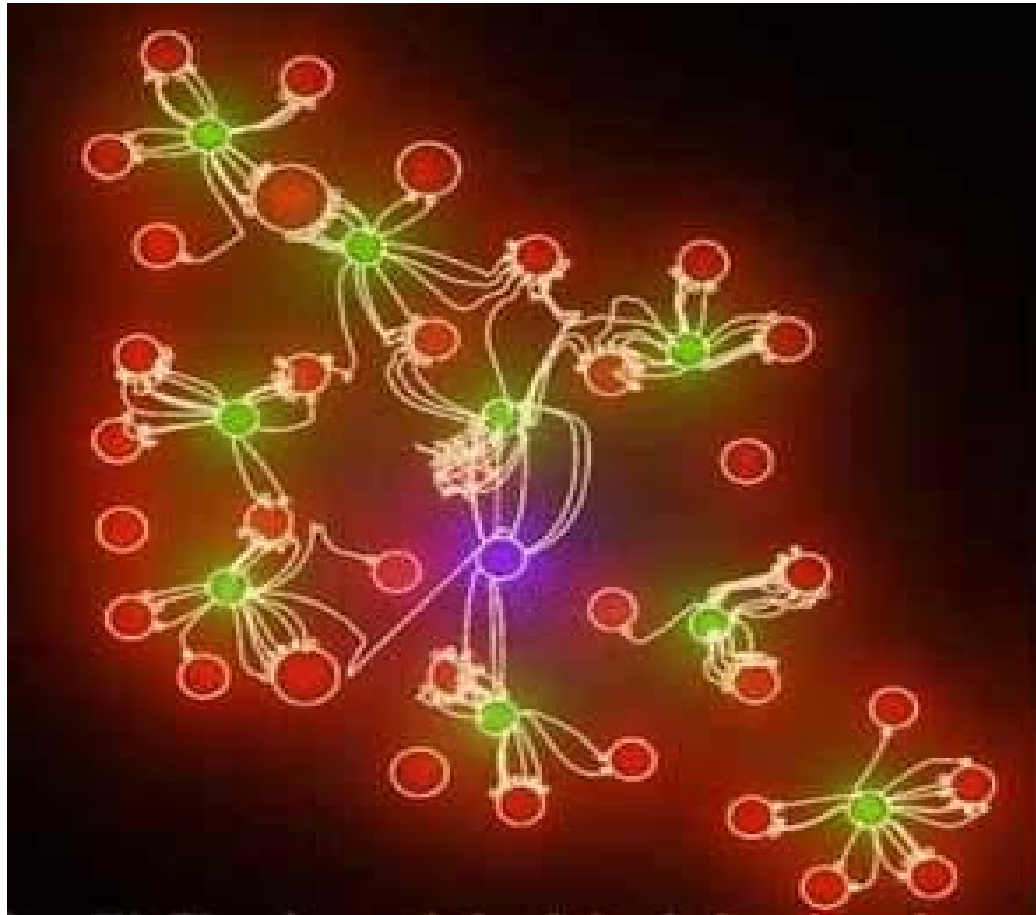


AS PER THE SYLLABUS OF BPUT FOR SEVENTH SEMESTER OF AE&IE BRANCH.

RIT,
BERHAMPUR

SOFT COMPUTING (PECS 3401)- NEURAL NETWORKS



Lecture Notes | KISHORE KUMAR SAHU

CHAPTER-04 NEURAL NETWORKS FUNDAMENTAL

INTRODUCTION

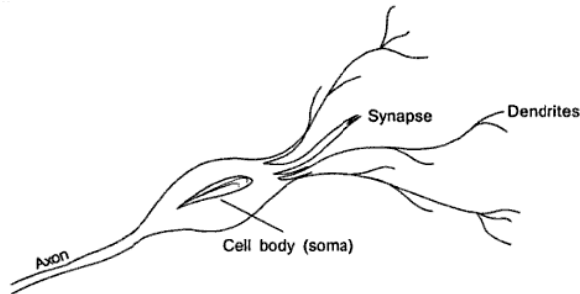
Neural networks are simplified models of biological neuron systems that have the ability to learn and acquire the knowledge and make it available for future use.

BIOLOGICAL NEURON

The basic unit of neuron systems is a *neuron*. The components of a neuron are as follows:

i. Dendrite (Synapse)

They act as the i/p channels to the neuron. They are responsible to feed the inputs to the neurons from the neighbouring neurons.



ii. Cell body (Soma)

They are the heart of neuron system. They are responsible to process or manipulate the inputs received from the synapses.

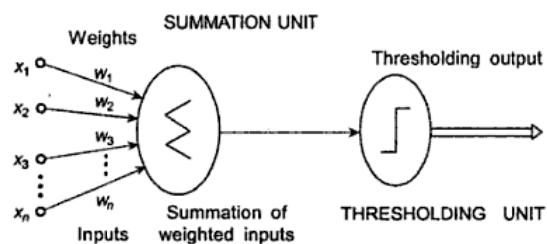
iii. Axon

They act as the o/p channel for the neuron. They are responsible for giving outputs to the neighbouring neurons.

ARTIFICIAL NEURON MODEL

They are the models that are inspired from the biological neurons. The components of *artificial neuron networks (ANN)* are similar to that of biological neuron.

i. Inputs



They are responsible for receiving inputs i.e. x_1, x_2, \dots, x_n and summing up then get the final input to the activation functions. The input to the neuron is the sum of the weighed inputs i.e. $I = w_1x_1 + w_2x_2 + \dots + w_nx_n = \sum_{i=1}^n w_ix_i$.

ii. Activation Function

They are responsible for checking whether the input signal is to sent to output of not. This is possible due to a threshold value present in the activation unit with which the inputs are always checked. If the inputs exceed the threshold value, then the input is propagated to the output other not

$$\text{i.e. } y = \phi(I) = \phi(\sum_{i=1}^n w_ix_i - \theta) = \begin{cases} 1 & I > \theta \\ 0 & I < \theta \end{cases}$$

iii. Outputs

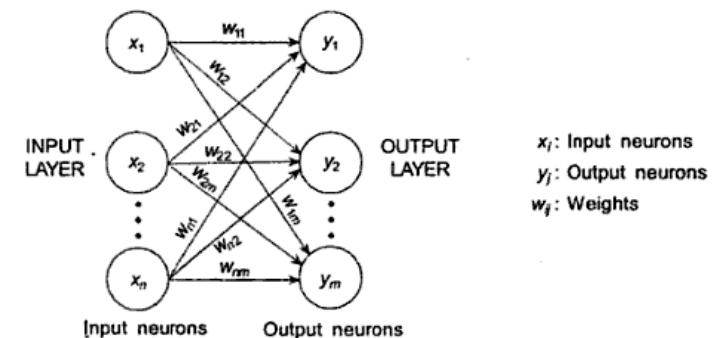
They are responsible for receiving the output from the activation functions depending on the decision taken by the activation function.

ARCHITECTURE OF NEURAL NETWORKS

An artificial neural networks is defined as a data processing system consisting of a large number of simple highly interconnected processing elements (ANN) is an architecture inspired by the structure of the cerebral cortex of the brain. Generally, an ANN structure can be represented using a *directed graph*. Since the flow of signals is always restricted in one direction only so we make use of directed graphs, where the vertices represents *neurons*, and the edges represents the synaptic links. Broadly there are three classes of neural networks as follows.

i. Single layer feed-forward networks

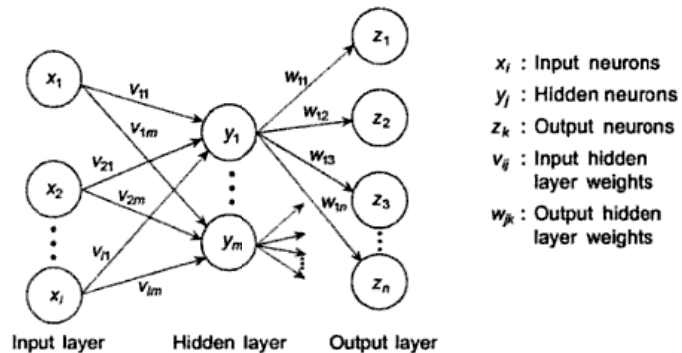
This type of networks comprises of two layers, namely the *input* and *output layer*. The input layers receive the input signal and the output layers receive the output signals. The synaptic links carrying the



weights connect the every input layer to the output neuron but not vice-versa. Such a networks is said to be *feed-forward* in type or acyclic in nature. Despite two layer the network is called as single layer as it is the output layer alone that performs the computation. The input layer just pass the signals to output layer.

ii. Multi layer feed-forward networks

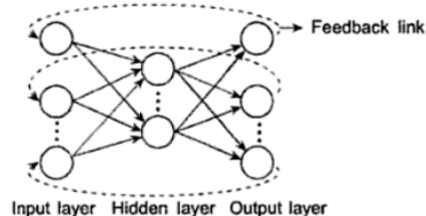
This architecture is called as multi layer NN due to the presence of a number of intermediate layers called as *hidden layer*. The unit of hidden layer is called as *hidden neurons* or *hidden units*. These hidden layer are responsible for performing the processing of input signals along with the output layer. They are connected with input layer with weight called as *input-hidden layer weight* and connect with output layer with *hidden-output layer weight*. If a multi-layered NN is having 3 input neuron, 5 hidden neuron in one layer and in the other, and 2 in the output layer. Then this multi layered neural is represented as 3-5-4-2.



