Radial-Basis Networks

Radial basis function (RBF) neural network is based on supervised learning. RBF networks were independently proposed by many researchers and are a popular alternative to the MLP. RBF networks are also good at modeling nonlinear data and can be trained in one stage rather than using an iterative process as in MLP and also learn the given application quickly. They are useful in solving problems where the input data are corrupted with additive noise. The transformation functions used are based on a Gaussian distribution. If the error of the network is minimized appropriately, it will produce outputs that sum to unity, which will represent a probability for the outputs.

Consider a noisy data pattern.

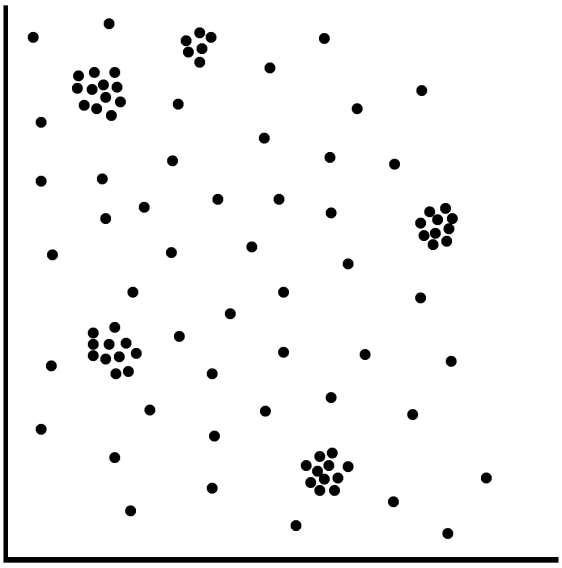


Fig. 6-0: Noisy data pattern

In such data pattern, MLPs find difficulty classifying the input set. For this data pattern, the clustered can be separated from the sparse data if we create a new dimension that represents average distance of the data points from other data points. Once the average distance is obtained, a single plane can classify the data. This technique is known as data-clustering techinique.

In contrast to sigmoidal functions, radial basis functions have **radial symmetry** about a center in n-space (n = # of inputs). The **farther** from the center the input is, the **less** the activation. This models the "**on-center off-surround**" phenomenon found in certain **real neurons** in the visual system, for example.

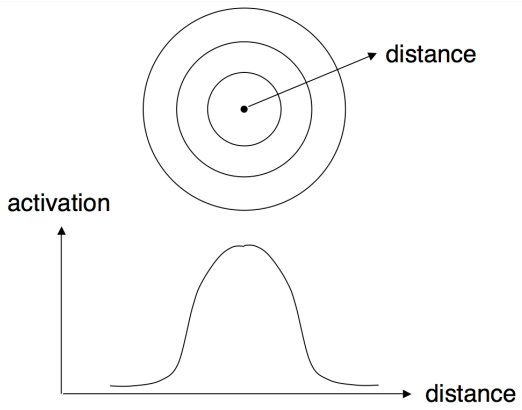


Fig. 6: On-center, off-surround

The network architecture

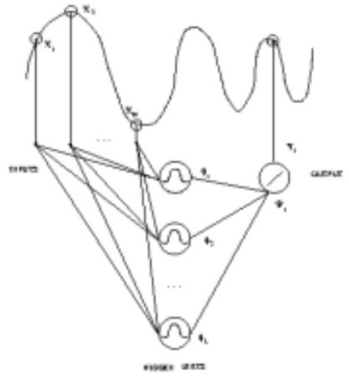


Figure 1. RBF network in time series modeling.

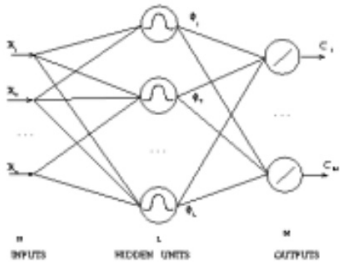


Figure 2. RBF network in pattern classification.

