Major Classes of Neural Networks

Outline

- Multi-Layer Perceptrons (MLPs)
- Radial Basis Function Network
- Kohonen's Self-Organizing Network
- Hopfield Network

Multi-Layer Perceptrons (MLPs)

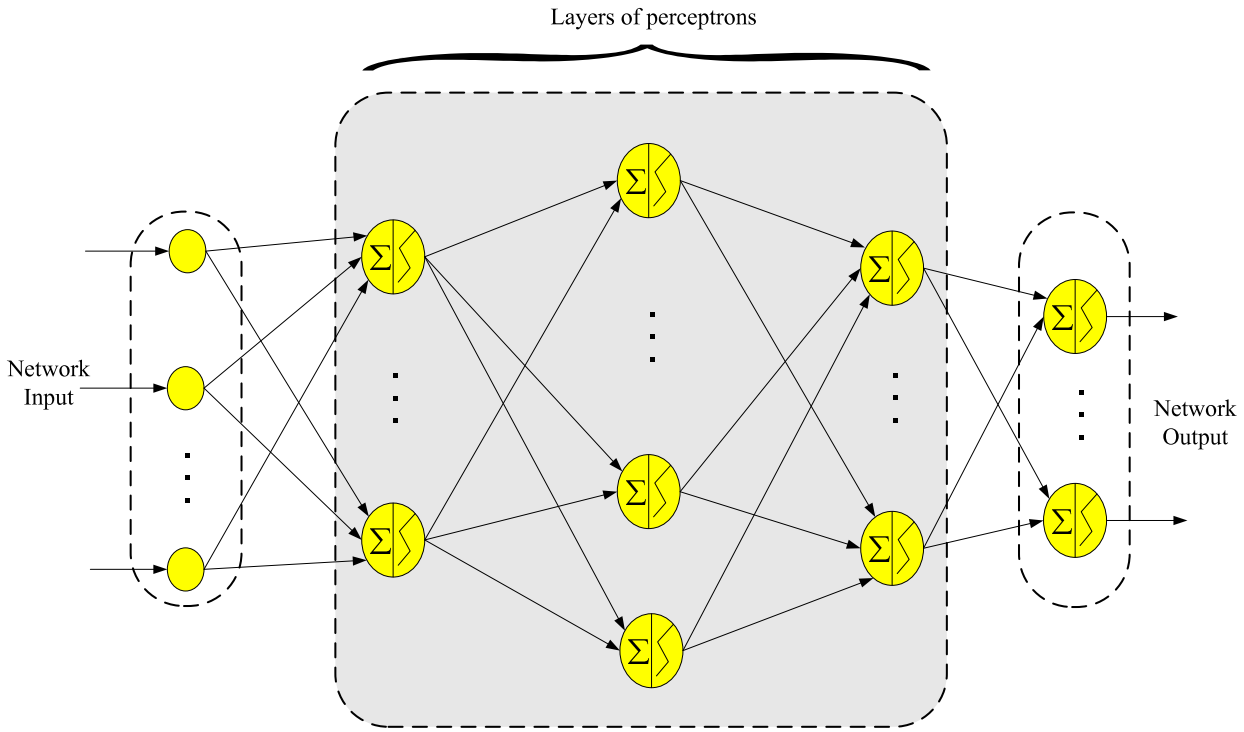
Background

- The perceptron lacks the important capability of recognizing patterns belonging to non-separable linear spaces.
- The madaline is restricted in dealing with complex functional mappings and multi-class pattern recognition problems.
- The multilayer architecture first proposed in the late sixties.

Background (cont.)

- MLP re-emerged as a solid connectionist model to solve a wide range of complex problems in the mid-eighties.
- This occurred following the reformulation of a powerful learning algorithm commonly called the Back Propagation Learning (BPL).
- It was later implemented to the multilayer perceptron topology with a great deal of success.

Schematic Representation of MLP Network



This is the struture of the neural network. It is the most general structure. An input layer is where signals enter the system. These are usually the features of an object. The yellow circles are just units. We call them perceptrons, this is a bit misleading. These perceptrons are different from the ones we saw earlier. The earlier ones create discrete values but they instead (most of the time) they have a smooth, nonlinear activation function. Their activation functions are very differentiable. They have the same structure though. The output layer generates the actual output of the system.

