Neural Network Applications
In
Signal Processing

Dr. Yogananda Isukapalli

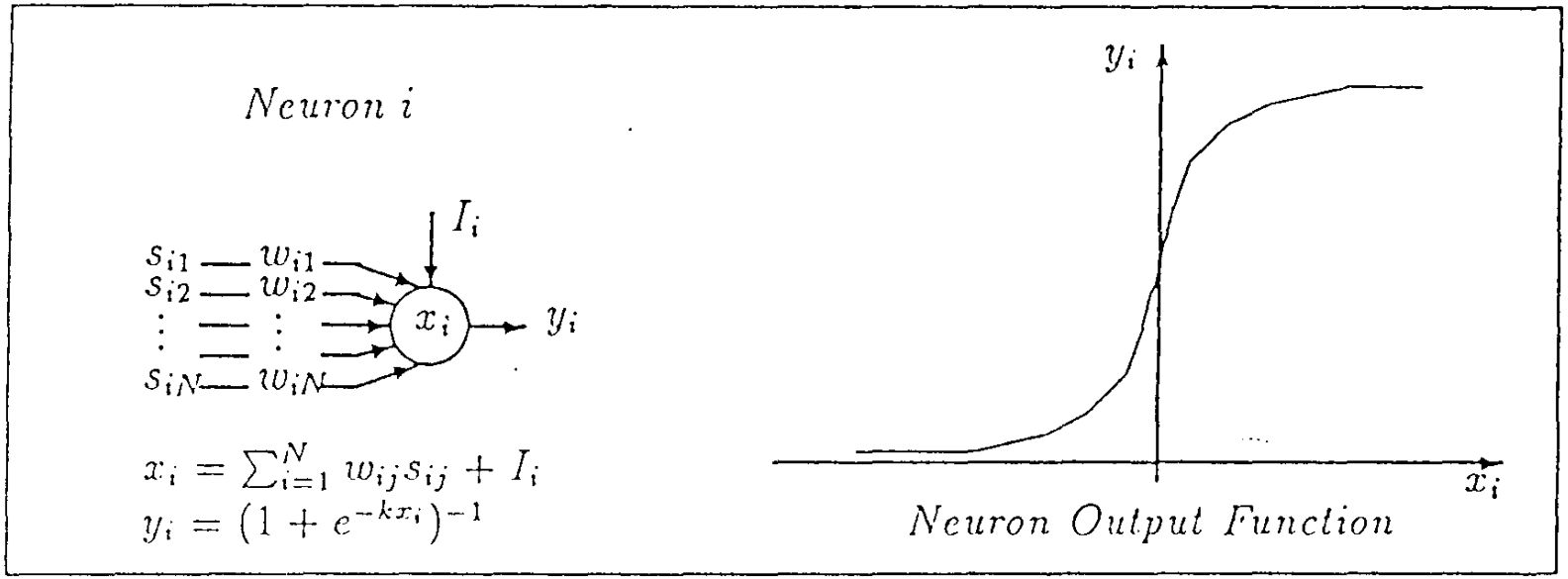
Neural Network Applications in Signal Processing

- Signal Detection by Noise Reduction
- Nonlinear Time Series Prediction
- Target Classification in RADAR, SONAR etc.
- Transient Signal Detection & Classification
- Frequency Estimation & Tracking
- Nonlinear System Modeling
- Detector for Different Signal Constellations in Digital Communication
- Speech/Speaker Recognition
- Data Compression (Speech/Image)
- Image / Object Recognition & Processing
- Principal Component Extraction & Subspace Estimation

Neural Network Architectures and Algorithms

- Multilayered Feedforward Networks
- Backpropogation Algorithms
- Radial Basis Function Networks
- Recurrent Neural Networks
- Real-time Recurrent Learning Algorithm
- Hopfield Networks
- Unsupervised Learning Algorithms

Neuron Representation and Typical Output Function



The Multilayer Perceptron

- Initialize weights and thresholds, set all weights and thresholds to small random values