These hidden layer are responsible for performing the processing of input signals along with the output layer. They are connected with input layer with weight called as *input-hidden layer weight* and connect with output layer with *hidden-output layer weight*. If a multi-layered NN is having 3 input neuron, 5 hidden neuron in one layer and in the other, and 2 in the output layer. Then this multi layered neural is represented as 3-5-4-2.

iii. Recurrent networks

Networks in this class has at least one feedback loop. This is what make them different from other networks. It is also possible that the neurons will also have a self loop i.e. feed back into itself.



LEARNING METHODS

Leaning methods in Neural Networks can be broadly classified into three basic types: *supervised*, *unsupervised* and *reinforced*.

i. **Supervised Learning:** In this, every input pattern that is used to train the network is associated with an output pattern, which is the target or the desired pattern. A teacher is assumed to be present during the learning process, when a comparison is made between the network's computed output and the correct expected output, to determine the error. The error can then be used to change network parameters, which result in an improvement in performance.

ii. **Gradient descent learning:** This is based on the minimization of error E defined in terms of weights and the activation function of the network. Also, it is required that the activation function employed by the network is differentiable, as the weight update is dependent on the gradient of the error E . Thus, if ΔW_{ij} is the weight update of the link connecting the i th and j th neuron of the two neighbouring layers, then $\Delta W_{ij} = \eta \frac{\partial E}{\partial W_{ij}}$, where η is the leaning rate parameter and $\frac{\partial E}{\partial W_{ij}}$ is the error gradient with reference to the weight W_{ij}

ii. **Stochastic learning:** In this method, weights are adjusted in a probabilistic fashion. An example is evident in simulated annealing-the learning mechanism employed by Boltzmann and Cauchy machines, which are a kind of NN systems.

ii. **Unsupervised Learning:** in this learning method, the target output is not present to the network. It is as if there is no teacher to present the desired patterns and hence, the system learns of its own by discovering and adapting to structural features in the input patterns.

ii.i. **Hebbian learning:** It is based on correlative weight adjustment. This is the oldest leaning mechanism inspired by biology. In this, the input-output pattern pair (X_i, Y_j) are associated b the weight matrix W , known as the correlation matrix. It is computed as $W = \sum_{i=0}^n X_i Y_i^T$. Here Y_i^T is the transpose of associated output vector Y_i .

ii.ii. **Competitive Learning:** In this method, those neurons which respond strongly to input stimuli have their weights updated. When an input pattern is presented, all neurons in the layer compete and the winning neurons undergoes weight adjustment. Hence, it is a "winner-takes-all" strategy.

iii. **Reinforced Learning:** In this method, a teacher though available, does not present the expected answer but only indicated if the computed output is correct or incorrect. The information provided helps the network in its

learning process. A reward is given for a correct answer computed and a penalty for a wrong answer. But, reinforced learning is not one of the popular forms of learning.

Neural networks have shown remarkable progress in the recognition of visual images, handwritten characters, printed characters, speech and other PR based tasks.

ii. Optimization/ constrain satisfaction:

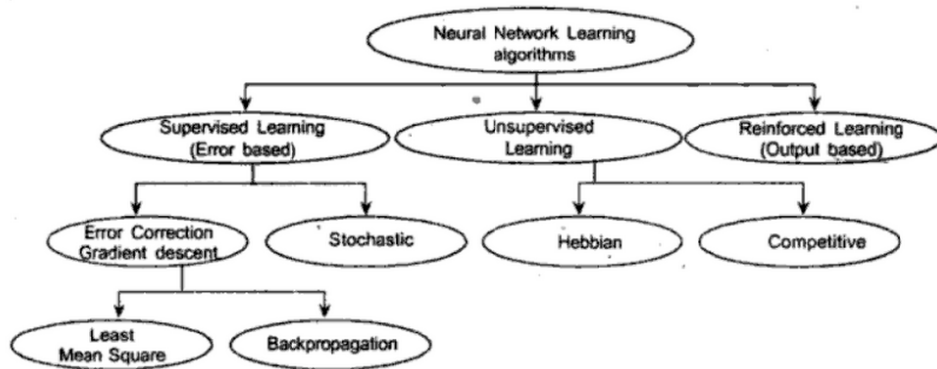
This comprises problems which need to satisfy constraints and obtain optimal solutions. Examples of such problems include manufacturing scheduling, finding the shortest possible tour given a set of cities, etc. Several problems of this nature arising out of industrial and manufacturing fields have found acceptable solutions using NNs.

iii. Forecasting and risk assessment:

Neural networks have exhibited the capability to predict situation from past trends. They have, therefore, found ample applications in areas such as meteorology, stock market, banking, and econometrics with high success rates.

iv. Control systems:

Neural networks have gained commercial ground by finding applications in control systems. Dozens of computer products, especially, by the Japanese companies incorporating NN technology, is a standing example. Besides they have also been used for the control of chemical plants, robots and so on.



CHARACTERISTICS OF NEURAL NETWORKS

- i. The NNs exhibit mapping capabilities, that is, they can map input pattern to their associated output patterns.
- ii. The NNs learn by examples. Thus, NN architecture can be *trained* with known examples of a problem before they are tested for their *inference* capability on unknown instance of the problem. They can identify new objects previously untrained.
- iii. The NNs possess the capability to generalize. This they can predict new outcomes form past trends.
- iv. The NNs are robust systems and are fault tolerant. They therefore, recall full patterns form incomplete, partial or noisy patterns.
- v. The NNs can process information in parallel, at high speed, and in a distributed manner.

SOME APPLICATION DOMAINS

Neural networks have successfully applied for the solution of a variety of problems. Some of them of are listed below:

- i. **Pattern recognition (PR)/image processing:**

CHAPTER-05 BACKPROPAGATION NETWORKS

ACTIVATION FUNCTIONS

To generate the final output y , the sum input ($I = w_1x_1 + w_2x_2 + \dots + w_nx_n = \sum_{i=1}^n w_ix_i$) is passed on to a non-linear filter ϕ called *activation function/ transfer function/ squash function* which releases the output. i.e. $y = \phi(I)$.

A very commonly used activation function is the *Thresholding function*. In this, the sum is compared with a threshold value θ . If the value of I is greater than θ , then the output is 1 else it is 0. i.e. $y = \phi(\sum_{i=1}^n w_ix_i - \theta)$, where, ϕ is the *step function* known as *Heaviside function* and is such that $y = \phi(I) = \begin{cases} 1, I > \theta \\ 0, I \leq \theta \end{cases}$.

Signum function

Also known as the *quantizer* function, defined as $y = \phi(I) = \begin{cases} +1, I > \theta \\ -1, I \leq \theta \end{cases}$.

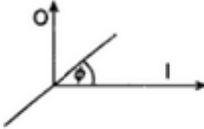
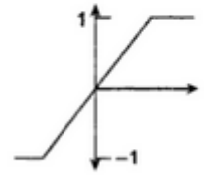

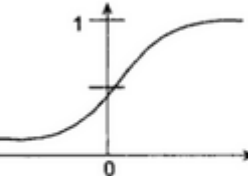
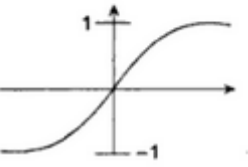
Sigmoidal function

This function is a continuous function that varies gradually between the asymptotic values 0 and 1 or -1 and +1 is given by: $\phi(I) = \frac{1}{1+e^{-\alpha I}}$. Where α is the slope parameter, which adjust the abruptness of the function as it changes between the two asymptotic values. Sigmoidal functions are differentiable, which is an important feature of NN theory.

Hyperbolic tangent function

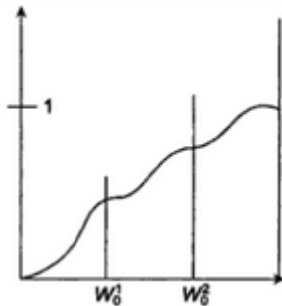
The function is given by $\phi(I) = \tanh(I)$ and can produce negative outputs values.

The other activation functions are *Radial Basis function, Unipolar Multimodal, Piecewise Linear, Hard Limiter, Unipolar Sigmoidal and Bipolar Sigmoidal*.