Backpropagation Learning Algorithm (BPL)

- The backpropagation learning algorithm is based on the **gradient descent technique** involving the **minimization of the network cumulative error**.

$$E(k) = \sum_{i=1}^{q} [t_i(k) - o_i(k)]^2$$

  - i represents i-th neuron of the output layer composed of a total number of q neurons.

- It is designed to **update the weights in the direction of the gradient descent of the cumulative error**.

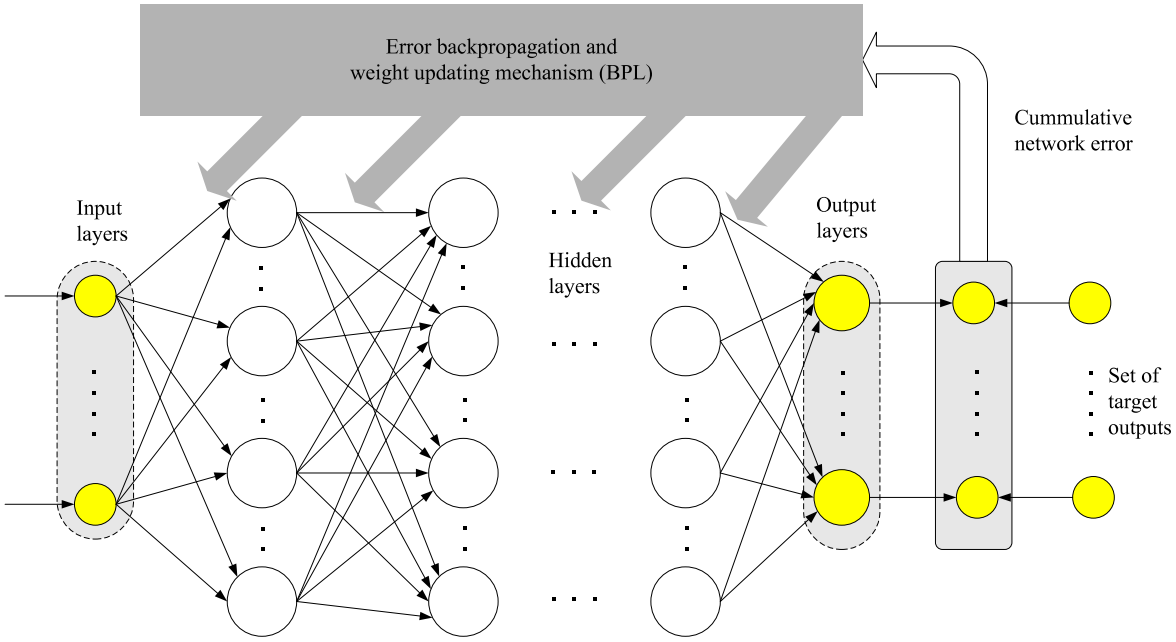
Backpropagation Learning Algorithm (cont.)

A Two-Stage Algorithm

1. First, patterns are presented to the network.
2. A feedback signal is then propagated backward with the main task of updating the weights of the layers connections according to the back-propagation learning algorithm.

BPL: Schematic Representation

- Schematic Representation of the MLP network illustrating the notion of error back-propagation



The multilayer perceptron has many layers to propagate signals. Most of the time, these kinds of models are supervised (we have to teach them).

This is still an optimization problem. We have some weights that we want to optimize our weights so that they have the least error. These kinds of systems have a ton of dimensions to optimize across. There will be very complex structures which make it very hard to actually solve. This is why we tend towards using other systems to get an optima even if its not the global optima.

When a solution is found (weights are known) we use this to find the weights of the next layer. This backpropagates the error signal through the layers backwards untill it reaches the first layer. You keep feeling in training signals until you run out and update all the weights. At this point we have finished one epoch.

Say you have $\overrightarrow{x}(x_1, ..., x_m)$ then you have the output $\overrightarrow{y}(y_1, ..., y_l)$ so we then get the mapping $\overrightarrow{y} = f(\overrightarrow{x})$. So your first training signal has to be a bunch of mappings of x to y. Roughly k signals.