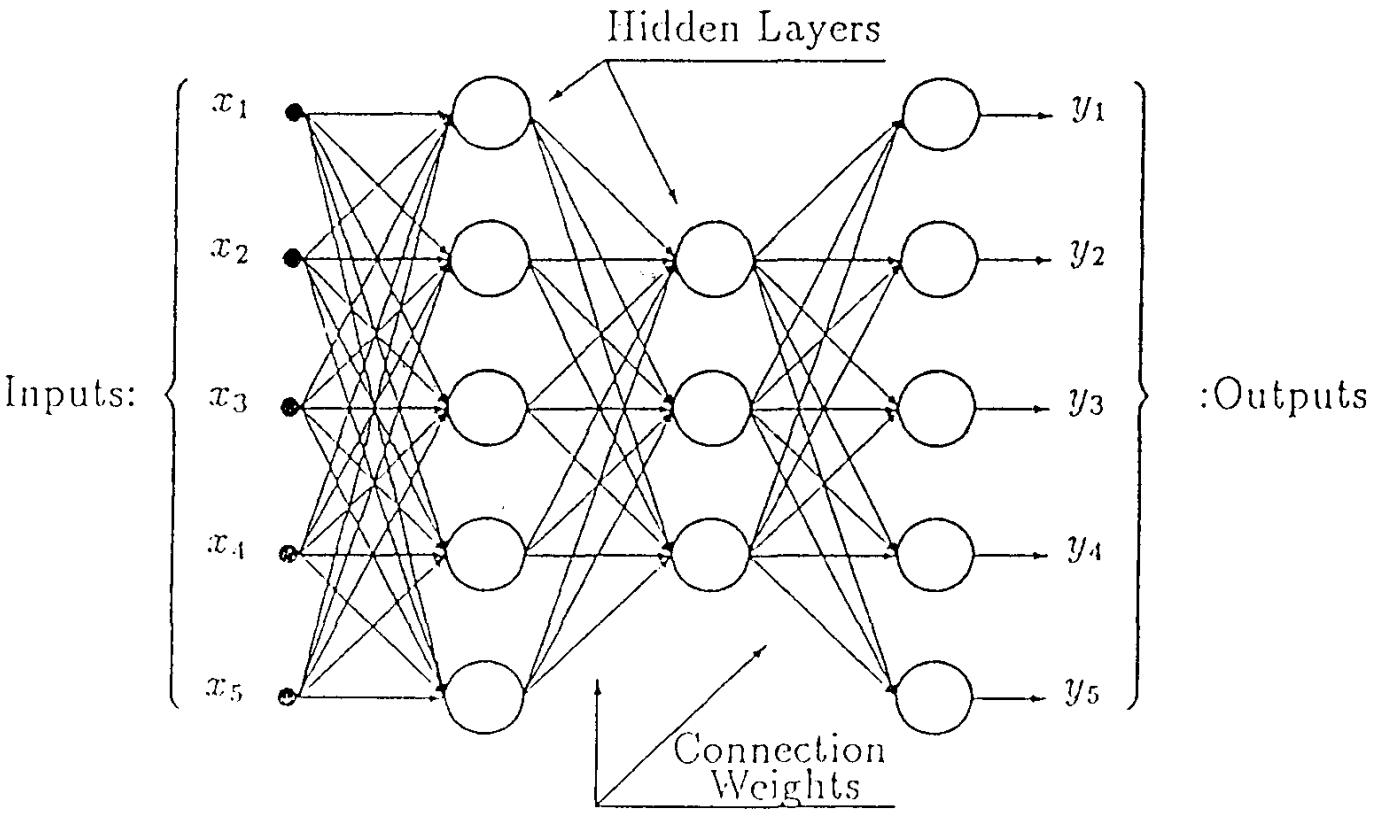
- Present input and desired output:

  Present input $X_p = x_0, x_1, x_2, \ldots\ldots x_{n-1}$

  Target output $T_p = t_0, t_1, t_2, \ldots\ldots t_{m-1}$

  where, '*n*' is the number of input nodes and '*m*' is the number of output nodes. Set w_0 to be $-\theta$, the bias, and x_0 to be always 1.

Multilayer Feed-Forward Networks



Multilayer Feed-Forward Networks

A mapping from one multi-dimensional vector space into Another multi-dimensional vector space:

Consider a pair of vectors (R^k, S^k):

$$R^k = (r_1^k, \ldots r_m^k)^T \quad : \quad \text{Input to the network}$$
$$k = 1, \ldots, p$$

$$S^k = (s_1^k, \ldots s_m^k)^T$$

For each input : Output $R^k = Y(W, R^k) = R^k = (y_1^k, \ldots y_m^k)^T$

Y(W, .) is the output of network parameterized by its connection matrix W

Multilayer Feed-Forward Networks

- If the error incurred by the network is defined as the mean square error E, then there exists a multi-layer feed-forward neural network W such that:

$$E = \min_{W} \left(\frac{1}{2} \sum_{k=1}^{p} \left[Y(W, R^k) - S^k \right]^T \left[Y(W, R^k) - S^k \right] \right)$$

Neural Networks: Learning Algorithms

- Hebbian Learning
- Widrow-Hoff Learning
- Perceptron Learning
- Backpropagation Learning
- Instar and Outstar (Grossberg) Learning
- Kohonen Learning
- Kosko / Klopf Learning
- Other Learning algorithms