Type	Equation	Functional form
Linear	$O = gI$ $g = \tan \phi$	
Piecewise Linear	$O = \begin{cases} 1 & \text{if } mI > 1 \\ gI & \text{if } mI < 1 \\ -1 & \text{if } mI < -1 \end{cases}$	
Hard Limiter	$O = \text{sgn } [I]$	
Unipolar Sigmoidal	$O = \frac{1}{1 + \exp(-\lambda I)}$	
Bipolar Sigmoidal	$O = \tanh [\lambda I]$	

Unipolar Multimodal

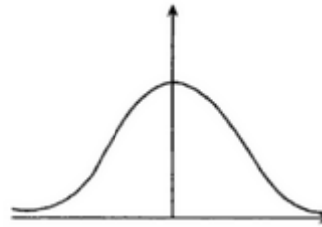
$$O = \frac{1}{2} \left[1 + \frac{1}{M} \sum_{m=1}^M \tanh (g^m (I - W_0^m)) \right]$$



Radial Basis Function (RBF)

$$O = \exp(I)$$

$$I = \left[\frac{-\sum_{i=1}^N (W_i(t) - X_i(t))^2}{2\sigma^2} \right]$$



SINGLE LAYER PERCEPTRON

A single layer perceptron consists of an input layer and an output layer. Here we have a perceptron with two input x_1 and x_2 as inputs with weights w_1 and w_2 and a bias x_0 with weight w_0 . The output of the perceptron is calculated as the sum of weighted inputs and bias i.e. $net = w_0 + w_1x_1 + w_2x_2$. This represents the equation of a straight line, which makes clear that single layer perceptron are responsible of representing problems that are linearly separable. Since logic gates corresponds to problems that are linearly separable, hence can be easily simulated by single layer perceptron.

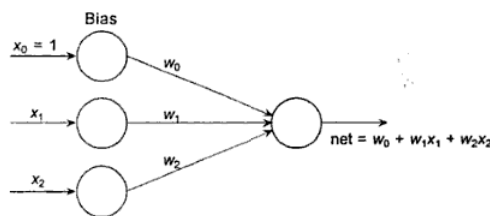
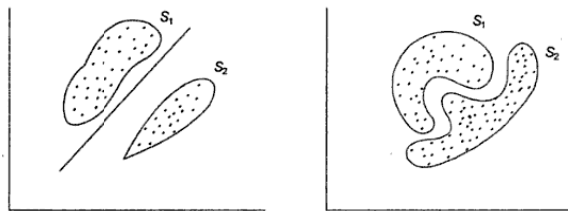


Table 2.3 XOR truth table

Inputs	Inputs	Output
0	0	0
1	1	0
0	1	1
1	0	1



(a) Linearly separable patterns (b) Non-linearly separable patterns

Algorithm for training a single layer perceptron

- Step 1. Initialize the weights and bias to zero. Also set the learning rate α in between 0-1.
- Step 2. While stopping condition fails do steps-3 to step-7.
- Step 3. For each training pair do step-4 to step-6.
- Step 4. Set the activation of i/p.
- Step 5. Compute the net i/p to the activation function $y_{in} = b + (\sum_{i=1}^n w_i x_i)$. Activation function is used to compute the o/p $y_k = f(y_{in}) = \begin{cases} +1, y_{in} > 0 \\ -1, y_{in} \leq 0 \end{cases}$.
- Step 6. If o/p and target are not equal then change the weights and bias as follows: $w_i(new) = w_i(old) + \alpha T_k x_i$ and $b_i(new) = b_i(old) + \alpha T_k$.
else $w_i(new) = w_i(old)$ and $b_i(new) = b_i(old)$.
- Step 7. Test for stopping condition.

Example: Train a single layer perceptron for an AND gate.

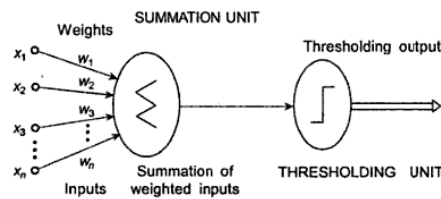
The training set for AND gate is as follows:

x_1	x_2	T
-1	-1	-1
1	-1	-1
-1	1	-1
1	1	1

Input			Net	Output	Target	Weight Change			Updated Weight		
x_1	x_2	b	y_{in}	y	T	Δw_1	Δw_2	Δb	w_1	w_2	b
-1	-1	0	0	0	-1	1	1	-1	1	1	-1
1	-1	0	0	0	-1	-1	1	-1	-1	1	-1
-1	1	0	0	0	-1	1	-1	-1	1	-1	-1
1	1	0	0	0	1	1	1	1	1	1	1
-1	-1	-1	-3	-1	-1	Training is successful since the output of the single layer perceptron is equal to the target output.					
1	-1	-1	-3	-1	-1						
-1	1	-1	-3	-1	-1						
1	1	1	3	1	1						

ADALINE NETWORKS

ADA line networks (*adaptive linear neuron networks*) are similar to that of single layer perceptron. All the components are similar to that of a single layer perceptron. So we can conclude the ADA line networks are same as single layer perceptron, except the in the training algorithm we have the update equation are as follows: $w_i(new) = w_i(old) + \alpha(y_{in} - T_k)x_i$ and $b_i(new) = b_i(old) + \alpha(y_{in} - T_k)$. i.e. T_k is replaced with the error $(y_{in} - T_k)$.



Algorithm for training a single layer perceptron

- Step 1. Initialize the weights and bias to zero. Also set the learning rate α in between 0-1.
- Step 2. While stopping condition fails do steps-3 to step-7.
- Step 3. For each training pair do step-4 to step-6.
- Step 4. Set the activation of i/p.
- Step 5. Compute the net i/p to the activation function $y_{in} = b + (\sum_{i=1}^n w_i x_i)$. Activation function is used to compute the o/p $y_k = f(y_{in}) = \begin{cases} +1, & y_{in} > 0 \\ -1, & y_{in} \leq 0 \end{cases}$
- Step 6. If o/p and target are not equal then change the weights and bias as follows:

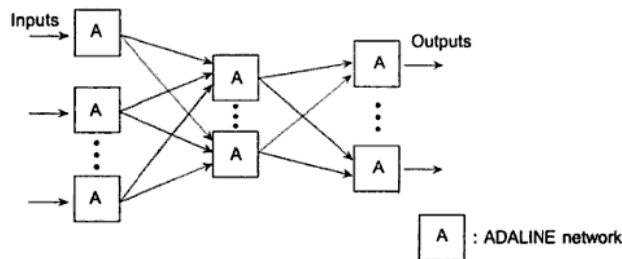
$$w_i(new) = w_i(old) + \alpha(y_{in} - T_k)x_i$$
 and

$$b_i(new) = b_i(old) + \alpha(y_{in} - T_k)$$
 else

$$w_i(new) = w_i(old) \text{ and } b_i(new) = b_i(old)$$
- Step 7. Test for stopping condition.

MADALINE NETWORKS

The MADA line networks can handle linearly separable problem. This is a network which is composed of several constituent ADA line network or single layer perceptron. This network is called as MADA line due to reason that is has a number of hidden layers.



Algorithm for MADA line

- Step 1. Initialize the i/p and hidden neuron layer weights and bias to small random values and hidden layer and o/p layer weights and bias to 0.5.
- Step 2. While stopping condition fails do steps-3 to step-9.
- Step 3. For each training pair do step-4 to step-8.
- Step 4. Set the activation of i/p units.
- Step 5. Compute the net i/p to the hidden units $z_{in(i)} = b_i + (\sum_{i=1}^n w_i x_i)$.
- Step 6. Activation function is used to compute hidden unit output, $z_i = f(z_{in(i)})$.
- Step 7. Calculate the net i/p for o/p neuron $y_{in(i)} = b_i + (\sum_{i=1}^n z_i x_i)$ and apply activation function for each $y_i, y_i = f(y_{in(i)})$.
- Step 8. If o/p and target are not equal then change the weights and bias as follows:

$$w_i(new) = w_i(old) + \alpha(y_{in} - T_k)x_i$$
 and

$$b_i(new) = b_i(old) + \alpha(y_{in} - T_k)$$
 else

$$w_i(new) = w_i(old) \text{ and } b_i(new) = b_i(old)$$
- Step 9. Test for stopping condition.

Example: perform the training process with MADA line with 2 to 1 for X-OR.