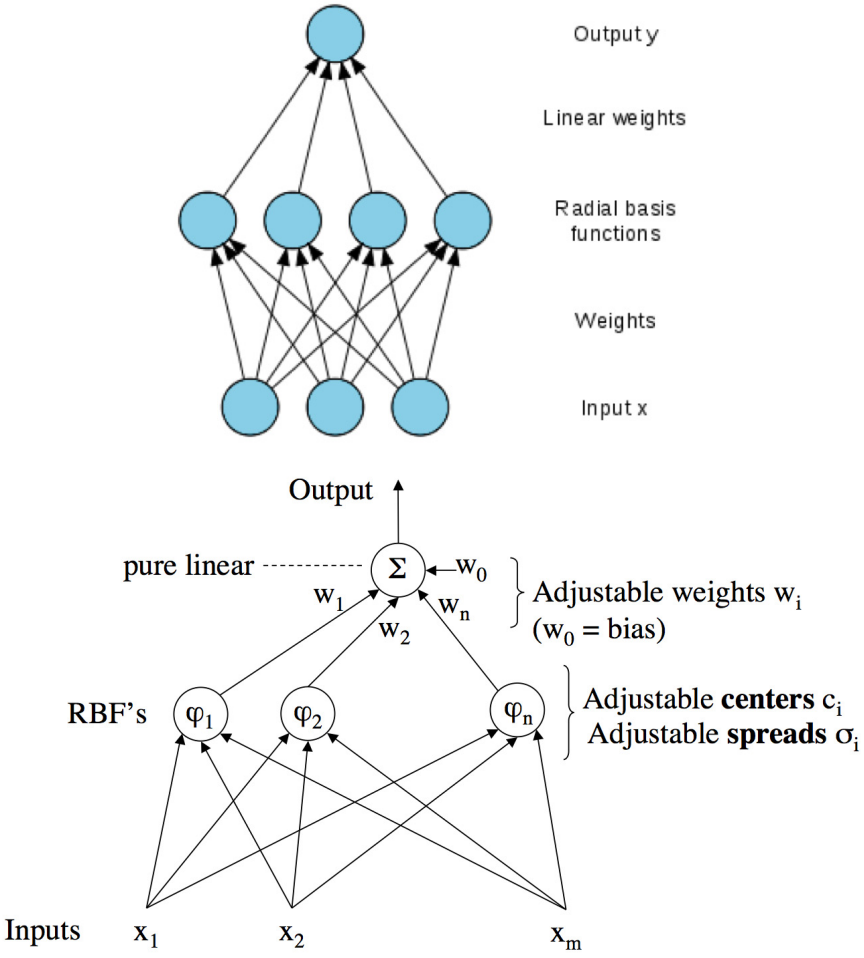


Fig. 7: RBF Topologies – Two layers only

The model

The activation of one neuron is;

$$\varphi_i(x) = G(\|x - c_i\|)$$

where, G is a decreasing function and Ci is the center.

Example: Gaussian:

$$G(y) = \exp\left(-\frac{y^2}{\sigma^2}\right)$$

where σ is a parameter called the **spread**, which indicates the **selectivity** of the neuron.

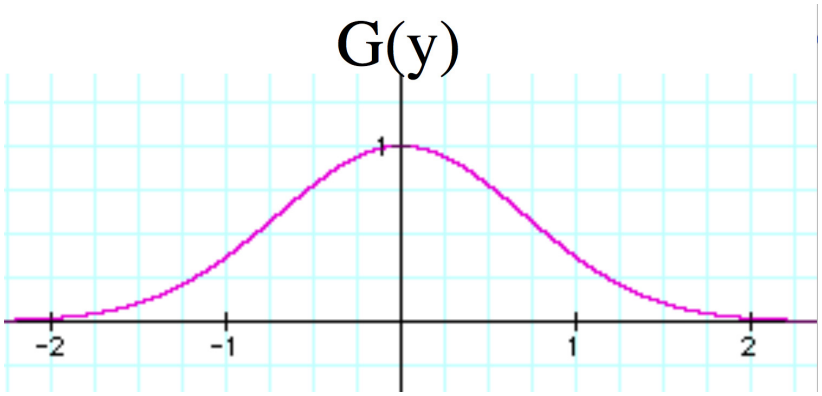


Fig. 8: Gaussian Distribution

Other RBF examples;

$$G(y) = \frac{1}{sqrt(y^2 + \sigma^2)}$$

$$G(y) = \frac{1}{1 + \exp(ay^2)}$$ (reflective sigmoid)

Spread = 1 / selectivity. Small spread will result into very selective network and large spread not very selective and hence not much accurate.

Output of the network = $\sum w_i \varphi_i(x)$ *where x is the input vector*

The data points are encapsulated as;

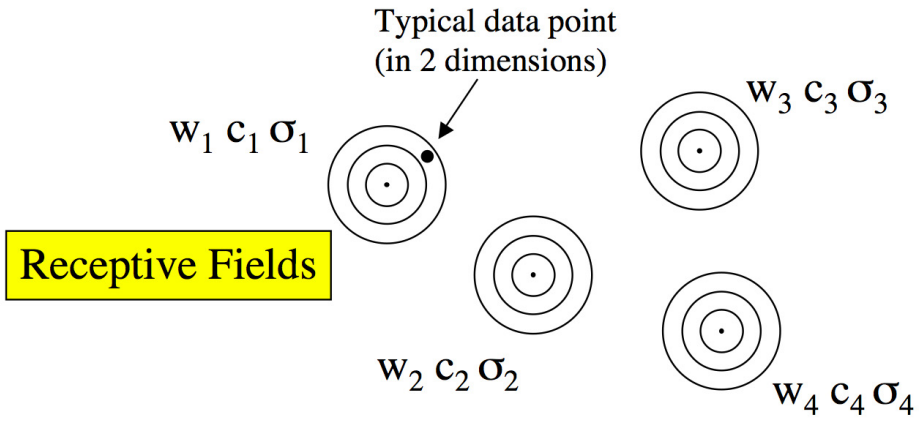


Fig. 9: Receptive fields

also known as **receptive fields**.

Determination of the parameters

Given a set of data, the weights, centers, and spreads need to be determined for the best fit.

Various approaches exist for the determination;

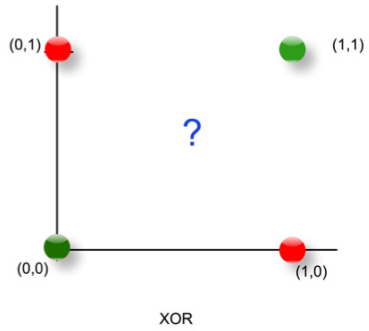
- Solving for all parameters
- Determining centers and spreads by clustering, then training weights
- Training for centers, spreads, and weights

These approaches assume a **specified number** of hidden-layer nodes.

Another approach is to add nodes successively until the approximation is good. This method is also known as **constructive neural network learning** approach.

Let's take an example for XOR classification.

- How to choose parameters to realize XOR with 2-unit RBF?
- Since output is linear, would need to add a **limiter** to general RBF.



Choose centers at (1,0), and (0, 1). Choose spreads as, say 0.1, find weights.