Back-propagation Learning Algorithm

Assume: M Layer

N_1 neuron / layer1 = 1, 2, …., M

$N_0 = m$ inputs in the first hidden layer

$N_m = n$ inputs in the output layer

The total activation of the *i*th neuron in layer1, denoted by x_1, is computed as follows:

$$x_{l,i} = \sum_{j=1}^{N_l - 1} W_{l,i,j} y_{l-1,j} + W_{l,i,0}$$

Where

$W_{1,i,j}$: Connection weight from neuron *j* in layer *l*-1 to neuron in layer1

$W_{1,i,j}$: the threshold of the neuron

Back-propagation Learning Algorithm

$$y_{l,i} = f(x_l, i) = \frac{1}{1 + \exp^{-kx_{l,i}}} \qquad k > 0$$

= Continuous, differentiable, monotonically (sigmoid) function of its total activation.

- When pattern K is presented to the network, $y^k{}_{0,i} = r_i^k$ for i = 1, … N_0
- The Neurons in the first hidden layer will compute their output and propagate their outputs to the layer above
- The output of the network will be given by the neurons at the very last layer, i.e. $y_{M,i}$, i = 1, 2, …. N_M

Back-propagation Learning Algorithm

- The goal is to find the best set of weights W such that the following error measure is minimized:

$$E = \min_{W} \sum_{k=1}^{p} E^{k} = \min_{W} \sum_{k=1}^{p} \sum_{i=1}^{n} (y^{k}{}_{M,i} - S_{i}^{K})^{2}$$

The partial derivative of the error with respect to the weights of the neurons in the output layer is:

$$\frac{\partial E}{\partial W_{M,i,j}} = \sum_{k=1}^{p} \frac{\partial E^{K}}{\partial y_{M,i}} \cdot \frac{\partial y^{k}{}_{M,i}}{\partial x_{M,i}} \cdot \frac{\partial x^{k}{}_{M,i}}{\partial W_{M,i,j}}$$

$$\frac{\partial E}{\partial W_{M,i,j}} = \sum_{k=1}^{p} \left(y^{k}{}_{M,i} - S_{i}^{k} \right) \frac{dy^{k}{}_{M,i}}{dx_{M,i}} . y^{k}{}_{M-1,j}$$

Back-propagation Learning Algorithm

- Note that

$$\frac{dy^{k}{}_{l,i}}{dx^{k}{}_{l,i}} = f'(x^{k}{}_{l,i})$$

- define $\delta^{k}{}_{m,i}$ as

$$\delta^{k}{}_{M,i} = (y^{k}{}_{M,i} - S_{i}^{k}).f'(x^{k}{}_{M,i})$$

Back-propagation Learning Algorithm

The partial derivative of the error with respect to the weights Of the neurons in the output layer is:

$$\frac{\partial E}{\partial W_{M,i,j}} = \sum_{k=1}^{p} \frac{\partial E^{K}}{\partial y_{M,i}} \cdot \frac{\partial y^{k}{}_{M,i}}{\partial x_{M,i}} \cdot \frac{\partial x^{k}{}_{M,i}}{\partial W_{M,i,j}}$$

$$\frac{\partial E}{\partial W_{M,i,j}} = \sum_{k=1}^{p} \left(y^{k}{}_{M,i} - S_{i}^{\,k} \right) \frac{dy^{k}{}_{M,i}}{dx_{M,i}} . y^{k}{}_{M-1,j}$$

Back-propagation Learning Algorithm

- $\delta^k{}_{m,i}$ is generalized for the other layers by defining

$$\delta^k{}_{M,i} = f'(x^k{}_{M,i}) \sum_{j=1}^{N_{l+1}} \delta^k{}_{l+1} . W_{l+1,j,i}$$

* * l = M-1, ...,1; i = 1, …. N_1

This makes it clear how the errors at one layer are propagated backwards into the previous layer

- By combing * and * *, the partial derivative of the error with respect to any of the weights in the hidden layer can be written as:

$$\frac{\partial E}{\partial W_{l,i,j}} = \sum_{k=1}^{p} \delta^k{}_{l,i} y^k{}_{l-1,j}$$

Back-propagation Learning Algorithm

- To "change" the weights by a small amount in the direction which causes the error to be reduced, the following update procedure is introduced:

$$W_{l,i,j}^{(t+1)} = W_{l,i,j}^{(t)} - \alpha_t \frac{\partial E^{(t)}}{\partial W_{l,i,j}}$$

The procedure is shown to converge provided that α_t, the learning rates were chosen such that

$$\sum_{t=0}^{\infty} \alpha_t^{\,2} < +\infty \text{ and } \sum_{t=1}^{\infty} \alpha_t^{\,2} < +\infty$$

These conditions are satisfied by the particular choice of

$$\alpha_t = \alpha_0 / t \text{ for some } \alpha_0 > 0$$

but the convergence may become very slow

Back-propagation Learning Algorithm

- Increasing the rates of convergence

A. If we use a constant learning rate at such that $0.1<\alpha<0.9$, the algorithm still approximates the steepest descent path, and should produce a reasonably good solution

B. To reduce the effects of a very shallow error surface, a momentum term $\beta\Delta w$ is added as follows:

$$\Delta W_{l,i,j}^{(t+1)} = -\alpha(1-\beta)\frac{\partial E^{(t)}}{\partial W_{l,i,j}} + \beta\Delta W_{l,i,j}^{(t)}$$

where β is the amount of momentum that should be added.