TRAINING X-OR BY MADALINE ALGORITHM

x1	x2	w1	w2	b1	yin	y	T	Δw1	Δw2	Δb1	w1n	w2n	b1n
-1	-1	0.43	0.43	0.5	-0.36	-1	-1	0	0	0	0.43	0.43	0.5
1	1	0.43	0.43	0.5	1.36	1	-1	-0.6	0	-0.6	-0.17	0.43	-0.1
-1	1	0.43	0.43	0.5	0.5	1	1	0	0	0	0.43	0.43	0.5
1	-1	0.43	0.43	0.5	0.5	1	1	0	0	0	0.43	0.43	0.5

x1	x2	w1	w2	b1	yin	y	T	Δw1	Δw2	Δb1	w1n	w2n	b1n
-1	-1	0.43	0.43	0.5	-0.36	-1	-1	0	0	0	0.43	0.43	0.5
1	1	-0.17	0.43	0.5	0.76	1	-1	-0.6	0	-0.6	-0.77	0.43	-0.1
-1	1	0.43	0.43	0.5	0.5	1	1	0	0	0	0.43	0.43	0.5
1	-1	0.43	0.43	0.5	0.5	1	1	0	0	0	0.43	0.43	0.5

x1	x2	w1	w2	b1	yin	y	T	Δw1	Δw2	Δb1	w1n	w2n	b1n
-1	-1	0.43	0.43	0.5	-0.36	-1	-1	0	0	0	0.43	0.43	0.5
1	1	-0.77	0.43	0.5	0.16	1	-1	-0.6	0	-0.6	-1.37	0.43	-0.1
-1	1	0.43	0.43	0.5	0.5	1	1	0	0	0	0.43	0.43	0.5
1	-1	0.43	0.43	0.5	0.5	1	1	0	0	0	0.43	0.43	0.5

x1	x2	w1	w2	b1	yin	y	T	Δw1	Δw2	Δb1	w1n	w2n	b1n
-1	-1	0.43	0.43	0.5	-0.36	-1	-1	0	0	0	0.43	0.43	0.5
1	1	-1.37	0.43	0.5	-0.44	-1	-1	0	0	0	-1.37	0.43	0.5
-1	1	0.43	0.43	0.5	0.5	1	1	0	0	0	0.43	0.43	0.5
1	-1	0.43	0.43	0.5	0.5	1	1	0	0	0	0.43	0.43	0.5

BACK PROPAGATION NEURAL NETWORKS

A multilayer feed forward back propagation neural network with one layer of z hidden units is shown in the figure. The layer having the bias from $w_{01}, w_{02}, \dots, w_{0m}$ and hidden neuron having the bias $v_{01}, v_{02}, \dots, v_{0m}$. The figure only feed forward network is shown but during the back propagation phase of learning the signals are sent in the reverse direction, i.e. from o/p to i/p layer.

The training algorithm of back propagation involves four steps:

Initialization of weights and bias- The bias and the weights are chosen as small numbers.

Feedforward-Each i/p pair receives the i/p and transmits to the hidden layer. Each hidden unit then calculates the o/p and transmits this signal to the o/p units.

Back propagation of error-Each o/p is compared with target o/p and the determines the error. Based on the error the factor δ_k is computed and is used to distribute the error at o/p unit y_k back to all units in the previous layer. Similarly the factor δ_j is computed for each hidden unit.

Update the weights and the bias- The weights and the bias are updated using the δ -factor and activation. $\delta = (t_k - y_k) f'(y_k)$ (Generalized δ rule of learning).

Algorithm for Back propagation

- Step 1. Initialize weights and bias to small random values.
 Step 2. While stopping condition fails do steps-3 to step-10.
 Step 3. For each training pair do step-4 to step-9.
 Step 4. Each i/p unit receives the signal x_i and transmits this signal to all other units.
 Step 5. Compute the net i/p to the hidden units $z_{in(j)} = v_{0j} + (\sum_{i=1}^n v_{ij} \cdot x_i)$. Activation function is used to compute the o/p of hidden units, $z_j = f(z_{in(j)})$.

- Step 6. Calculate the net i/p for o/p neuron $y_{in(k)} = w_{ok} + (\sum_{j=1}^n w_{jk} \cdot z_j)$ and apply activation function for each $y_k, y_k = f(y_{in(k)})$.
 Step 7. For each y_k calculate the $\delta_k = (t_k - y_k) f'(y_{in(k)}) = (t_k - y_k) \cdot f'(y_{in(k)}) \cdot (1 - f'(y_{in(k)}))$.
 Step 8. Each hidden unit sums its δ i/p from the previous layer and o/p is given as: $\delta_{in(j)} = \sum_{k=1}^m \delta_k \cdot w_{jk}$. The error information is calculated as $\Delta_j = \delta_{in(j)} \cdot f'(z_{in(j)})$.
 Step 9. Each of the o/p unit update its bias and weights as given below. $w_{jk}(new) = w_{jk}(old) + \Delta w_{jk}$ where $\Delta w_{jk} = \alpha \delta_k z_j$ and bias $w_{ok}(new) = w_{ok}(old) + \Delta w_{ok}$ where $\Delta w_{ok} = \alpha \delta_k$. Each hidden unit update its weights and bias as follows $v_{ij}(new) = v_{ij}(old) + \Delta v_{ij}$ where $\Delta v_{ij} = \alpha \delta_j x_i$ and bias $v_{0j}(new) = v_{0j}(old) + \Delta v_{0j}$ where $\Delta v_{0j} = \alpha \delta_j$.
 Step 10. Test for stopping condition.

Merits of Back propagation Algorithm

1. The mathematical formulation is so compatible for any kind of networks.
2. Multilayer neural network trained with back propagation algorithm has got a greater representation capability. Any non linear activation function.
3. It requires good set of training data. It can tolerate noise and missing data in training sample.
4. Easy to implement.
5. The computation time is reduced if the weights chosen are small at the beginning stage.
6. Wider application.
7. Can be used to store a huge amount of pattern.
8. The batch update of weights exists, which provides a smoothing effect on weight corrections.

Demerits of Back propagation Algorithm

1. Learning often takes longer time to converge.
2. Complex functions requires more iterations.
3. Gradient descent method used in back propagation algorithm gives guarantee to minimize error at local minima.
4. The network may be trapped in a local minima, though a better solution is available nearby.
5. The training may sometimes cause temporal instability to the system.

Example: Find the new weights for a network with i/p pattern [0.6 0.8 0] and target o/p is 0.9. The i/p hidden weights are $v = \begin{bmatrix} 2 & 1 & 0 \\ 1 & 2 & 2 \\ 0 & 3 & 1 \end{bmatrix}$ and hidden o/p weights are $w = [-1 \ 1 \ 2]^T$. $v_{03} = -1$ and $w_{01} = -1$. The learning rate = 0.3 and use binary sigmoid activation function.

Solution:

$$f(I) = \frac{1}{1+e^{-aI}}; z_{in(j)} = v_{oj} + \left(\sum_{i=1}^n v_{ij} \cdot x_i\right) \text{ and } z_j = f(z_{in(j)}) = \frac{1}{1+e^{-az_{in(j)}}}$$

$$z_{in(1)} = v_{o1} + \left(\sum_{i=1}^n v_{i1} \cdot x_i\right) = 0 + (0.6*2 + 0.8*1 + 0*0) = 2;$$

$$z_{in(2)} = v_{o2} + \left(\sum_{i=1}^n v_{i2} \cdot x_i\right) = 0 + (0.6*1 + 0.8*2 + 0*3) = 2.2;$$

$$z_{in(3)} = v_{o3} + \left(\sum_{i=1}^n v_{i3} \cdot x_i\right) = -1 + (0.6*0 + 0.8*2 + 0*1) = 0.6;$$

$$z_1 = f(z_{in(1)}) = \frac{1}{1+e^{-az_{in(1)}}} = \frac{1}{1+e^{-0.3*2}} = \frac{1}{1+e^{-0.6}} = 0.645;$$

$$z_2 = f(z_{in(2)}) = \frac{1}{1+e^{-az_{in(2)}}} = \frac{1}{1+e^{-0.3*2.2}} = \frac{1}{1+e^{-0.66}} = 0.659;$$

$$z_3 = f(z_{in(3)}) = \frac{1}{1+e^{-az_{in(3)}}} = \frac{1}{1+e^{-0.3*0.6}} = \frac{1}{1+e^{-0.18}} = 0.5448;$$

$$y_{in(k)} = w_{ok} + \left(\sum_{i=1}^n w_{ik} \cdot z_i\right) \text{ and } y_k = f(y_{in(k)}) = \frac{1}{1+e^{-ay_{in(k)}}}$$

$$y_{in(1)} = w_{o1} + \left(\sum_{i=1}^n w_{i1} \cdot z_i\right) = -1 + (0.645*-1 + 0.659*1 + 0.5448*2) = 0.1;$$

$$y_1 = f(y_{in(1)}) = \frac{1}{1+e^{-ay_{in(1)}}} = \frac{1}{1+e^{-0.3*0.1}} = \frac{1}{1+e^{-0.03}} = 0.50749 \approx 0.5075;$$

$$\delta_k = (t_k - y_k) f'(y_{in(k)}) = (t_k - y_k) f(y_{in(k)}) \cdot (1 - f(y_{in(k)}))$$