Backpropagation Learning Algorithm (cont.)

Objective Function

- Using the **sigmoid function** as the activation function for all the neurons of the network, we define E_c as

$$E_c = \sum_{k=1}^{n} E(k) = \frac{1}{2} \sum_{k=1}^{n} \sum_{i=1}^{q} [t_i(k) - o_i(k)]^2$$

Here i is the index of the element we are looking at and k is the index of the training set we are currently looking at. This makes an error function. We compute the sum of the square error for each node and propagate it backwards. We keep going backwards until we reach the communication layer (one right after the input layer).

Backpropagation Learning Algorithm (cont.)

- The formulation of the **optimization problem** can now be stated as **finding the set of the network weights** that minimizes E_c or $E(k)$.

Objective Function: Off-Line Training

$$min_w E_c = min_w \frac{1}{2} \sum_{k=1}^{n} \sum_{i=1}^{q} [t_i(k) - o_i(k)]^2$$

Objective Function: On-Line Training

$$min_w E(k) = min_w \frac{1}{2} \sum_{i=1}^{q} [t_i(k) - o_i(k)]^2$$

BPL: On-Line Training

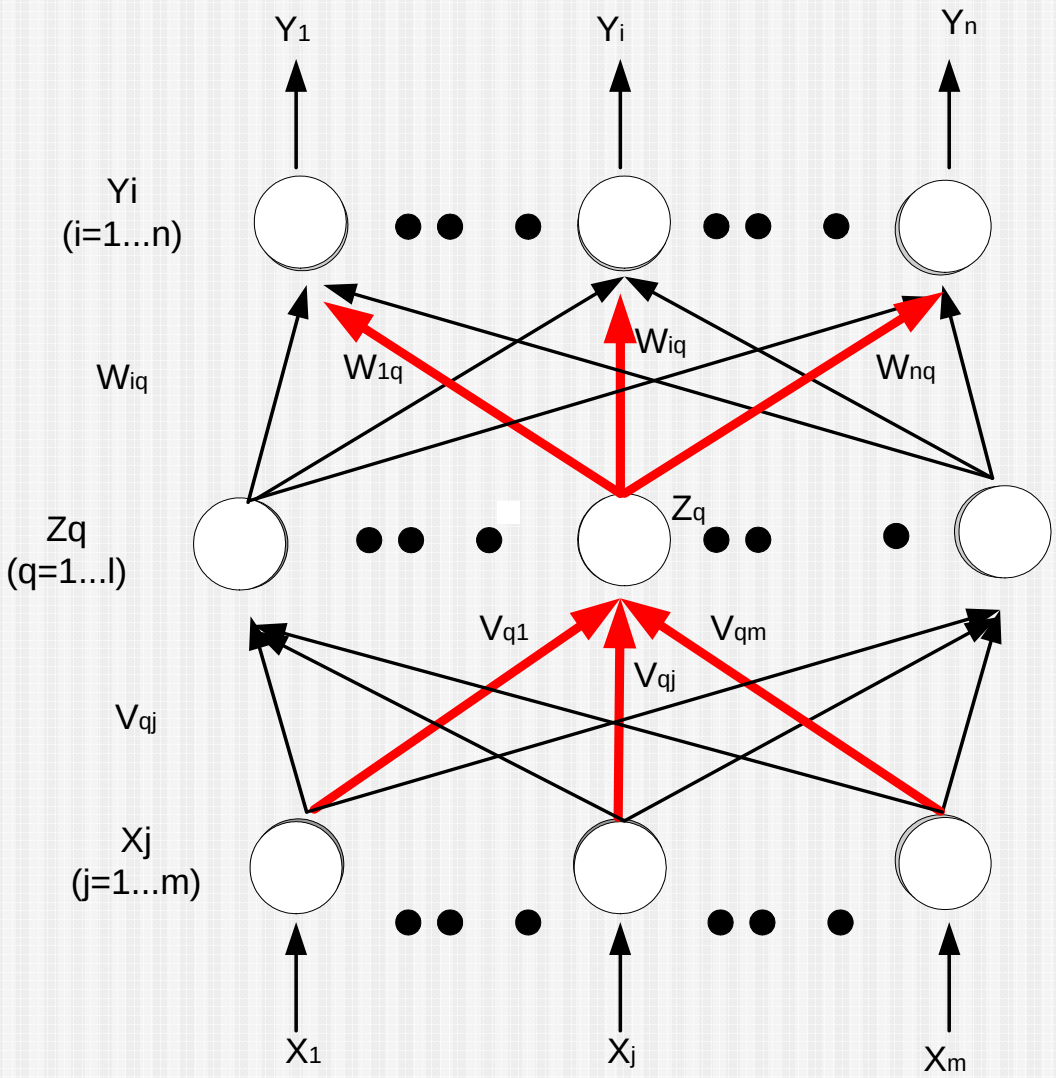
- Objective Function: $min_w E(k) = min_w \frac{1}{2} \sum_{i=1}^{q} [t_i(k) - o_i(k)]^2$