Back-propagation Learning Algorithm

In practice β should range from 0 to 0.4. Rewriting the above equation:

$$\Delta W_{l,i,j}^{(t+1)} = -\alpha(1-\beta)\sum_{k=0}^{t}\beta^{k}\frac{\partial E^{(t-k)}}{\partial W_{l,i,j}}$$

The change to the weights becomes the exponentially weighted Sum of the current and the past error's partial derivative with Respect to the weight being changed.

Kolmogorov's Mapping Neural Network

Existence Theorem: (How many hidden layers are needed?)

Given any continuous function $\Phi: I^d \rightarrow R^c$, $\Phi(x) = y$, where I is the closed interval [0,1] (and therefore I^d is the d-dimensional unit cube), Φ can be implemented exactly by a three-layer neural network having d processing elements in the input layer, (2d+1) processing elements in the single (hidden) layer, and c processing elements in the output layer. The input layer serves merely to 'hold' or freeze the input and distribute each input to the hidden layer.

RADIAL BASIS FUNCTION NETWORKS

- Examples of Radial Basis Functions

$$\phi(X) = \exp\left(\frac{X^2}{\sigma^2}\right) \qquad \text{Gaussian}$$

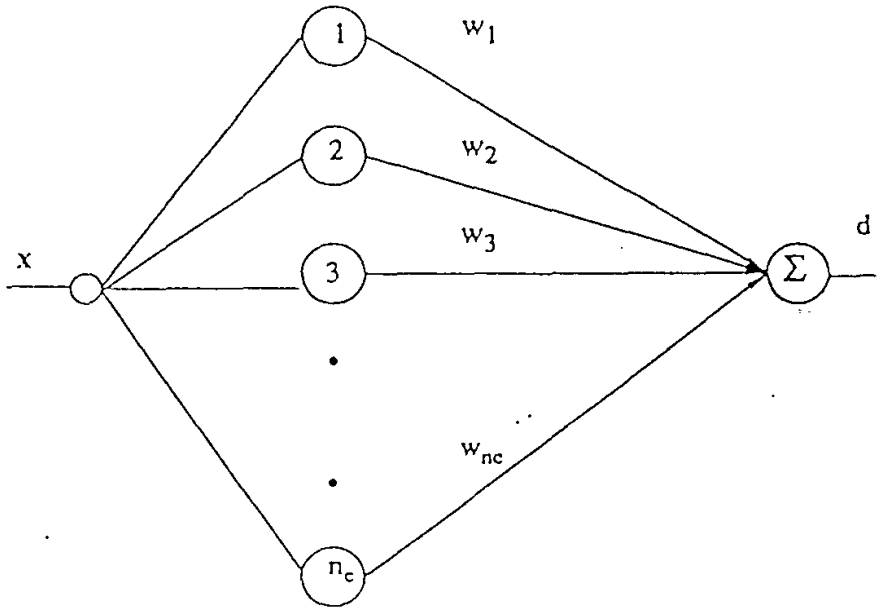
$$\phi(X) = X^2 \log(X) \qquad \text{Thin plate Spline}$$

$$\phi(X) = (X^2 + \sigma^2)^{\frac{1}{2}} \qquad \text{Multi-quadratic function}$$

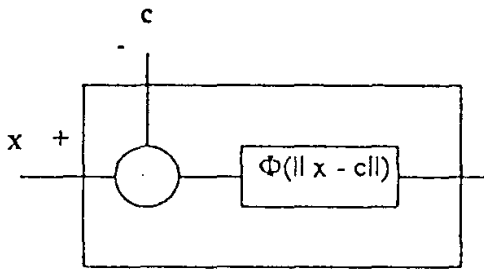
$$\phi(X) = (X^2 + \sigma^2)^{-\frac{1}{2}} \qquad \text{Inverse multi-quadratic function}$$

X is the euclidian norm of the difference between the input vector and the rbf center for that node