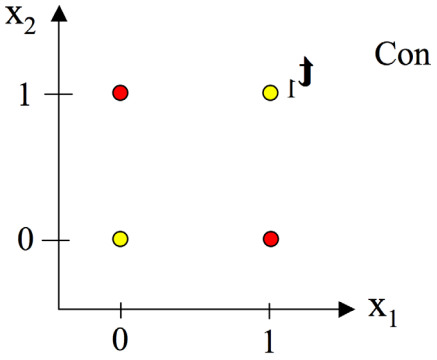


Fig. 10: XOR

Consider the non-linear functions to map the input vector X to the $\varphi_1 - \varphi_2$ space.

Then,

$$X = [x_1 \quad x_2]$$

$$\varphi_1(x) = e^{-\|x - t_1\|^2} \qquad \varphi_2(x) = e^{-\|x - t_2\|^2}$$

$$t_1 = [1 \quad 1]^T \qquad t_2 = [0 \quad 0]^T$$

$$\|x - t_1\|^2 = (x_0 - t_{10})^2 + (x_1 - t_{11})^2$$

which will result into;

0 1

Input x	$\varphi_1(x)$	$\varphi_2(x)$
(1,1)	1	0.1353
(0,1)	0.3678	0.3678
(1,0)	0.3678	0.3678
(0,0)	0.1353	1

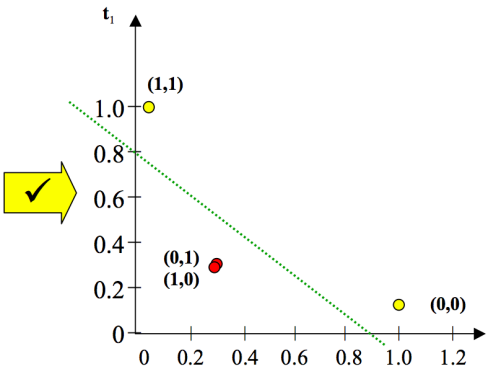


Fig. 11: XOR Linearly Separable

RBF Properties

- RBF networks tend to have good **interpolation** properties,
- but not as good **extrapolation** properties as MLP's.
- For extrapolation, using a given number of neurons, an MLP could be a much better fit.

Training-Performance and Universality

- With proper setup, RBFNs can train in time **orders of magnitude faster** than backpropagation.
- RBFNs enjoy the same **universal approximation** properties as MLPs: given sufficient neurons, any reasonable function can be approximated (with just 2 layers).

Possible Applications

- Face recognition
- Odor sensing
- Color image classification
- Time series applications, forecasting

Matlab

```
%   NEWRB(PR,T,GOAL,SPREAD,MN,DF) takes these arguments,
%     P      - RxQ matrix of Q input vectors.
%     T      - SxQ matrix of Q target class vectors.
%     GOAL   - Mean squared error goal, default = 0.0.
%     SPREAD - Spread of radial basis functions, default = 1.0.
%     MN     - Maximum number of neurons, default is Q.
%   and returns a new radial basis network.
%   The larger that SPREAD is the smoother the function approximation
%   will be.  Too large a spread means a lot of neurons will be
%   required to fit a fast changing function.  Too small a spread
%   means many neurons will be required to fit a smooth function,
%   and the network may not generalize well.  Call NEWRB with
%   different spreads to find the best value for a given problem.
```

The training approach for RBFNs

The weights, centers, and spreads are iteratively trained using **gradient descent**.

If, 'j' is the sample index, and 'i' is the weight index;

$$Error = \varepsilon = \frac{1}{2}\sum_{j=1}^{N} e_j^2$$

$$e_j = d_j - \sum_{i=1}^{M} w_k . \varphi\left(\left\| x_j - t_i \right\|\right)$$

$$G\left(\left\| x_j - t_i \right\|\right)_c = \varphi\left(\left\| x_j - t_i \right\|\right)$$

TABLE 5.4 Adaptation Formulas for the Linear Weights and the Positions and Spreads of Centers for RBF Network[a]

1. *Linear weights* (output layer)

$$\frac{\partial \mathscr{E}(n)}{\partial w_i(n)} = \sum_{j=1}^{N} e_j(n) G(\|\mathbf{x}_j - \mathbf{t}_i(n)\|_{C_i})$$

$$w_i(n+1) = w_i(n) - \eta_1 \frac{\partial \mathscr{E}(n)}{\partial w_i(n)}, \qquad i = 1, 2, \ldots, \mathbf{m}_1$$

2. *Positions of centers* (hidden layer)

$$\frac{\partial \mathscr{E}(n)}{\partial \mathbf{t}_i(n)} = 2w_i(n) \sum_{j=1}^{N} e_j(n) G'(\|\mathbf{x}_j - \mathbf{t}_i(n)\|_{C_i}) \boldsymbol{\Sigma}_i^{-1} [\mathbf{x}_j - \mathbf{t}_i(n)]$$

$$\mathbf{t}_i(n+1) = \mathbf{t}_i(n) - \eta_2 \frac{\partial \mathscr{E}(n)}{\partial \mathbf{t}_i(n)}, \qquad i = 1, 2, \ldots, \mathbf{m}_1$$

3. *Spreads of centers* (hidden layer)

$$\frac{\partial \mathscr{E}(n)}{\partial \boldsymbol{\Sigma}_i^{-1}(n)} = - \; w_i(n) \sum_{j=1}^{N} e_j(n) G'(\|\mathbf{x}_j - \mathbf{t}_i(n)\|_{C_i}) \mathbf{Q}_{ji}(n)$$

$$\mathbf{Q}_{ji}(n) = [\mathbf{x}_j - \mathbf{t}_i(n)][\mathbf{x}_j - \mathbf{t}_i(n)]^T$$

$$\boldsymbol{\Sigma}_i^{-1}(n+1) = \boldsymbol{\Sigma}_i^{-1}(n) - \eta_3 \frac{\partial \mathscr{E}(n)}{\partial \boldsymbol{\Sigma}_i^{-1}(n)}$$

[a]The term $e_j(n)$ is the error signal of output unit j at time n. The term $G'(\cdot)$ is the first derivative of the Green's function $G(\cdot)$ with respect to its argument.