Updating Rule for Connection Weights

$$\Delta w^{(l)} = -\eta \frac{\partial E(k)}{\partial w^l},$$

- l is layer (l-th) and η denotes the learning rate parameter,
- $\Delta w_{ij}^{(l)}$: the weight update for the connection linking the node j of layer $(l-1)$ to node i located at layer l.

Here is the update function for the weights. It looks pretty standard. We work layer by layer.


Y1
Yi
Yn
Yi
(i=1...n)
Wiq
W1q
Wiq
Wnq
Zq
Zq
(q=1...l)
Vq1
Vqm
Vqj
Vqj
Xj
(j=1...m)
X1
Xj
Xm

Here we have input layer x with m nodes in it, middle layer z with l nodes in it, and output layer y with n nodes in it. There are three weights going between x and z labeled with v, there are other ones but we are going to ignore them for simplicity in this example. There are other weights from z to y labeled as w, again there are others that we are just ignoring to make the example easier to go through. The subscript on the notation is the destination node,origin node notation. So how do we find the weights?

$$\begin{aligned}
a &= \text{activation function} \\
net_q &= \sum_{j=1}^{n} v_{qj} x_j \\
z_q &= a(net_q) \\
&= a(\sum_{j=1}^{n} v_{qj} x_j) \\
net_i &= \sum_{q=1}^{l} w_{iq} z_q \\
y_i &= a(net_i) \\
y_i &= a(\sum_{q=1}^{l} w_{iq} z_q) \\
y_i &= a(\sum_{q=1}^{l} w_{iq} a(\sum_{j=1}^{n} v_{qj} x_j))
\end{aligned}$$

In this case the two a functions do not have to be the same but its much easier if they are.

What about the target data (denoted d_i)?

$$E = \frac{1}{2} \sum_{i=1}^{n} (d_i - y_i)^2$$

Remember we want gradient decay that eventually minimizes the error, after all this is an optimization problem.

Now we need to find our update functions for the shit ton of weights we have:

$$
\begin{aligned}
\Delta w_{iq} &= -\eta \nabla_{w_{iq}} E(w) \\
&= -\eta \frac{\delta E}{\delta w_{iq}} \\
&= -\eta \frac{\delta E}{\delta y_i} \frac{\delta y_i}{\delta net_i} \frac{\delta net_i}{\delta w_{iq}} \\
&= \eta (d_i - y_i) a'(net_i) z_q \\
&= \eta \delta_{oi} z_q \\
\delta_{oi} &= (d_i - y_i) a'(net_i) \\
\Delta v_{qj} &= -\eta \nabla_{v_{qj}} E(w) \\
&= -\eta \frac{\delta E}{\delta v_{qj}} \\
&= -\eta \frac{\delta E}{\delta net_q} \frac{\delta net_q}{\delta v_{qj}} \\
&= -\eta \frac{\delta E}{\delta z_q} \frac{\delta z_q}{\delta net_q} \frac{\delta net_q}{\delta v_{qj}} \\
&= \eta \sum_{i=1}^{n} \left((d_i - y_i) a''(net_i) w_{iq} \right) a'(net_q) x_j \\
&= \eta \delta_{hq} x_j \\
\delta_{hq} &= a'(net_q) \sum_{i=1}^{n} \delta_{oi} w_{iq}
\end{aligned}
$$

This is the output for only one hidden layer, more work is required for more layers.