RADIAL BASIS FUNCTION NETWORKS



Radial Basis Function Network



A Radial Basis Neuron

Approximation by Radial Basis Functions

Given N different input vectors, X_i, $X_i \varepsilon R^n$, i = 1, 2, … N, and the corresponding N real outputs, y_i, $y_i \varepsilon R$, i = 1, 2, … N, find a function F, $F:X_i, R^n \rightarrow R$, satisfying the interpolation conditions

$$F(X_i) = y_i \qquad i = 1, 2, \ldots, N$$

The solution to the above problem is a function which is linear combination of N radial basis functions, i.e.,

$$F(x) = \sum_{i=1}^{N} C_i \Phi(\| X - X_i \|)$$

Neural Network Architecture Using Radial Basis Functions

Parallel network like structure

Points in the data space are chosen as basis functions for interpolation

The outputs are the weighted and summed quantities of the outputs of the RBF centers, i.e.,

$$O_j = \sum_{i=1}^{N} W_{ij}\phi_i$$

n_c: the number of RBF centers
O_j: the output of the jth output node

- Implementing RBF Networks

1. Gaussian Kernel is typically chosen
2. the spread σ (Gaussian Kernel) is chosen by K nearest neighbor rule
3. Fix the RBF centers by choosing the subset of the data vectors that represent the entire data set, The vectors are generally obtained by a K-means clustering algorithm
4. Compare the weights to the output layer by

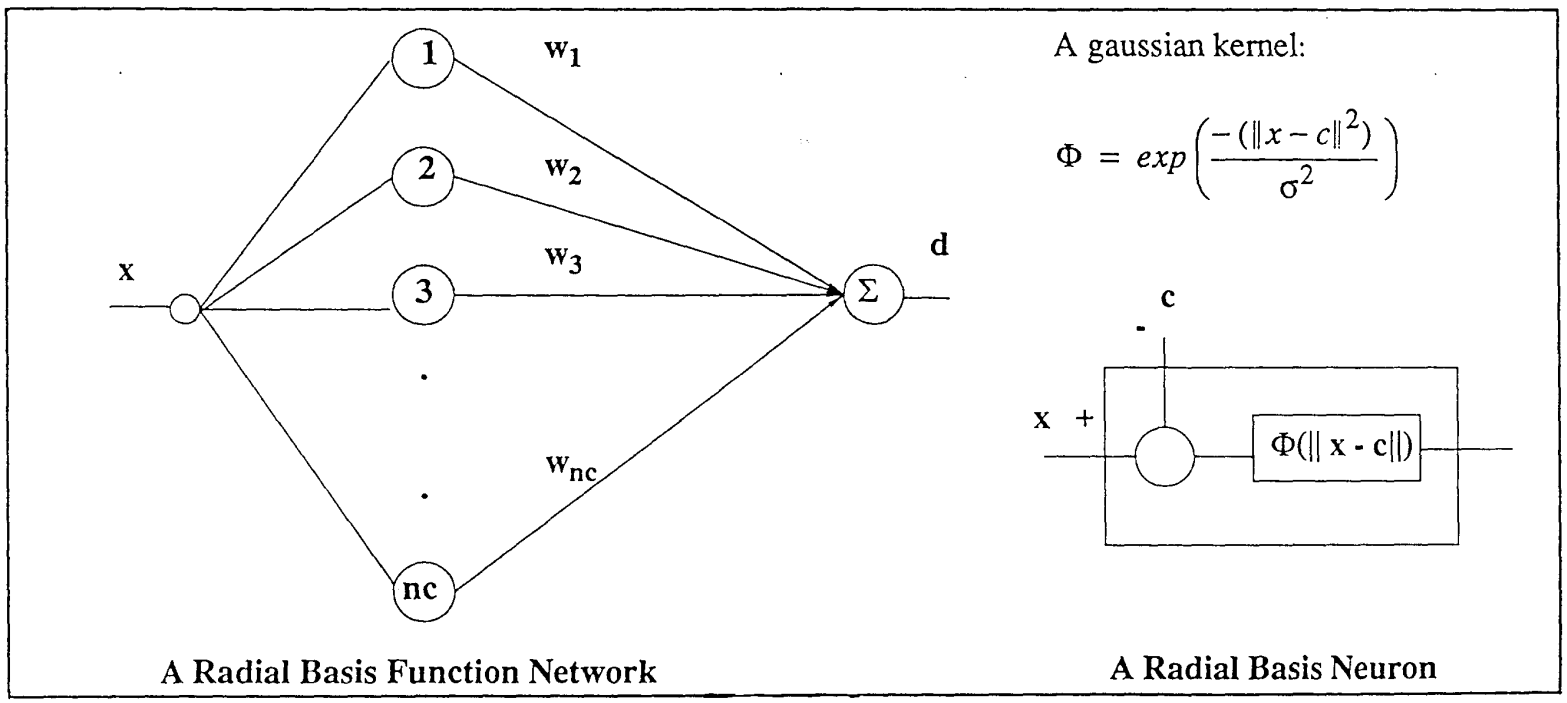
$$W = A^{j}D$$

where A represents the output of the first layer units for all the available training patters; D represents the desired output patterns and A^{j} is the Moore-penrose pseudo-inverse of A