In RBFN, every individual input 'x' might be weighted.

Therefore,

$$\|x\|_c^2 = (Cx)^T (Cx)$$
$$= x^T C^T C x$$
$$C = m - by - m \;\; weighting \;\; matrix$$
$$m = \text{dim}ension \;\; of \;\; input \;\; vector \;\; x$$

Now,

$$G(\|x - t_i\|)_c = e^{-(x-t_i)^T C^T C (x-t_i)}$$
$$= e^{-\frac{1}{2}(x-t_i)^T \mathrm{E}^{-1} (x-t_i)}$$

$$where, \; \mathrm{E} \; is \;\; non \;\; directional \;\; definite \;\; positive \;\; matrix$$

Refer to Section 5.7, Haykin for more definitions of E.

Some guidelines for RBFN's training

- Training for centers and spreads is apparently very slow
- Thus some have taken the approach of computing these parameters by other means and just training for the weights (at most).

***A* Solving Approach for RBF**

- Consider the spreads are fixed.
- Choose the N data points themselves as centers.
- It remains to find the weights.
- Define $\varphi_{ji} = \varphi(|| x_i - x_j ||)$ where φ is the radial basis function, xi, xj are training samples.
- The matrix Φ of values φ_{ji} is called the **interpolation matrix**.
- The **interpolation matrix** has the property that $\Phi.\mathbf{w} = \mathbf{d}$

  Where

  w is the weight vector

  d is the desired output vector over all training samples (since the samples are both data points and centers).

  (If Φ is non-singular, then we can **solve** for weights as $\mathbf{w} = \Phi^{-1}\mathbf{d}$).

Comparison of RBF Networks and MultiLayer Perceptrons

Radial-basis function (RBF) networks and multilayer perceptrons are examples of nonlinear layered feedforward networks. They are both universal approximators. It is therefore not surprising to find that there always exists an RBF network capable of accurately mimicking a specified MLP, or vice versa. However, thse two networks differ from each other in several important respects;

1. An RBF network (in its most basic form) has a single hidden layer, whereas an MLP may have multiple hidden layers.
2. Computational nodes in MLP share a common neuronal model but in RBF, computational functions are different at different layers.
3. The hidden layer of RBF network is nonlinear but output layer is linear. However, the hidden and output layers of MLP, for pattern classification, are non-linear.
4. The argument of the activation function of each hidden unit in an RBF network computes the *Euclidean norm (distance)* between input vector and the center of that unit. In AMLP, activation function computes the inner product of input vector and synaptic weight.
5. MLPs construct global approximations to nonlinear input-output mapping. RBF networks use exponentially decaying localized nonlinearities construct local approximations to non-linear input-output mappings.
6. The no. of parameters required for MLPs are smaller than RBF network.

Source	Application	MLP	RBF
Dong	Satellite image classification		Faster runtime
Finan	Speaker recognition		More accurate, less sensitive to bad training data
Hawickhorst	Speech recognition		Faster training, better retention of generalization
Li	Surgical decision making	Fewer hidden nodes	Shorter training time, lower errors
Lu	Channel Equalization	Statistically insignificant differences	
Park	Nonlinear system identification		Better convergence to global min., less retraining time
Roppel	Odor recognition	Higher identification rates	

"Probablistic" Neural Networks
A "Probabilistic" Neural Network (PNN) is the name given to a radial-basis function network modified for **classification** purposes.

The linear output layer is followed by a **competitive** layer which makes a **classification** based on the RBF unit with the **largest output**.

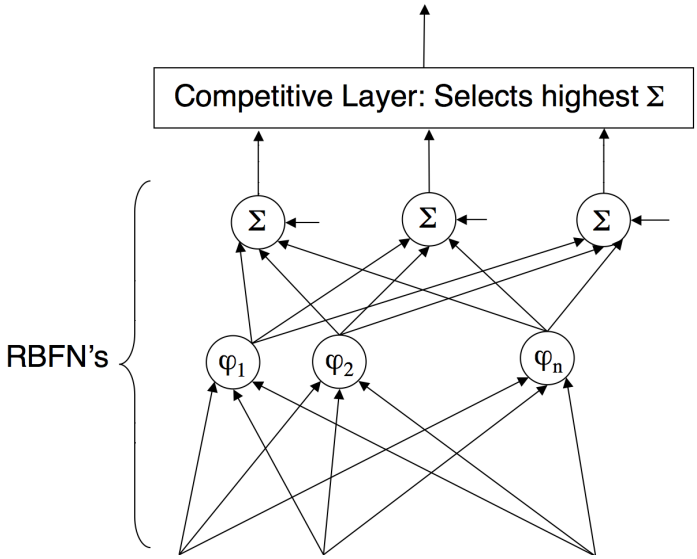


Fig. 14: Probabilistic Neural Network with Competitive Layers

Hands-on practice on Matlab

Let's try with default settings of RBFNs.

```
P = [1 2 3];
T = [2.0 4.1 5.9];
net = newrb(P,T);
X = [0.5 0.9 2.7];
Y = sim(net, X)
      Y =
            2.6503    2.0260    5.6073
plot(P, T, X, Y, 'o')
```

Notice the data points.