$$\delta_1 = (0.9 - 0.5075) * 0.5075 * (1 - 0.5075) = 0.0977.$$

$$\text{Error at hidden layers } \delta_{in(j)} = \sum_{k=1}^m \delta_j \cdot w_{jk};$$

$$\delta_{in(1)} = \sum_{k=1}^m \delta_1 \cdot w_{11} = 0.0977 * -1 = -0.0977;$$

$$\delta_{in(2)} = \sum_{k=1}^m \delta_1 \cdot w_{12} = 0.0977 * 1 = 0.0977;$$

$$\delta_{in(3)} = \sum_{k=1}^m \delta_1 \cdot w_{13} = 0.0977 * 2 = 0.1954;$$

$$\text{Error term at hidden layer: } \Delta_j = \delta_{in(j)} \cdot f'(z_{in(j)}) = \delta_{in(j)} \cdot f(z_{in(j)}) (1 - f(z_{in(j)}));$$

$$\Delta_1 = \delta_{in(1)} \cdot f(z_{in(1)}) (1 - f(z_{in(1)})) = -0.0977 * 0.645 * (1 - 0.645) = -0.01802;$$

$$\Delta_2 = \delta_{in(2)} \cdot f(z_{in(2)}) (1 - f(z_{in(2)})) = 0.0977 * 0.659 * (1 - 0.659) = 0.01769;$$

$$\Delta_3 = \delta_{in(3)} \cdot f(z_{in(3)}) (1 - f(z_{in(3)})) = 0.1954 * 0.5448 * (1 - 0.5448) = 0.03906;$$

Weight Updating

Change in weight (i/p-hidden layer)

$$\Delta v_{ij} = \alpha \Delta_i x_i$$

$$\Delta v_{11} = \alpha \Delta_1 x_1 = 0.3 * -0.02238 * 0.6 = -0.00324;$$

$$\Delta v_{12} = \alpha \Delta_2 x_1 = 0.3 * -0.01769 * 0.6 = 0.00318;$$

$$\Delta v_{13} = \alpha \Delta_3 x_1 = 0.3 * 0.03906 * 0.6 = 0.000703;$$

$$\Delta v_{21} = \alpha \Delta_1 x_2 = 0.3 * -0.02238 * 0.8 = 0.00432;$$

$$\Delta v_{22} = \alpha \Delta_2 x_2 = 0.3 * 0.01769 * 0.8 = 0.00424;$$

$$\Delta v_{23} = \alpha \Delta_3 x_2 = 0.3 * 0.03906 * 0.8 = 0.000937;$$

$$\Delta v_{31} = \alpha \Delta_1 x_3 = 0.3 * -0.02238 * 0 = 0;$$

$$\Delta v_{32} = \alpha \Delta_2 x_3 = 0.3 * 0.01769 * 0 = 0;$$

$$\Delta v_{33} = \alpha \Delta_3 x_3 = 0.3 * 0.03906 * 0 = 0;$$

New weights are as follows $v_{ij}(\text{new}) = v_{ij}(\text{old}) + \Delta v_{ij}$

$$v_{\text{new}} = \begin{bmatrix} 2 - 0.00324 & 1 + 0.00318 & 0 + 0.000703 \\ 1 + 0.00432 & 2 + 0.00424 & 2 + 0.000937 \\ 0 + 0 & 3 + 0 & 1 + 0 \end{bmatrix} = \begin{bmatrix} 0 & 0 & 0 \\ 0 & 0 & 0 \\ 0 & 0 & 0 \end{bmatrix}$$

Change in bias (hidden layer): $\Delta v_{oj} = \alpha \Delta_j$

$$\Delta v_{o1} = \alpha \Delta_1 = 0.3 * -0.02238 = -0.0054;$$

$$\Delta v_{o2} = \alpha \Delta_2 = 0.3 * -0.01769 = 0.0053;$$

$$\Delta v_{o3} = \alpha \Delta_3 = 0.3 * 0.03906 = 0.01117;$$

New bias are $v_{oj}(new) = v_{oj}(old) + \Delta v_{oj}$;

$$v_o = [0 - 0.0054 \ 0 + 0.0053 \ -1 + 0.01117] = [-0.0054 \ 0.0053 \ -0.99883].$$

Change in the weights for hidden-o/p layer: $\Delta w_{jk} = \alpha \delta_k z_j$

$$\Delta w_{11} = \alpha \delta_1 z_1 = 0.3 * 0.0977 * 0.645 = 0.01526;$$

$$\Delta w_{21} = \alpha \delta_1 z_2 = 0.3 * 0.0977 * 0.659 = 0.01558;$$

$$\Delta w_{31} = \alpha \delta_1 z_3 = 0.3 * 0.0977 * 0.5448 = 0.01287;$$

New weights for the hidden-o/p layer: $w_{jk}(new) = w_{jk}(old) + \Delta w_{jk}$

$$w = [-1 + 0.01526 \ 1 + 0.01558 \ 2 + 0.01287] = [-0.98474 \ 1.01558 \ 2.01287].$$

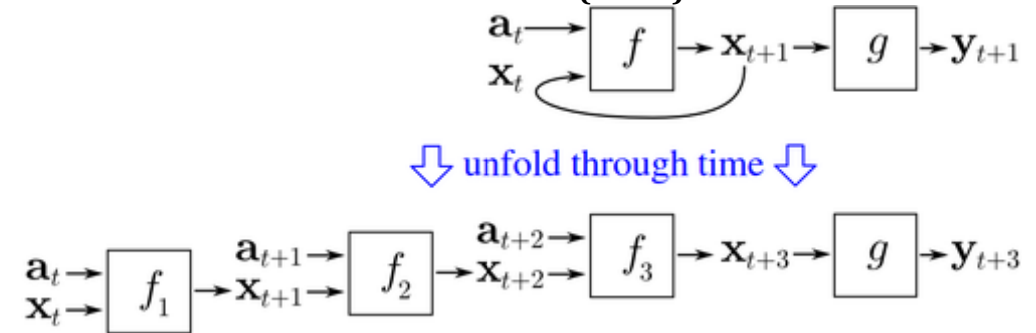
Change in bias for o/p layer: $\Delta w_{ok} = \alpha \delta_k$

$$\Delta w_{o1} = \alpha \delta_1 = 0.3 * 0.0977 = 0.023;$$

New bias for output layer $w_{ok}(new) = w_{ok}(old) + \Delta w_{ok}$.

$$w_{o1}(new) = w_{o1}(old) + \Delta w_{o1} = -1 + 0.023 = -0.976.$$

BACK PROPAGATION THROUGH TIME (BPTT) ALGORITHM



To train a recurrent neural network using BPTT, some training data is needed. This data should be an ordered sequence of input-output pairs,

$$\langle \langle \mathbf{a}_0, \mathbf{y}_0 \rangle, \langle \mathbf{a}_1, \mathbf{y}_1 \rangle, \langle \mathbf{a}_2, \mathbf{y}_2 \rangle, \dots, \langle \mathbf{a}_{n-1}, \mathbf{y}_{n-1} \rangle \rangle .$$

Also, an initial value must be specified for \mathbf{x}_0 . Typically, the vector with zero-magnitude is used for this purpose.

BPTT begins by unfolding a recurrent neural network through time as shown in this figure. This recurrent neural network contains two feed-forward neural networks, f and g . When the network is unfolded through time, the unfolded network contains k instances of f and one instance of g . In the example shown, the network has been unfolded to a depth of $k=3$.

Training then proceeds in a manner similar to training a feed-forward neural network with backpropagation, except that each epoch must run through the observations, \mathbf{y}_t , in sequential order. Each training pattern consists of $\langle \mathbf{x}_t, \mathbf{a}_t, \mathbf{a}_{t+1}, \mathbf{a}_{t+2}, \dots, \mathbf{a}_{t+k-1}, \mathbf{y}_{t+k} \rangle$. (All of the actions for k time-steps are needed because the unfolded network contains inputs at each unfolded level.) Typically, backpropagation is applied in an online manner to update the weights as each training pattern is presented.

After each pattern is presented, and the weights have been updated, the weights in each instance of f (f_1, f_2, \dots, f_k) are averaged together so that they all have the same weights. Also, \mathbf{x}_{t+1} is calculated as $\mathbf{x}_{t+1} = f(\mathbf{x}_t, \mathbf{a}_t)$, which provides the

information necessary so that the algorithm can move on to the next time-step, $t+1$.

Pseudo-code

Pseudo-code for BPTT:

```

Back_Propagation_Through_Time(a, y) // a[t] is the input
at time t. y[t] is the output
    Unfold the network to contain k instances of f
    do until stopping criteria is met:
        x = the zero-magnitude vector; // x is the current
context
        for t from 0 to n - 1 // t is time. n is
the length of the training sequence
            Set the network inputs to x, a[t], a[t+1], ...,
a[t+k-1]
            p = forward-propagate the inputs over the whole
unfolded network
            e = y[t+k] - p; // error = target -
prediction
            Back-propagate the error, e, back across the
whole unfolded network
            Update all the weights in the network
            Average the weights in each instance of f
together, so that each f is identical
            x = f(x); // compute the
context for the next time-step

```

Advantages

BPTT tends to be significantly faster for training recurrent neural networks than general-purpose optimization techniques such as evolutionary optimization.