5. Generalize the strict interpolation method to a successive approximation radial basis network function network (Xiangdong He and Lapedes)

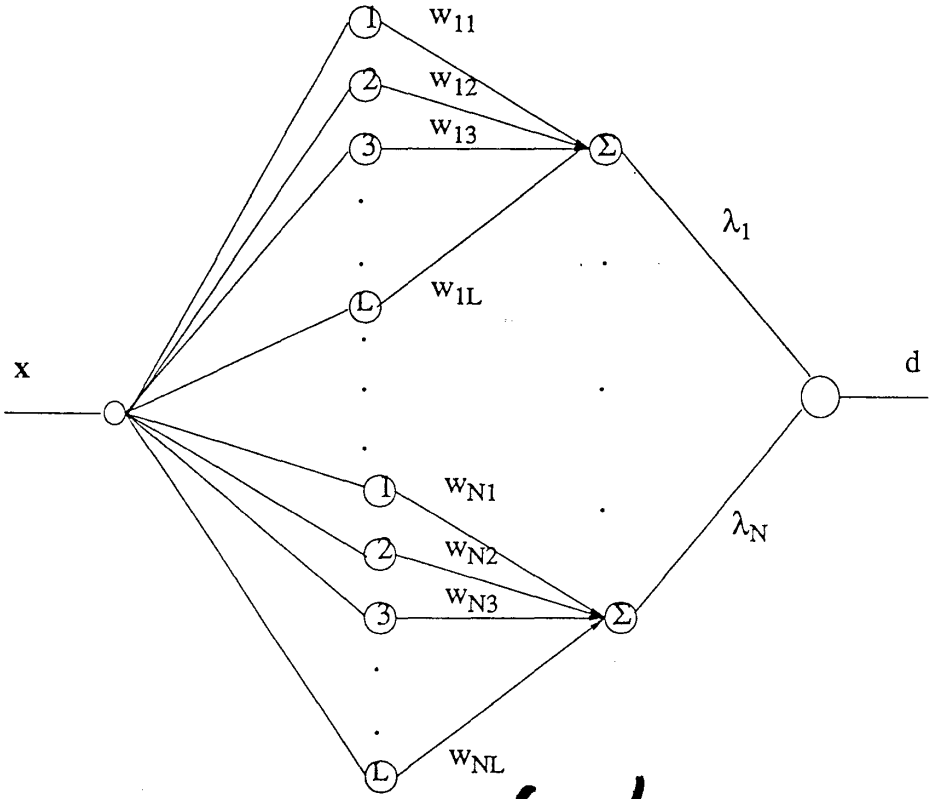
Radial Basis Function Networks

Basis functions are the centers of the RBF neurons. All the available training patterns are prospective candidates. The centers can be fixed by k-means clustering or by the orthogonal least squares method. The function Pi is a radial function, i.e., it is a function of the distance of a particular vector x from the RBF center c. The weights are obtained by a least squares fit over all the training patterns.