Now try with small spread; spread = 0.01

```
P = [1 2 3];
T = [2.0 4.1 5.9];
net = newrb(P, T, 0.0, 0.01);
X =[0.5 0.9 2.7];
Y = sim(net, X)
      Y =
            5.9000    5.9000    5.9000
plot(P, T, X, Y, 'o')
```

Check the new data points with previous one.

Let's try with larger spread, spread = 100.

```
P = [1 2 3];
T = [2.0 4.1 5.9];
net = newrb(P,T, 0.0, 100);
NEWRB, neurons = 0, MSE = 2.54
X =[0.5 0.9 2.7];
Y = sim(net, X)
      Y =
          0.8378    1.7735    5.3915
plot(P, T, X, Y, 'o')
```

Sample application of RBFN for Face Recognition

Reference: Powell, et al. at University of Sussex
(from RBF_application.pdf in ./04-LearningMechanisms/RBF)

Cascade-Correlation Networks

Cascade-Correlation is a new architecture and supervised learning algorithm for artificial neural networks. Instead of just adjusting the weights in a network of fixed topology, Cascade-Correlation begins with a minimal network, then automatically trains and adds new hidden units one by one, creating a multi-layer structure. Once a new hidden unit has been added to the network, its input-side weights are frozen. This unit then becomes a permanent feature-detector in the network, available for producing outputs or for creating other, more complex feature detectors. The Cascade-Correlation architecture has several advantages over existing algorithms: it learns very quickly, the network determines its own size and topology, it retains the structures it has built even if the training set changes, and it requires no back-propagation of error signals through the connections of the network.

Why is Back-Propagation Learning So Slow?

The Cascade-Correlation learning algorithm was developed in an attempt to overcome certain problems and limitations of the popular back-propagation (or "backprop") learning algorithm. The most important of these limitations is the slow pace at which backprop learns from examples. Even on simple benchmark problems, a back-propagation network may require many thousands of epochs to learn the desired behavior from examples. (An *epoch* is defined as one pass through the entire set of training examples.) There are two major problems that contribute to the slowness. We call these the *step-size problem* and the *moving target problem*.

The Step-Size Problem

The step-size problem occurs because the standard back-propagation method computes only $\partial E / \partial w$, the partial first derivative of the overall error function with respect to each weight in the network. Given these derivatives, we can perform a gradient descent in weight space, reducing the error with each step. It is straightforward to show that if we take infinitesimal steps down the gradient vector, running a new training epoch to recompute the gradient after each step, we will eventually reach a local minimum of the error function. Experience has shown that in most situations this local minimum will be a global minimum as well, or at least a "good enough" solution to the problem at hand.

In a practical learning system, however, we do not want to take infinitesimal steps; for fast learning, we want to take the largest steps that we can. Unfortunately, if we choose a step size that is too large, the network will not reliably converge to a good solution. In order to choose a reasonable step size, we need to know not just the slope of the error function, but something about its higher-order derivatives—its curvature—in the vicinity of the current point in weight space. This information is not available in the standard back-propagation algorithm.

The Moving Target Problem

A second source of inefficiency in back-propagation learning is what we call the moving target problem. Briefly stated, the problem is that each unit in the interior of the network is trying to evolve into a feature detector that will play some useful role in the network's overall computation, but its task is greatly complicated by the fact that all the other units are changing at the same time. The hidden units in a given layer of the net cannot communicate with one another directly; each unit sees only its inputs and the error signal propagated back to it from the network's outputs. The error signal defines the problem that the unit is trying to solve, but this problem changes constantly. Instead of a situation in which each unit moves quickly and directly to assume some useful role, we see a complex dance among all the units that takes a long time to settle down.