We snag E and keep it constant though all the layers and let it run until E updates with a new training signal. So the E value should be the same through out all the udpates.

To see the full slide of this:

- *ANN (*m *inputs,* n *outputs, 1 hidden layer)*
- *k examples of the form (x,d)*
- *Given an example (x,d),*

$$\text{net}_q = \sum_{j=1}^{m} v_{qj} x_j$$

⇒ *It produces an output*

$$z_q = a(\text{net}_q) = a\left(\sum_{j=1}^{m} v_{qj} x_j\right)$$

⇒ *where* a *is the activation function.*

- *The net input for a perceptron (PE) i in the output layer is given by*

$$\text{net}_i = \sum_{q=1}^{l} w_{iq} z_q = \sum_{q=1}^{l} w_{iq} a\left(\sum_{j=1}^{m} v_{qj} x_j\right)$$

- *The network produces an output*

$$y_i = a(\text{net}_i) = a\left(\sum_{q=1}^{l} w_{iq} a\left(\sum_{j=1}^{m} v_{qj} x_j\right)\right)$$

Let the cumulative error between the desired and the actual outputs be given by

$$E(w) = \frac{1}{2}\sum_{i=1}^{n}(d_i - y_i)^2 = \frac{1}{2}\sum_{i=1}^{n}\left(d_i - a\left(\sum_{q=1}^{l} w_{iq} z_q\right)\right)^2$$

where *E(w)* is a matrix of all the weights.

Using gradient method:

$$\Delta w_{iq} = -\eta \frac{\partial E}{\partial w_{iq}}$$

$$= -\eta \left[\frac{\partial E}{\partial y_i}\right]\left[\frac{\partial y_i}{\partial \text{net}_i}\right]\left[\frac{\partial \text{net}_i}{\partial w_{iq}}\right]$$

$$= \eta \left[d_i - y_i\right]\left[a'(\text{net}_i)\right]\left[z_q\right]$$

$$= \eta \delta_{oi} z_q$$

where δ_{oi} is the error in the i-th output neuron.

$$\delta_{oi} = \frac{\partial E}{\partial \text{net}_i} = \left[d_i - y_i\right]\left[a'(\text{net}_i)\right]$$

