patterns.
5. Start with smaller system and add neurons as needed. In this case criterion for adding neurons would be an oscillatory weight changes during training process.

K. Deficiency of statistical approaches

Statistical methods, such as correlation analisis or PC-Principle Component analysis wotks well only for linear or close to linear cases. Fig. 28 shows how different results can be obtained by changing only ther range of analysis.

VI. RECURENT 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.

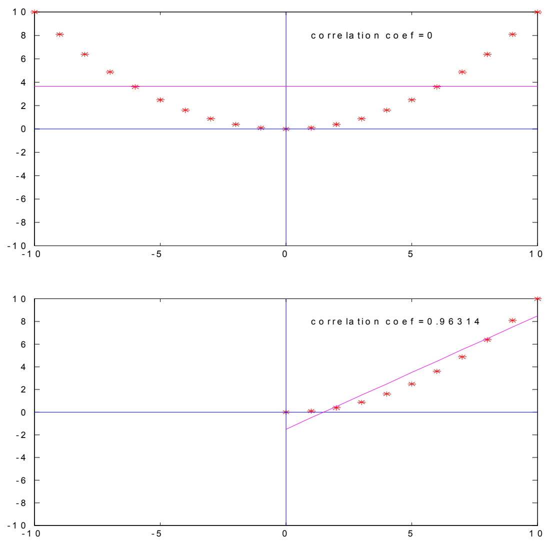


Fig. 28. Comparison of results of corelation analysis which lead to results range dependent.

A. Hopfield network

The single layer recurrent network was analyzed by Hopfield (1982). This network shown in Fig. 17 has unipolar hard threshold neurons with outputs equal to *0* or *1*. Weights are given by a symmetrical square matrix ***W*** with zero elements ($w_{ij} = 0 \text{ for } i=j$) on the main diagonal. The stability of the system is usually analyzed by means of the *energy function*

$$E = -\frac{1}{2}\sum_{i=1}^{N}\sum_{j=1}^{N} w_{ij} v_i v_j \qquad (65)$$

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

$$\Delta w_{ij} = \Delta w_{ji} = (2v_i - 1)(2v_j - 1) \qquad (66)$$

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.

B. Autoassociative memory

Hopfield (1984) extended the concept of his network to autoassociative memories. In the same network structure as shown in Fig. 17, 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

$$\mathbf{W} = \sum_{m=1}^{M} \mathbf{s}_m \mathbf{s}_m^T - M\,\mathbf{I} \qquad (67)$$

where M is the number of stored pattern, and $\boldsymbol{I}$ is the unity matrix. Note, that $\boldsymbol{W}$ 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.

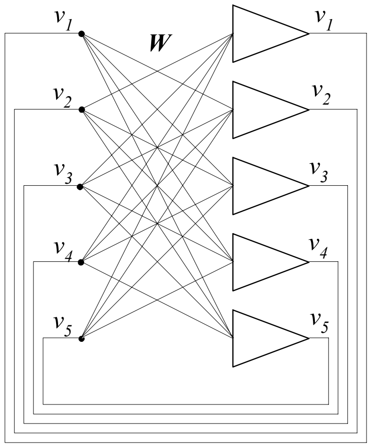


Fig. 29. Autoassociative memory

This stable point will be at the closest minimum of the energy function

$$E(\mathrm{v}) = -\frac{1}{2}\mathbf{v}^{\mathbf{T}}\mathbf{W}\mathbf{v} \tag{68}$$

Like 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.

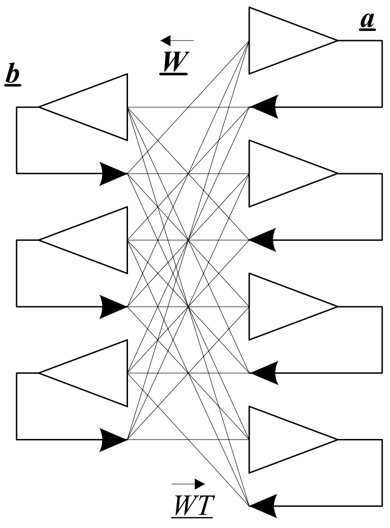


Fig. 30. BAM - Bidirectional Autoassociative Memory

C. BAM - Bidirectional Autoassociative Memories

The concept of the autoassociative memory was extended to bidirectional associative memories - BAM by Kosko (1987,1988). This memory shown in Fig. 18 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

$$\mathbf{W} = \sum_{m=1}^{M}\mathbf{a_m}\,\mathbf{b_m} \tag{69}$$

where M is the number of stored pairs, and $\boldsymbol{a_m}$ and $\boldsymbol{b}_m$ are the stored vector pairs. The BAM concept can be extended for association of three or more vectors.

REFERENCES

[1] S. Deutsch and A. Deutsch, Understanding the Nervous System: An Engineering Perspective, IEEE Press, Piscataway, NJ, 1993.

[2] M. L. Padgett, P. J. Werbos, and T. Kohonen, Strategies and Tactics for the Application of Neural Networks in Industrial Electronics, CRC Handbook for Industrial Electronics, J. D. Irwin Ed., CRC Press and IEEE Press, pp. 835-857, 1997.

[3] Zurada, J., Introduction to Artificial Neural Systems, West Publishing Company, 1992.

[4] Wilamowski, B. M. , R. C. Jaeger, and M. O. Kaynak, "Neuro-Fuzzy Architecture for CMOS Implementation" IEEE Transaction on Industrial Electronics, vol. 46, No. 6, pp. 1132-1136, Dec. 1999.

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

[6] Special Issue on New Trends in Neural Network Implementations in International Journal of Neural Systems (IJNS) Vol. 10, No. 3, June, 2000.

[7] Computational Intelligence Imitating Life edited by J. Zurada and others, IEEE Press 1994

[8] Silicon Implementation of Pulse Coded neural Networks edited by M. Zaghloul and others, Kluwer Academic 1994.

[9] Analog VLSI Signal and Information Processing edited by M. Ismail and T. Fiez, McGraw-Hill 1994

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

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