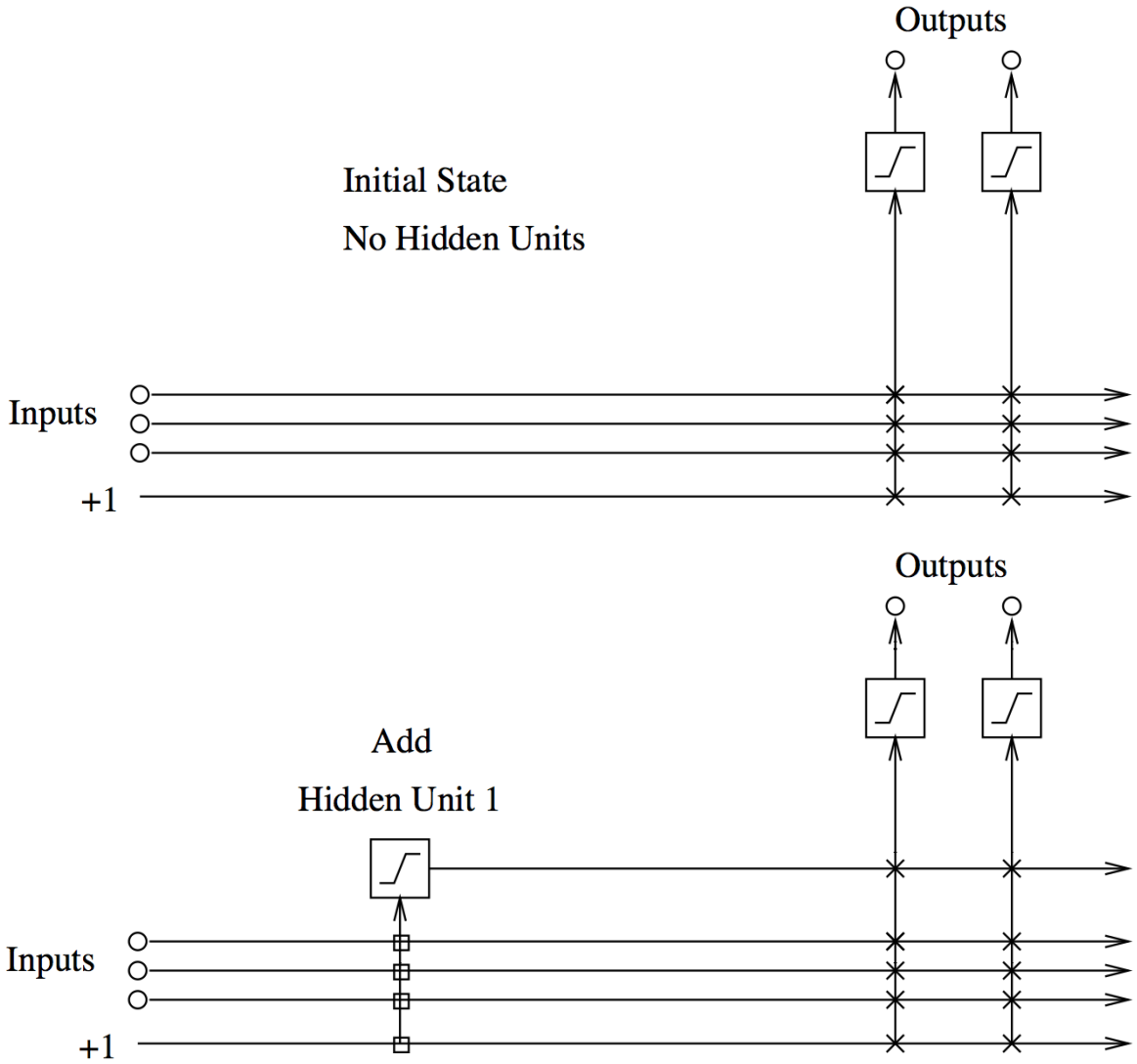
Many experimenters have reported that backprop learning slows down dramatically (perhaps exponentially) as we increase the number of hidden layers in the network. In part, this slowdown is due to an attenuation and dilution of the error signal as it propagates

backward through the layers of the network. We believe that another part of this slowdown is due to the moving-target effect. Units in the interior layers of the net see a constantly shifting picture as both the upstream and downstream units evolve, and this makes it impossible for such units to move decisively toward a good solution.

Description of Cascade-Correlation

Cascade-Correlation combines two key ideas: The first is the *cascade architecture*, in which hidden units are added to the network one at a time and do not change after they have been added. The second is the learning algorithm, which creates and installs the new hidden units. For each new hidden unit, we attempt to maximize the magnitude of the *correlation* between the new unit's output and the residual error signal we are trying to eliminate.

The cascade architecture is illustrated in Figure 15. It begins with some inputs and one or more output units, but with no hidden units. The number of inputs and outputs is dictated by the problem and by the I/O representation the experimenter has chosen. Every input is connected to every output unit by a connection with an adjustable weight. There is also a *bias* input, permanently set to +1.



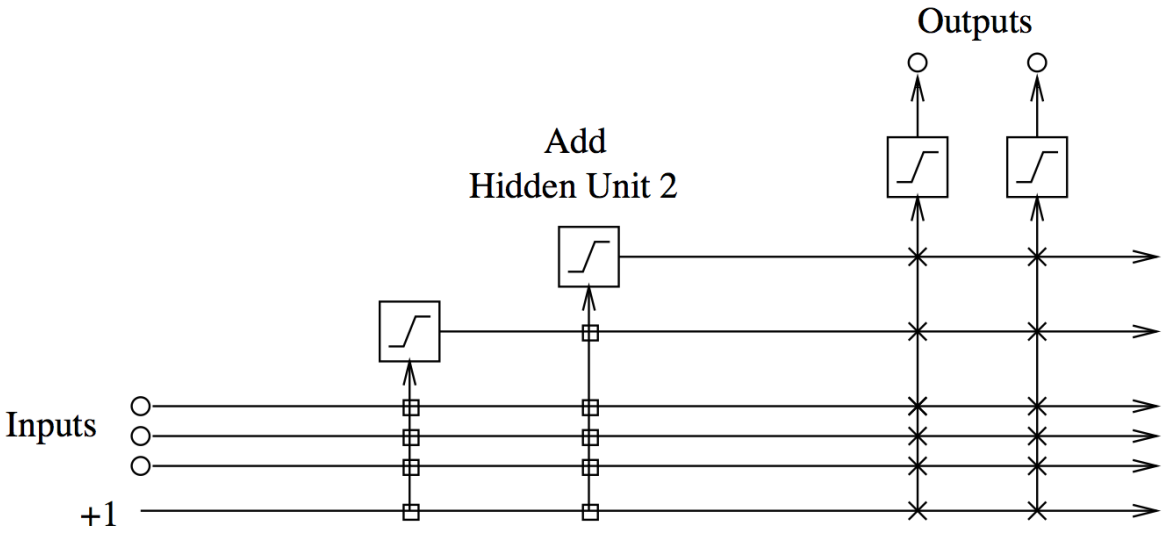


Fig. 15: Architecture of Cascade-Correlation Network

The output units may just produce a linear sum of their weighted inputs, or they may employ some non-linear activation function (symmetric sigmoidal activation function - hyperbolic tangent) whose output range is -1.0 to +1.0.

We add hidden units to the network one by one. Each new hidden unit receives a connection from each of the network's original inputs and also from every pre-existing hidden unit. The hidden unit's input weights are frozen at the time the unit is added to the net; only the output connections are trained repeatedly. Each new unit therefore adds a new one-unit "layer " to the network, unless some of its incoming weights happen to be zero. This leads to the creation of very powerful high-order feature detectors; it also may lead to very deep networks and high fan-in to the hidden units.