$$a'(\text{net}_i) = \frac{\partial a(\text{net}_i)}{\partial \text{net}_i}$$

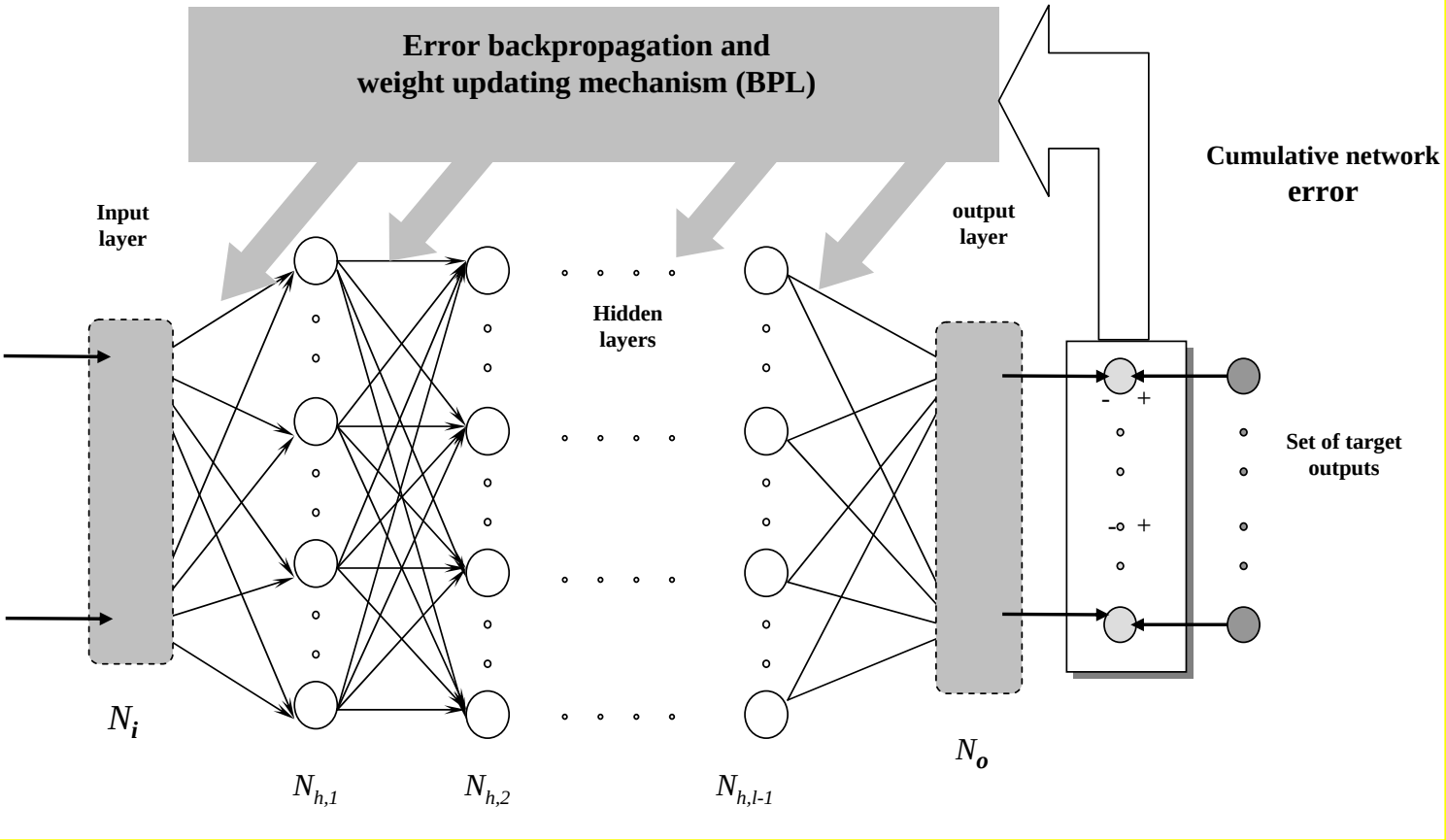
$$\Delta v_{qj} = -\eta\left[\frac{\partial E}{\partial v_{qj}}\right] = \eta\sum_{i=1}^{n}\left[(d_i - y_i)a'(\text{net}_i)w_{iq}\right]a'(\text{net}_q)x_j$$

$$= \eta\delta_{hq}x_j$$

where δ_{hq} is the error in the q-th neuron of the h-th hidden layer.

$$\delta_{hq} = \frac{-\partial E}{\partial \text{net}_q} = a'(\text{net}_q)\sum_{i=1}^{n}\delta_{oi}w_{iq}$$

Backpropagation Algorithm



Consider a ANN with Q *feedforward layers* q=1,…,Q.

$^{q}net_i$ and $^{q}y_i$ *denote the net input and output of the i-th unit in the q-th layer, respectively.*

Let $^{q}w_{ij}$ *denote the connection weight from* $^{q-1}y_j$ *and* $^{q}y_i$.

Input: A set of training pairs of $\left\{(x^{(k)}, d^{(k)}), k = 1,\ldots, p\right\}$

The input vector will be augmented with a last element

$$x_{m+1}^{(k)} = -1$$

Step 0: (Initialization) *Choose* $\eta > 0$ *and a tolerance error* E_{max}. *Initialize the weights to small random values in [0,1].*

Set E=0 *and* k=1.

Now lets say we have an arbitrary number of layers Q, so the z layer in the previous example becomes ${}^{q-1}y$ and the y layer becomes ${}^{q}y$.

The tolerance error is the max error we will allow. We know it will never reach 0 so we instead give it a threshold to work for. Something that is good enough.