Disadvantages

BPTT has difficulty with local optima. With recurrent neural networks, local optima is a much more significant problem than it is with feed-forward neural

networks. The recurrent feedback in such networks tends to create chaotic responses in the error surface which cause local optima to occur frequently, and in very poor locations on the error surface.

RADIAL BASIS FUNCTION NETWORK (RBFN)

Let us look at some regression models:

1. Polynomial regression with one variable

$$y(x, w) = w_0 + w_1x + w_2x^2 + \dots = \sum w_i x^i$$
2. Simple linear regression with D variable

$$y(x, w) = w_0 + w_1x_1 + \dots + w_Dx_D = w^T X,$$
in one-dimensional case $y(x, w) = w_0 + w_1x$, which is a straight line.
3. Linear regression with Basis function $\phi_j(x)$

$$y(x, w) = w_0 + \sum_{j=1}^{M-1} w_j \phi_j(x) = w^T \phi(x)$$
there are now M parameters instead of D parameters

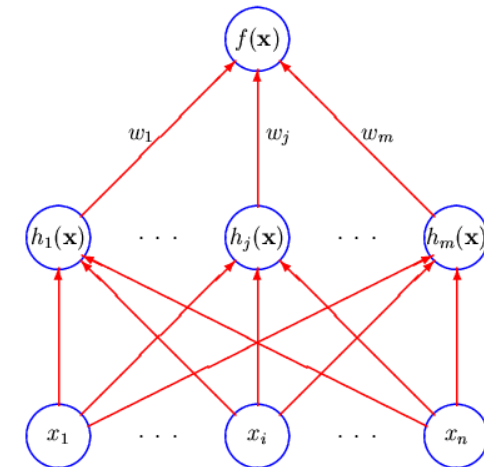
For this basis function we use *Radial Basis functions*. A radial basis function depends only on the radial distance (typically Euclidean) from the origin, i.e. $\phi(x) = \phi(\|x\|)$. If the basis function is centered at then $\phi_j(x) = h(\|x - \mu_j\|)$. We would look at radial basis functions centered at the data points $x_n, n=1, \dots, N$.

Typically $h(x)$ is a Gaussian $\phi_j(x) = \exp\left(-\frac{\|x - \mu_j\|^2}{2\sigma_j}\right)$ or a logistic function $\phi_j(x) = \frac{1}{1 + \exp\left(\frac{\|x - \mu_j\|^2}{\sigma_j}\right)}$. The RBFN is characterised

by two types of weights i.e. *hypothetical* (fixed weights of input-hidden layer) and *adjustable* weights (weights that can be adjusted, hidden-output layer weights i.e. w_i).

Algorithm for Radial Basis Function N/W

- Step 1. Initialize weights to small random values.
- Step 2. While stopping conditions fails do step-3 to step-9.
- Step 3. Activate the inputs neurons by applying a set of inputs.



Step 4. For each input do step-5 to step-8.

Step 5. Calculate RBF.

Step 6. Choose centre for each radial basis function.

Step 7. Calculate output for each hidden neuron as $h_j(x) = \exp\left(-\sum_{i=0}^n \left[\frac{\|x_i - \mu_i\|^2}{2\sigma_j}\right]\right)$

where x_i is the applied input, μ_i is the centre of Gaussian function and σ_j is the smoothing parameter or width of the Gaussian function.

Step 8. Calculate the output of the network as $y_k(x) = \sum_{j=1}^m w_{kj} h_j(x) + w_{k0}$ where w_{kj} weights of the adjustable connections, $h_j(x)$ is the response of the RB neurons, and w_{k0} is the bias for output neurons.

Step 9. Test for the stopping condition. $E = \frac{1}{2} \sum_n \sum_k y_k(x^n) - t_k^n$, where E is the error and where n is the number of input patterns, k is the sum of the values for each output node k. $y_k(x^n)$ is the achieved output for node k given input (x^n) and t_k^n is the desired output for k node given input n.

Comparison of RBFs and BP-MLP

RBF N/W	BP_MLP
Always consists of three layers, i.e. input, output and hidden layer.	Consists of one input and output layer and a number of hidden layer.
The input-hidden layer weights are hypothetical (cannot be changed).	All the connections are adjustable.
The response of the hidden layers follows the Gaussian function.	The response of the hidden layers is a linear connection of all the inputs of that neuron
Constructs local approximation to non-linear input-output mapping.	Constructs global approximation to non-linear input-output mapping.
Has non-linear activation function	It has linear activation function.

KOHONEN SELF ORGANIZING FEATURE MAP (SOM)

The key principle for map formation is that training should be taken place over an extended region of the network, which uses the concept of neighbourhood neurons. The competitive networks is similar to a single layer network, except there exist an interconnection between the output neurons because of which all the output neurons compete with each other and the winner will be selected.

There are two methods to select a winner.

1. Squared Euclidean method: Here we will find the square of the distance between input vector x_i and weight vector w_{ij} i.e. $D(j) = \sum_{i=1}^n (x_i - w_i)^2$. The winner is that neuron which is having small distance.
2. Dot Product method: Using this method will find the dot product of input vector and weight vector which is given by $P(j) = \sum_{i=1}^n x_i * w_{ij}$. That neuron is treated as winner that is having the large amount of dot products.

Algorithm for Kohonen Self Organizing Feature Map

Initially the weights and learning rates (α) are set to a small values. The inputs vector to be clustered presented to the network. Once the input vector is given and based on the initial weights all the output neurons will compete with each other and the winner will be selected. Based on the winner selection, weights are up to date for a particular unit.

Step 1. Initialize weights and learning rate.

Step 2. While stopping condition fails do step-3 to step-9.

Step 3. For each input vector do step-4 to step-6.

Step 4. For each j compute the squared Euclidean distance $D(j) = \sum_{i=1}^n (x_i - w_i)^2$ for $1 \leq i \leq n$ and $1 \leq j \leq k$.

Step 5. Find the index j for minimum D_j .

Step 6. For all units j for a specified neighbourhood of j and for all i update the weights. $w(n) = w_{old} + \alpha(x_i - w_{ij})$.

Step 7. Update the learning rule.

Step 8. Reduce the radius of topological neighbourhood.

Step 9. Test for stopping condition.

Example: $w_{ij} = \begin{bmatrix} 0.2 & 0.6 & 0.4 & 0.9 & 0.2 \\ 0.3 & 0.5 & 0.7 & 0.6 & 0.8 \end{bmatrix}$, $x_i = [0.3 \quad 0.4]$, $\alpha=0.3$.

Solution: $D(j) = \sum_{i=1}^n (x_i - w_{ij})^2$

$$D(1) = (x_1 - w_{11})^2 + (x_2 - w_{21})^2 = (0.3 - 0.2)^2 + (0.4 - 0.3)^2 = 0.01 + 0.01 = 0.02,$$

$$D(2) = (x_1 - w_{12})^2 + (x_2 - w_{22})^2 = (0.3 - 0.6)^2 + (0.4 - 0.5)^2 = 0.09 + 0.01 = 0.1,$$

$$D(3) = (x_1 - w_{13})^2 + (x_2 - w_{23})^2 = (0.3 - 0.4)^2 + (0.4 - 0.7)^2 = 0.01 + 0.09 = 0.1,$$

$$D(4) = (x_1 - w_{14})^2 + (x_2 - w_{24})^2 = (0.3 - 0.9)^2 + (0.4 - 0.6)^2 = 0.36 + 0.04 = 0.4,$$

$$D(5) = (x_1 - w_{15})^2 + (x_2 - w_{25})^2 = (0.3 - 0.2)^2 + (0.4 - 0.8)^2 = 0.01 + 0.16 = 0.17,$$

$$w(n) = w_{old} + \alpha(x_i - w_{ij})$$

$$w_{11n} = w_{11o} + \alpha(x_1 - w_{11}) = 0.2 + 0.3(0.3 - 0.2) = 0.2 + 0.03 = 0.23;$$

$$w_{21n} = w_{21o} + \alpha(x_2 - w_{21}) = 0.3 + 0.3(0.4 - 0.3) = 0.3 + 0.03 = 0.33;$$

$$w_{ij} = \begin{bmatrix} 0.23 & 0.6 & 0.4 & 0.9 & 0.2 \\ 0.33 & 0.5 & 0.7 & 0.6 & 0.8 \end{bmatrix}.$$

LEARNING VECTOR QUANTIZATION (LVQ)

The architecture of LVQ is similar to SOM architecture. In LVQ networks each output unit has a known class. Hence it uses supervised learning method. The method for initializing the reference vector.