The learning algorithm begins with no hidden units. The direct input-output connections are trained as well as possible over the entire training set. With no need to back-propagate through hidden units, we can use the Widrow-Hoff or "delta" rule, the Perceptron learning algorithm, or any of the other well-known learning algorithms for single-layer networks.

At some point, the network will have no significant error reduction. At that point, the training process stops.

Steps

1. Start with minimum network having only the necessary input and output units with bias input always equal to +1.
2. The net is trained until no further processing is possible. Then error is calculated.
3. Then hidden unit is added to the net in two step processes.
4. In first step, the candidate input is connected each of the input units, but is not connected to the output units.
    a. The weights from the input unit to the candidate unit are adjusted
    b. The adjustment is made to maximize the correlation between the candidate's output and the residual error at the output units.
    c. The residual error is the difference between the target and the computed output, which is being multiplied by the derivative of the output units' activation function.
    d. After the training process is completed, the weights are frozen and the candidate unit becomes a hidden unit in the net.
5. In second step, the new unit added to the net starts processing.
    a. The new hidden unit is connected to the output units and the weights on the connection are adjustable

b. Now the connections to the output units are trained.

The connection from the input units are trained again and the new connection from the hidden unit are trained for the first time.

6. The process of adding a new unit, training its connection weights from the input units and previously added hidden units, and hidden freezing all the weights, then training the connections to the output units, is continued until the error reaches an acceptable level or the required number of epoch.

Training algorithm

The training of CCN involves both adjusting the weights and modifying the architecture of the net.

The parameters involved are;

n – dim*ension of input vector*

m – dim*ension of output vector*

p – *total number of training patterns*

x_i – *input units* $i = 1$ *to* n

y_j – *output units* $j = 1$ *to* m

$x(p)$ – *training input vector* $p = 1$ *to* P

$t(p)$ – *target vector for input vector* $x(p)$

$y(p)$ – *computed output for input vector* $x(p)$

The residual error for output unit is;

$$E_j(p) = y_j(p) - t_j(p)$$

The average residual error for output unit is;

$$E_{av_j} = \frac{1}{p}\sum_{p=1}^{P} E_j(p)$$

$$Z_{av} = \frac{1}{p}\sum_{p=1}^{P} z(p)$$

$z(p)$ = *activation of candidate unit for input vector*

z_{av} = *average activation of candidate unit*

The correlation is defined as;

$c = \sum_{j=1}^{m} \left| \sum (z(p) - z_{av})(E_j(p) - E_{av_j}) \right|$, the correlation between candidates output and networks output.

Note: Mathematically, correlation between two variables is defined as;

$$C_{xy} = \sum xy - \frac{(\sum x)(\sum y)}{n}$$

The maximization of 'C' is done through a gradient descent calculated after computation of the following partial derivative;

$$\frac{\partial C}{\partial w_i} = \sum_{j,p} \sigma_p \ (E_j(p) - E_{av}) \ a'_j \ I_{i,j}$$

where, $a_j^{'}$ is the derivative for pattern 'p' of the candidate neuron's activation function with respect to the sum of its inputs
$I_{i,j}$ is the input the candidate neuron receives from neuron 'i' for pattern 'p'.

The computation of $\frac{\partial C}{\partial w_i}$ is achieved by the use of the chain rule.

The complete training algorithm is as follows;

1. Start with required input and output units
2. Train the net upto the error reaches a minimum
   - If the error is negligible, stop.
   - Else, compute $E_j(p)$ for each training pattern, then E-av$_j$ and move to step 3.
3. Then add first hidden unit.
4. A candidate unit 'z' is connected to each input unit.
   - Initialize weights from input units to 'z'
5. Train these weights to maximize 'C'
   - When no change in weights, they are frozen.
6. Train all weights θ to the output units
   - If acceptable error is reached, then stop else move to step 7.
7. While stopping condition is false, do steps 8-10 (add another hidden unit)
8. A candidate unit 'Z' is connected to each input unit and each previously added hidden unit.
9. Train these weights to maximize 'C'
   - When these weights stop changing, they are frozen.
10. Train all the weights 'U' to the output units.
    - If acceptable error or minimum number of units has been searched, stop.
    - Else proceed.