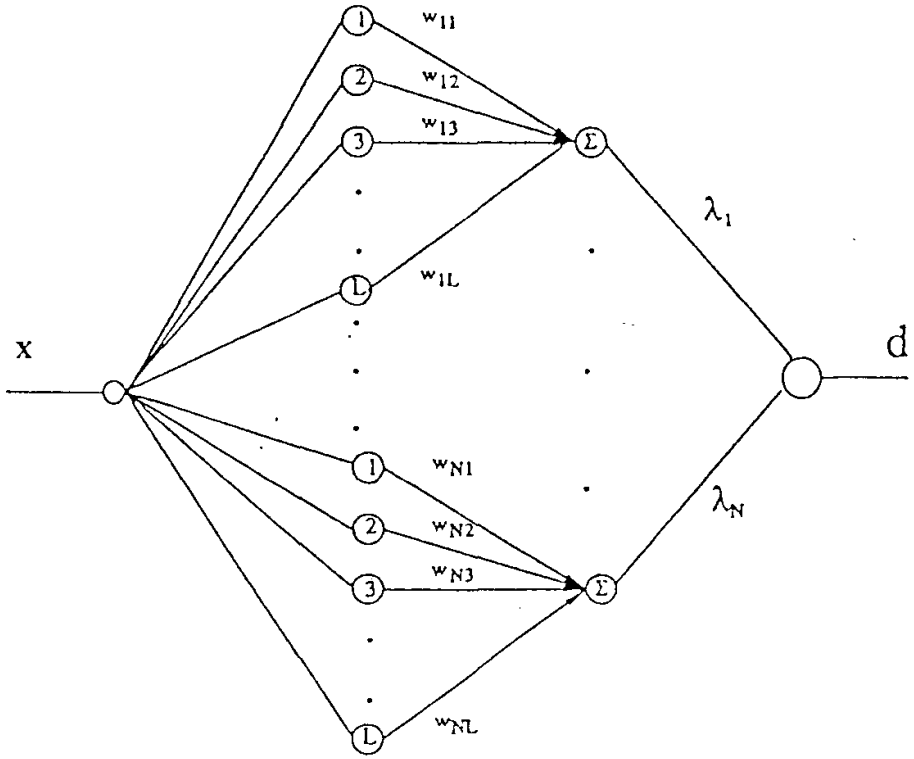
Two Layered RBF Networks

All training patterns are used as centers. The given patterns are divided into N groups with L centers each. The w's and λ's are obtained by least squares fit over all the patterns.



A Two layered Radial Basis Function Network

Radial Basis Function Networks



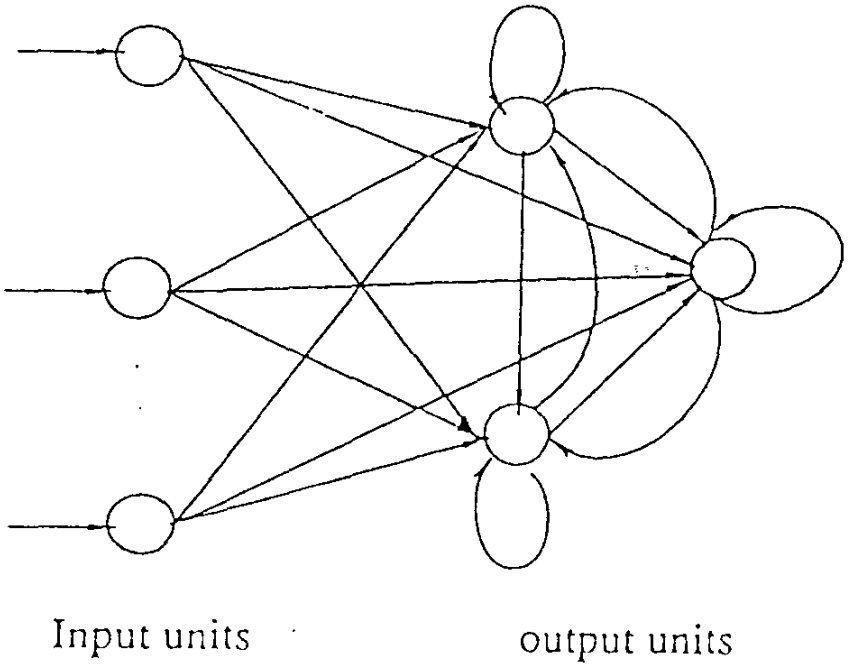
A Two layered Radial Basis Function Network

Recurrent Neural Networks

- A fully connected recurrent is one which all the outputs of each unit is connected to the input of all the units, including itself and excluding the external input nodes

- The recurrent network architecture allows the network to retain the information from the infinite past.

Recurrent Neural Networks



Recurrent Neural Networks

- Real-Time Recurrent Learning (RTRL) Algorithm (Williams and Zipser)

Assume:

- m: Input Nodes
- n: Output Nodes
- q: designated target units for which desired outputs are known
- n-q: The remaining nodes for which the desired outputs are set to zero
- W_{ij}: The weight connecting the i^{th} and j^{th} units

Let:

- Z = [x,y]
- X: the input vector of size 1 x m
- Y: the output unit of size 1 x n
- Z: the augmented vector of size 1 x (m+n)

Recurrent Neural Networks

then

$$S_k(t) = \sum_{l=1}^{m+n} W_{kl} Z_l$$

$$Y_k(t+1) = \frac{1}{1+(\exp)^{-S_k(t)}}$$

$$e_k(t+1) = y_k(t+1) - d_k(t+1)$$

d_k: the desired output for the k^{th} output unit

Recurrent Neural Networks

$$f'_k(S_k) = y_k(t+1)(1 - y_k(t+1))$$

The weights are updated by gradient descent method

Now: The minimum mean squared error is the optimization criterion as follows:

$$E = \frac{1}{2}\sum_j (d_j - y_j)^2$$

$$\frac{\partial E}{\partial W_{ij}} = \sum_{k=1}^{q} e_k(t+1).\frac{\partial}{\partial W_{ij}} y_k(t) = \sum_{k=1}^{q} e_k(t+1)\, p_{ij}^k(t)$$