1. Take first m training data and use them as weight vectors and remaining vectors used for training.
2. Initialize the reference vector randomly and assign initial weights and classes randomly.

Training Algorithm of LVQ

The algorithm for the LVQ network is to find the output unit that has a matched pattern with the input vector.

At the end of the process if x (input vector) and w (weight vector) belongs to the same class, then weights are moved towards the new input vector.

In this method winner neuron is identified. The winner neuron index is compared with the target and based on the comparison results weights are updated.

Step 1. Initialize the weights and learning rates.

Step 2. If stopping condition fails do step-3 to step-7.

Step 3. For each training input perform step-4 to step-5.

Step 4. Compute j using square Euclidean distance method. $D(j) = \sum_{i=1}^n (w_{ij} - x_i)^2$, find j when $D(j)$ is minimum.

Step 5. Update w_{ij} as follows: if $T=C_j$ then $w_{ij}(new) = w_{ij}(old) + \alpha(x_i - w_{ij}(old))$
else $w_{ij}(new) = w_{ij}(old) - \alpha(x_i - w_{ij}(old))$

Step 6. Reduce the learning rate α .

Step 7. Test for the stopping condition.

Example: $\begin{bmatrix} 1 & 0 & 1 & 0 \\ 0 & 0 & 1 & 1 \\ 1 & 1 & 0 & 0 \\ 1 & 0 & 0 & 1 \end{bmatrix}$ is the vector and class is given by $\begin{matrix} 1 \\ 2 \\ 1 \\ 2 \end{matrix}$ and $\alpha=0.1$ to 0.05.

Solution: Let us consider 1st two vectors as reference $w_1 = [1 \ 0 \ 1 \ 0]$ and $w_2 = [0 \ 0 \ 1 \ 1]$ and other two vectors used as training data. $\alpha=0.1$, $x_1 = [1 \ 1 \ 0 \ 0]$ and $T=1$.

$$\text{Calculate } D(j) = \sum_{i=1}^n (w_{ij} - x_i)^2$$

$$D(1) = (1-1)^2 + (0-1)^2 + (1-0)^2 + (0-0)^2 = 2.$$

$$D(2) = (0-1)^2 + (0-1)^2 + (1-0)^2 + (1-0)^2 = 4.$$

Therefore $D(1)$ is minimum i.e. $j=1$ and $C_j = 1$.

$$\text{Now } w_{ij}(new) = w_{ij}(old) - \alpha(x_i - w_{ij}(old))$$

$$w_1(new) = [1 \ 0 \ 1 \ 0] + 0.1 * ([1 \ 1 \ 0 \ 0] - [1 \ 0 \ 1 \ 0])$$

$$= [1 \ 0 \ 1 \ 0] + 0.1 * ([0 \ 1 \ -1 \ 0])$$

$$= [1 \ 0 \ 1 \ 0] + [0 \ 0.1 \ -0.1 \ 0] = [1 \ 0.1 \ 0.9 \ 0]$$

$$w_1 = [1 \ 0.1 \ 0.9 \ 0] \quad w_2 = [0 \ 0 \ 1 \ 1] \quad x_2 = [1 \ 0 \ 0 \ 1] \quad \alpha=0.1 \text{ and } T=2.$$

$$D(1) = (1 - 0)^2 + (0.1 - 0)^2 + (0.9 - 1)^2 + (0 - 1)^2 = 1.82$$

$$D(2) = (0 - 1)^2 + (0 - 0)^2 + (1 - 0)^2 + (1 - 1)^2 = 2$$

$D(1)$ is minimum and $j=1$ and $C_j = 1$ and $T \neq C_j$

$$w_1(\text{new}) = [1 \ 0.1 \ 0.9 \ 0] + 0.1 * ([1 \ 0 \ 0 \ 1] - [1 \ 0.1 \ 0.9 \ 0])$$

$$= [1 \ 0.1 \ 0.9 \ 0] + 0.1 * ([0 \ -0.9 \ -0.1 \ 1])$$

$$= [1 \ 0.1 \ 0.9 \ 0] + [0 \ -0.09 \ -0.01 \ 0.1] = [1 \ 0.11 \ 0.81 \ -0.1]$$

$$\alpha_n = 0.5 \times 0.1 = 0.05.$$

SIMULATED ANNEALING NEURAL NETWORKS

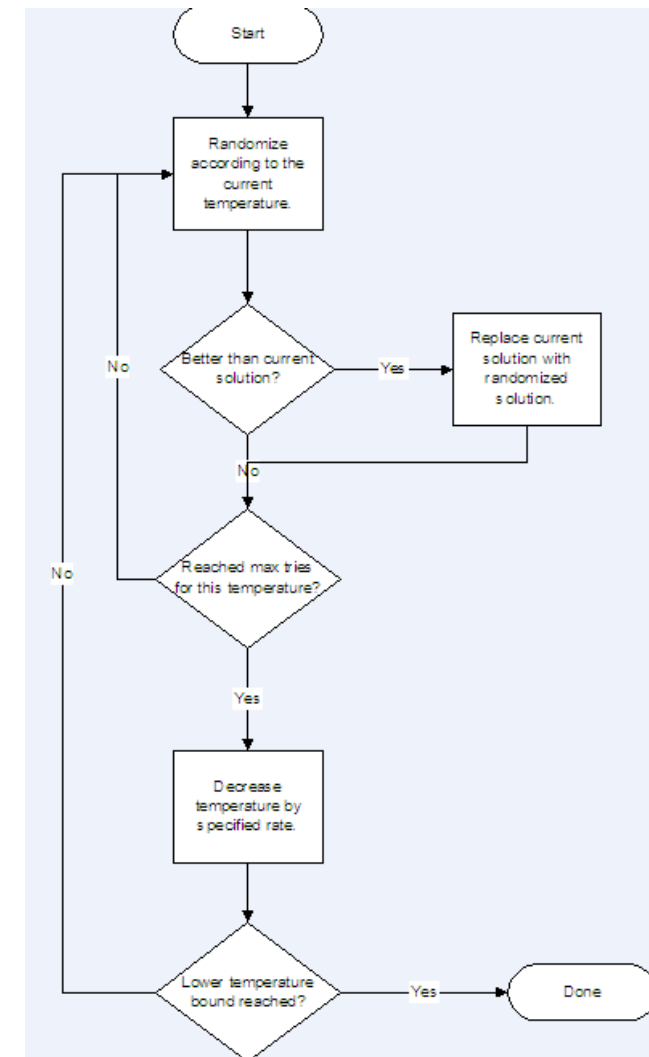
Simulated annealing was developed in the mid 1970's by Scott Kirkpatrick, along with a few other researchers. Simulated annealing was originally developed to better optimize the design of integrated circuit (IC) chips. Annealing is the metallurgical process of heating up a solid and then cooling slowly until it crystallizes. If this cooling process is carried out too quickly many irregularities and defects will be seen in the crystal structure. Ideally the temperature should be decreased at a slower rate. A slower fall to the lower energy rates will allow a more consistent crystal structure to form. This more stable crystal form will allow the metal to be much more durable.

Simulated annealing seeks to emulate this process. Simulated annealing begins at a very high temperature where the input values are allowed to assume a great range of random values. As the training progresses the temperature is allowed to fall. This restricts the degree to which the inputs are allowed to vary. This often leads the simulated annealing algorithm to a better solution, just as a metal achieves a better crystal structure through the actual annealing process.

Simulated annealing can be used to find the minimum of an arbitrary equation that has a specified number of inputs. In the case of a neural network, as we will learn in Chapter 10, this equation is the error function of the neural network. The weight matrix of a neural network makes for an excellent set of inputs for the simulated annealing algorithm to minimize for. Different sets of weights are used for the

neural network, until one is found that produces a sufficiently low return from the error function.

First, for each temperature the simulated annealing algorithm runs through a number of cycles. This number of cycles is predetermined by the programmer. As the cycle runs the inputs are randomized. Only randomizations which produce a better suited set of inputs will be kept. Once the specified number of training cycles has been completed, the temperature can be lowered. Once the temperature is lowered, it is determined if the temperature has reached the lowest allowed temperature. If the temperature is not lower than the lowest allowed temperature, then the temperature is lowered and another cycle of randomizations will take place. If the temperature is lower than the minimum temperature allowed, the simulated annealing algorithm is complete.



A neural network's weight matrix can be thought of as a linear array of floating point numbers. Each weight is independent of the others. It does not matter if two weights contain the same value. The only major constraint is that there are ranges that all weights must fall within. Because of this the process generally used to randomize the weight matrix of a neural network is relatively simple. Using the temperature, a random ratio is applied to all of the weights in the matrix. This ratio is calculated using the temperature and a random number. The higher the

temperature, the more likely the ratio will cause a larger change in the weight matrix. A lower temperature will most likely produce a smaller ratio.

