

## Notes on Radial Basis Function approach in Neural Networks

Neural element is defined differently:

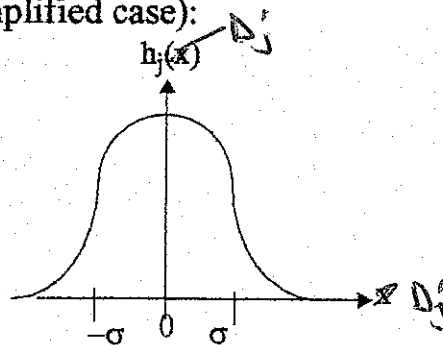
- a) Input portion: instead of the usual procedure of performing a dot product of the input ( $\underline{x}_i$ ) and weight ( $\underline{w}_j$ ) vectors, perform the following operation:

$$\text{Define } D_j^2 = (\underline{x}_i - \underline{w}_j)^T (\underline{x}_i - \underline{w}_j)$$

- b) Output portion: instead of a sigmoid activation function, use an exponential function (e.g., Gaussian):

$$\text{Define } h_j = \exp\left[\frac{-D_j^2}{2\sigma^2}\right]$$

For case of 1 input (simplified case):

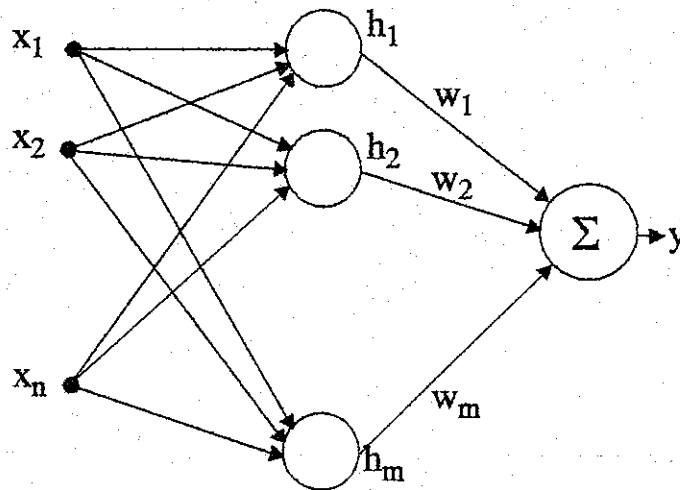


Thus, the output of element  $j$  has significant response to input  $x$  only over a range of values of  $x$ , called the receptive field of the neuron, which is determined by the value of  $\sigma$ .

For the RBF concept, activation functions other than exponential may be used. The major requirement is that the chosen function tend to zero "quite rapidly" as distance between input vector  $\underline{x}_i$  and mean of vector  $\underline{w}_j$  increases.

The key reason for using the exponential functions is a mathematical result (Girosi & Poggio, 1990) that they are in a class of functions possessing the "best approximation property," that is, there exists a set of weights that approximates the desired function better than any other set. This attribute is not shared by the sigmoidal functions.

Network configuration (more than one output element may be used):



The “instar” weight vector for each of the  $h_j$  elements is set equal to one of the train vectors  $\underline{x}_i$ . This essentially makes the corresponding exponential function to be centered on the train vector. The remaining parameter to be adjusted is the  $\sigma$  of each  $h_j$ .

Following these adjustments, the TRAINING of the NN only needs to determine values of the  $w_j$  to yield satisfactory outputs. We note that this turns out being a single-layer training, which reverts back to a simple Adaline kind of training -- i.e., we may have a simple quadratic surface in the weight space, and thus a fast training is likely, usually substantially faster than the multi-layer (feedforward) structure.

---

Regarding the selection of the parameter  $\sigma$  for each  $h_j$ , this could be accomplished via a search process of its own, and there exist strategies for finding “optimal” values for them. In general however, this would be computationally expensive, so the following heuristic is often good enough:

1. For each hidden layer neuron, find the RMS distance between its center and the center of its  $N$  nearest neighbors.
  2. Assign this value to the  $\sigma$  for that unit.
-

### Output-layer Weights:

Once the centers and  $\sigma$ 's have been CHOSEN, the output layer weights can be optimized (modulo these choices) via supervised training.

#### PROCESS:

[0. Set the instar weight vectors of each of the hidden units equal to one of the train vectors. Set the  $\sigma$  for each hidden unit (e.g., via K nearest-neighbor heuristic).]

1. Apply a train input vector  $\underline{x}_i$
2. Calculate outputs of hidden layer neurons,  $\underline{h}$
3. Calculate network output (vector, if more than one output element)  $\underline{y}$

Compare  $\underline{y}$  with the target (desired) output  $\underline{t}$

Adjust weights to reduce the difference, say, via the standard Delta rule:

$\Delta w_{ij} = \eta(t_i - y_i)h_j$ , where  $w_{ij}$  are weights from hidden neuron  $j$  to output neuron  $i$ ,  $t_i$  are the target outputs,  $y_i$  are the calculated outputs, and  $\eta$  is the train/learn rate.

4. Repeat steps 1-3 for each vector in the train set.
5. Repeat 1-4 until error is acceptably small, or some other terminating condition.

---

Note: In principle, one could calculate the output-layer weights rather than "train" them.

First, create a matrix  $\underline{H}$  where each row contains the outputs of the hidden-layer elements corresponding to one of the input vectors  $\underline{x}_i$ .

Next, create a matrix  $\underline{T}$  where each row contains the target outputs for the corresponding row of  $\underline{H}$ .

Construct the weight matrix  $\underline{W}$  such that each set of weights ("instar") to one of the output layer neurons is made a column of  $\underline{W}$ .

Then,  $\underline{T} = \underline{HW}$ , or,  $\underline{W} = \underline{H}^{-1}\underline{T}$ . Note that  $\underline{H}$  may not be square, so the inverse would have to be obtained via the "pseudo inverse" procedures.

[Cf. method developed by Lendaris, Mathia & Saeks (1995) to use Linear Hopfield NN to perform the pseudo-inverse.]