Recurrent Neural Networks

$$p_{ij}^{k}(t+1) = \frac{\partial E}{\partial S_k} \cdot \frac{\partial S_k}{\partial W_{ij}}$$

$$p_{ij}^{k}(t+1) = f'(S_k)\left[\sum_{l=1}^{n} W_{kl}\, p_{ij}^{l}(t) + \delta_{ik} Z_j(t)\right]$$

$$\Delta W_{ij}(t) = \alpha \sum_{k=1}^{n} e_k(t+1).p_{ij}^{k}(t+1)$$

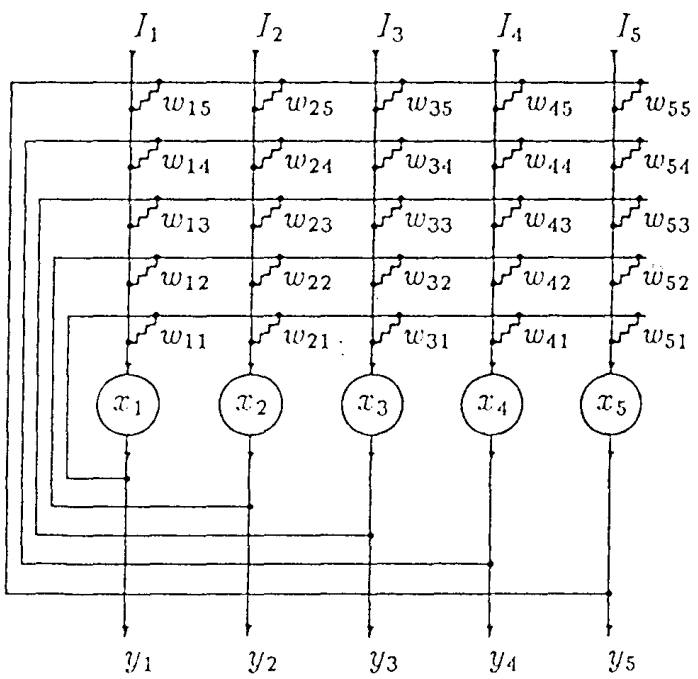
$$W_{ij}(t+1) = w_{ij}(t) + \Delta W_{ij}(t)$$

Recurrent Neural Networks

Comments on RTRL:

- Computationally intensive and higher storage requirements
- Computational complexity is $O(n^4)$
- Storage requirements $O(n^3)$
- Parallel implementation will reduce the computational time (Williams and Zipser, 1989)

The Little/Hopfield Neural Network Model



I_i: Neuron Threshold

Connection Matrix
W

$$x_i = \sum_{j=1}^{5} w_{ij} y_j + I_i$$

$$y_i = sign(x_i)$$

Hopfield Network

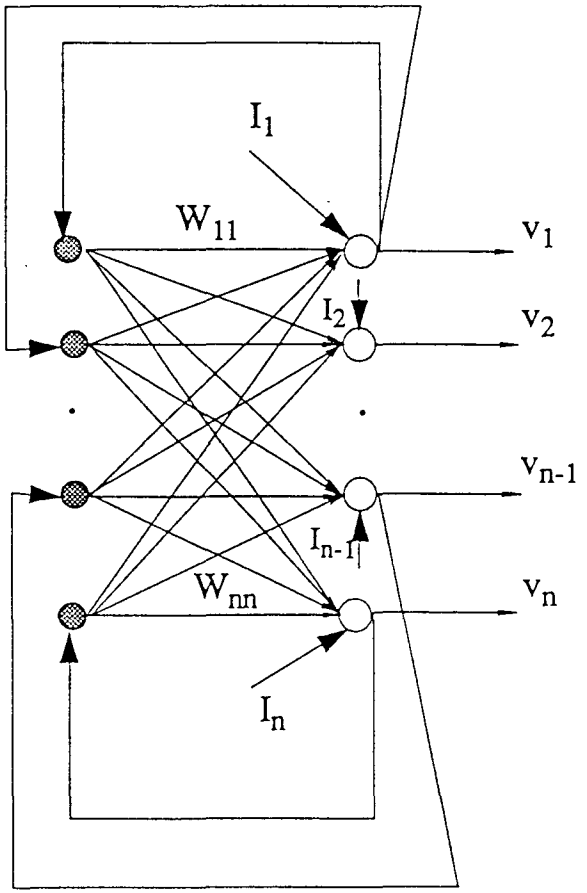
The outputs of the network are modified in such a way that the Lyapunov function E is minimized.

$$E = \frac{1}{2}\sum_{i=1}^{n}\sum_{j=1}^{n} W_{ij} v_i v_j - \sum_{i=1}^{n} I_i v_i$$

W_{ij} are the weights, I_i's are the inputs

$$v_i(t+1) = f\left(\sum W_{ki} v_k(t) + I_i\right)$$

Thus, if we can identify the above structure of E in an optimization problem, then the weights and inputs can be computed. The network is trained till the output v_i's converge.



A typical Hopfield Neural Network