HOPFIELD NETWORKS

Hopfield networks are constructed from artificial neurons. These artificial neurons have N inputs. With each input i there is a weight w_i associated. They also

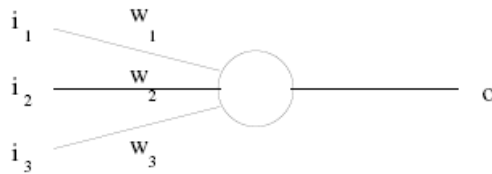


Figure 1: An artificial neuron as used in a Hopfield network.

The state of the output is maintained, until the neuron is updated. Updating the neuron entails the following operations:

- The value of each input, x_i is determined and the weighted sum of all inputs, $\sum_i w_i x_i$ is calculated.
- The output state of the neuron is set to +1 if the weighted input sum is larger or equal to 0. It is set to -1 if the weighted input sum is smaller than 0.
- A neuron retains its output state until it is updated again. Written as a formula:
$$o = \begin{cases} 1 & : \sum_i w_i x_i \geq 0 \\ -1 & : \sum_i w_i x_i < 0 \end{cases}$$

A Hopfield network is a network of N such artificial neurons, which are fully connected. The connection weight from neuron j to neuron i is given by a number w_{ij} . The collection of all such numbers is represented by the weight matrix W , whose components are w_{ij} .

Now given the weight matrix and the updating rule for neurons the dynamics of the network is defined if we tell in which order we update the neurons. There are two ways of updating them:

- Asynchronous: one picks one neuron, calculates the weighted input sum and updates immediately. This can be done in a fixed order, or neurons can be picked at random, which is called asynchronous random updating.
- Synchronous: the weighted input sums of all neurons are calculated without updating the neurons. Then all neurons are set to their new value, according to

the value of their weighted input sum. The lecture slides contain an explicit example of synchronous updating.

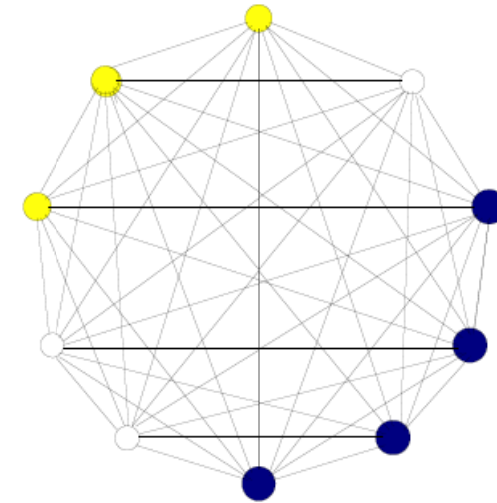


Figure 2: A Hopfield network as an autoassociator. One enters a pattern in blue nodes and let the network evolve. After a while one reads out the yellow nodes. The memory of the Hopfield network associates the yellow node pattern with the blue node pattern.

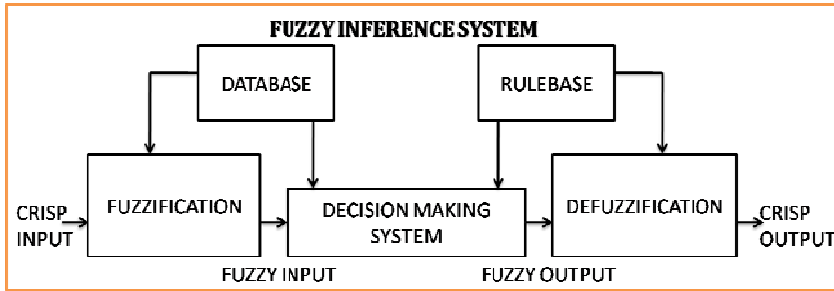
USE OF THE HOPFIELD NETWORK

The way in which the Hopfield network is used is as follows. A pattern is entered in the network by setting all nodes to a specific value, or by setting only part of the nodes. The network is then subject to a number of iterations using asynchronous or synchronous updating. This is stopped after a while. The network neurons are then read out to see which pattern is in the network.

The idea behind the Hopfield network is that patterns are stored in the weight matrix. The input must contain part of these patterns. The dynamics of the network then retrieve the patterns stored in the weight matrix. This is called Content Addressable Memory (CAM). The network can also be used for auto-association. The patterns that are stored in the network are divided in two parts: cue and association (see Fig. 2). By entering the cue into the network, the entire

pattern, which is stored in the weight matrix, is retrieved. In this way the network restores the association that belongs to a given cue.

ADAPTIVE NEURO FUZZY INFERENCE SYSTEM



The fuzzy inference system that we have considered is a model that maps

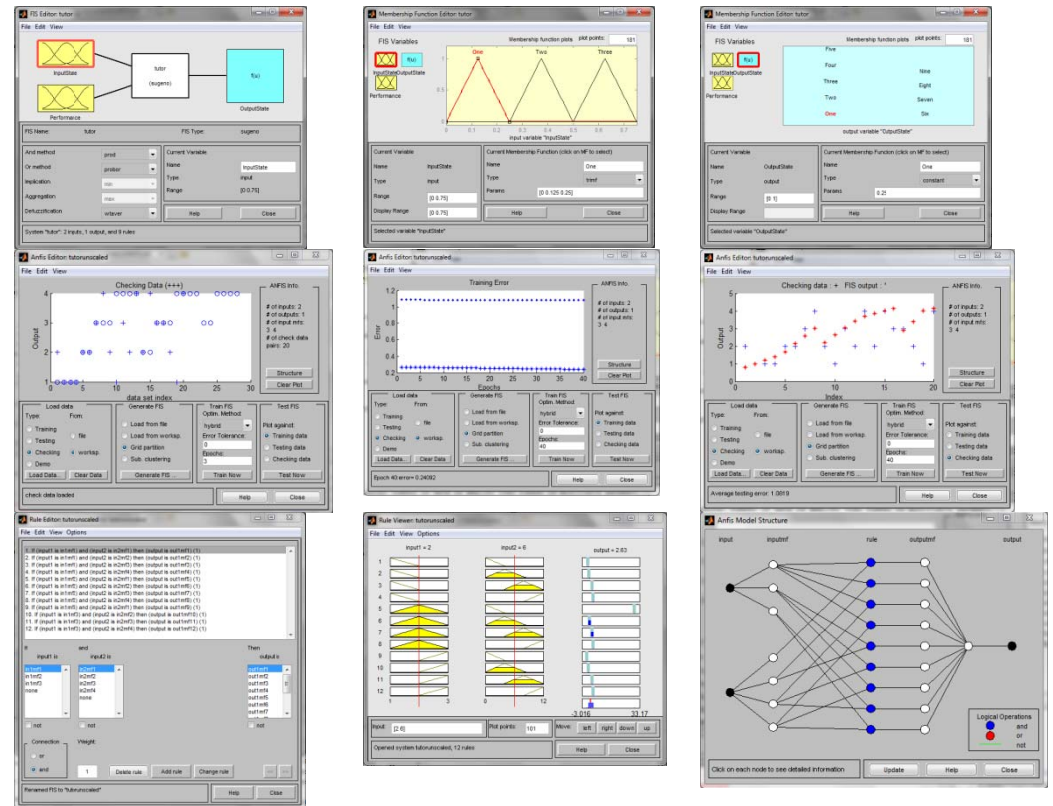
- input characteristics to input membership functions,
- input membership function to rules,
- rules to a set of output characteristics,
- output characteristics to output membership functions, and
- the output membership function to a single-valued output, or
- a decision associated with the output.

We have only considered membership functions that have been fixed, and somewhat arbitrarily chosen. Also, we have only applied fuzzy inference to modelling systems whose rule structure is essentially predetermined by the user's interpretation of the characteristics of the variables in the model. In general the shape of the membership functions depends on parameters that can be adjusted to change the shape of the membership function. The parameters can be automatically adjusted depending on the data that we try to model.

Model Learning and Inference Through ANFIS

Suppose we already have a collection of input/output data and would like to build a fuzzy inference model/system that approximate the data. Such a model would consist of a number of membership functions and rules with adjustable parameters similarly to that of neural networks. Rather than choosing the parameters associated with a given membership function arbitrarily, these parameters could be chosen so as to tailor the membership functions to the input/output data in order to account for these types of variations in the data values. The neuro-

adaptive learning techniques provide a method for the fuzzy modeling procedure to learn information about a data set, in order to compute the membership function parameters that best allow the associated fuzzy inference system to track the given input/output data. Using a given input/output data set, the toolbox function `anfis` constructs a fuzzy inference system (FIS) whose membership function parameters are tuned (adjusted) using either a backpropagation algorithm alone, or in combination with a least squares type of method. This allows your fuzzy systems to learn from the data they are modeling.



Figures: left to right and top to bottom, Sugeno FIS, Input, Output, Loading Datasets, Training, Testing, Rule base, FIS, ANFIS network.