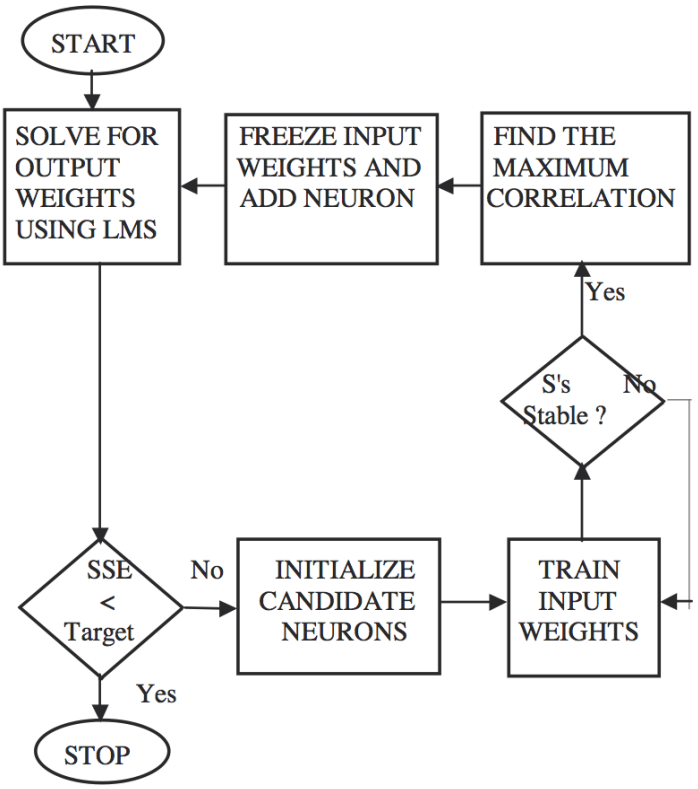


Fig. 16: CCN training algorithm

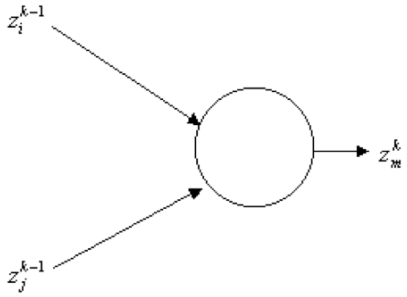
Polynomial (GMDH) Networks

Group Method of Data Handling (GMDH) polynomial neural networks are "self organizing" networks. The network begins with only input neurons. During the training process, neurons are selected from a pool of candidates and added to the hidden layers.

GMDH networks are self-organizing. This means that the connections between neurons in the network are not fixed but rather are selected during training to optimize the network. The number of layers in the network also is selected automatically to produce maximum accuracy without over-fitting.

Prior to detailed back-propagation technique, GMDH was developed that uses a more complex neuron featuring a polynomial transfer function. The interconnections between layers of neurons were simplified, and an automatic algorithm for structure design and weight adjustment was developed.

A single neuron is modeled as;



and it's output as;

$$z_m^k = a_m^k\left(z_i^{k-1}\right)^2 + b_m^k\left(z_i^{k-1}\right)\left(z_j^{k-1}\right) + c_m^k\left(z_j^{k-1}\right)^2 + d_m^k z_i^{k-1} + e_m^k z_j^{k-1} + f_m^k$$

A GMDH neural network maps a vector input **x** to a scalar output y'.

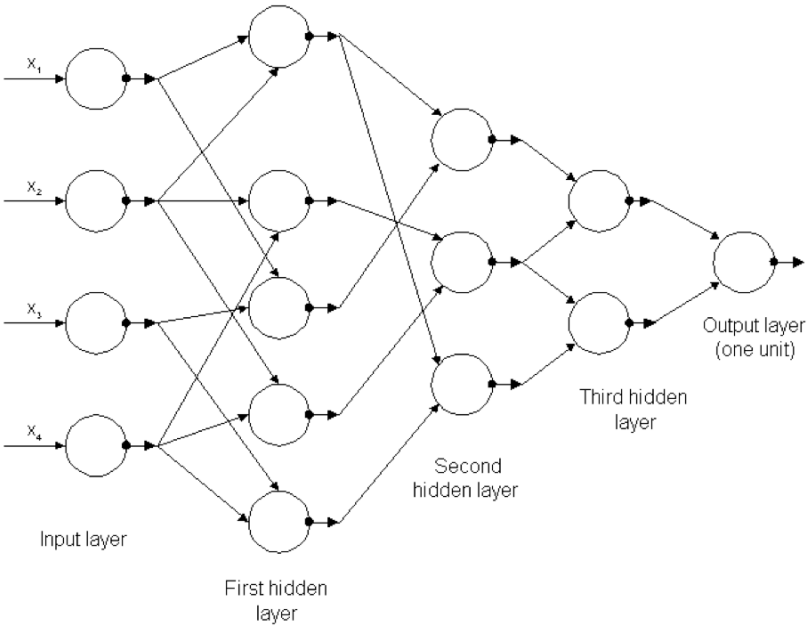


Fig. 17: GMDH Neural Network

The output y' is an estimate of the true function f(x) = y.

GMDH Training algorithm

Two sets of input data are used during the training process: (1) the primary training data, and (2) the control data which is used to stop the building process when overfitting occurs. The control data typically has about 20% as many rows as the training data. The percentage is specified as a training parameter.

The GMDH network training algorithm proceeds as follows:

1. Construct the first layer which simply presents each of the input predictor variable values.
2. Using the allowed set of functions, construct all possible functions using combinations of inputs from the previous layer. If only two-variable polynomials are enabled, there will be n*(n-1)/2 candidate neurons constructed where 'n' is the number of neurons in the previous layer. If the option is selected to allow inputs from the previous layer and the input layer, then 'n' will the sum of the number of neurons in the previous layer and the input layer. If the option is selected to allow inputs from any layer, then 'n' will the sum of the number of input variables plus the number of neurons in all previous layers.
3. Use least squares regression to compute the optimal parameters for the function in each candidate neuron to make it best fit the training data. Singular value decomposition (SVD) is used to avoid problems with singular matrices. If nonlinear functions are selected such as logistic or asymptotic, a nonlinear fitting routine based on Levenberg-Marquardt method is used.
4. Compute the mean squared error for each neuron by applying it to the *control data*.
5. Sort the candidate neurons in order of increasing error.
6. Select the best (smallest error) neurons from the candidate set for the next layer. A model-building parameter specifies how many neurons are used in each layer.
7. If the error for the best neuron in the layer as measured with the control data is better than the error from the best neuron in the previous layer, and the maximum number of layers has not been reached, then jump back to step 2 to construct the next layer. Otherwise, stop the training. Note, when overfitting begins, the error as measured with the control data will being to increase, thus stopping the training.

Advantage

Support for pruning the neurons.