Step 1: Apply the k-th *exemplar* (q=1)

$$ {}^{q}y_i = {}^{1}y_i = {}^{(k)}x_i \qquad \textit{for all } \text{i's} $$

Step 2: (Forward propagation) *Propagate the signal forward*

$$ {}^{q}y_i = a({}^{q}\text{net}_i) = a(\sum_j {}^{q}w_{ij}\, {}^{q-1}y_j) \quad \text{for } q = 1,\ldots,Q $$

Step 3: (Output error measure) *Compute the error value and the error signals* ${}^{Q}\delta_i$ *for the output layer*

$$ E = \tfrac{1}{2}\sum_{i=1}^{n}(d_i^{(k)} - {}^{Q}y_i)^2 + E $$

$$ {}^{Q}\delta_i = (d_i^{(k)} - {}^{Q}y_i)a'({}^{Q}\text{net}_i) $$

Step 1: apply your first set of data. Starting at the top layer (where $q + 1$).

Step 2: propagate the signal forward. Keep calculating the output of each layer until you get to the end.

Step 3: calculate the error of the system. The error equation shows two Es. This one being added at the time is the previous error, so its cumulative.

Step 4: (Error backpropagation) *Propagate the error signals backward*

$$\Delta {}^{q}w_{ij} = \eta\, {}^{q}\delta_i\, {}^{q-1}y_j \qquad {}^{q}w_{ij}^{\text{new}} = {}^{q}w_{ij}^{\text{old}} + \Delta {}^{q}w_{ij}$$

$${}^{q-1}\delta_i = a'({}^{q-1}\text{net}_i)\sum_j {}^{q}w_{ji}\, {}^{q}\delta_j \qquad \text{for } q = Q, Q-1, \ldots, 2$$

Step 5: (One epoch looping) *Check whether all the training data has been cycled once.*

If k<p *then* k:= k+1 *and go back to step 1;*

Otherwise go to step 6.

Step 6: (Total error checking)

If $E<E_{max}$ *then process terminated;*

Otherwise ($E \geq E_{max}$*), set* E=0 *and* k=1 *and go back to step 1.*

Step 4: propagate the error signal back. Here we are just updating our weights as we go.

Step 5: just cycle through a nice epoch

Step 6: check if your error is low enough. cycle if not

steps 5 and 6 are basically the same looping conditions of the adeline system.

Required Steps for Backpropagation Learning Algorithm

- Step 1: Initialize weights and thresholds to small random values.
- Step 2: Choose an input-output pattern from the training input-output data set $(x(k), t(k))$.
- Step 3: Propagate the k-th signal forward through the network and compute the output values for all i neurons at every layer (l) using $o_i^l(k) = f(\sum_{p=0}^{n_{l-1}} w_{ip}^l o_p^{l-1})$.
- Step 4: Compute the total error value $E = E(k) + E$ and the error signal $\delta_i^{(L)}$ using formulae $\delta_i^{(L)} = [t_i - o_i^{(L)}][f'(tot)_i^{(L)}]$.

Required Steps for BPL (cont.)

- Step 5: Update the weights according to $\Delta w_{ij}^{(l)} = -\eta \delta_i^{(l)} o_j^{(l-1)}$, for $l = L, \cdots, 1$ using $\delta_i^{(L)} = [t_i - o_i^{(L)}][f'(tot)_i^{(L)}]$ and proceeding backward using $\delta_i^{(l)} = o_i^l(1 - o_i^l) \sum_{p=1}^{n_l} \delta_p^{l+1} w_{pi}^{l+1}$ for $l < L$.
- Step 6: Repeat the process starting from step 2 using another exemplar. Once all exemplars have been used, we then reach what is known as one epoch training.
- Step 7: Check if the cumulative error E in the output layer has become less than a predetermined value. If so we say the network has been trained. If not, repeat the whole process for one more epoch.

Backpropagation Learning Algorithm (cont.)

- The formulation of the **optimization problem** can now be stated as **finding the set of the network weights** that minimizes E_c or $E(k)$.

Objective Function: Off-Line Training

$$min_w E_c = min_w \frac{1}{2} \sum_{k=1}^{n} \sum_{i=1}^{q} [t_i(k) - o_i(k)]^2$$

Objective Function: On-Line Training

$$min_w E(k) = min_w \frac{1}{2} \sum_{i=1}^{q} [t_i(k) - o_i(k)]^2$$

BPL: On-Line Training

- Objective Function: $min_w E(k) = min_w \frac{1}{2} \sum_{i=1}^{q} [t_i(k) - o_i(k)]^2$

Updating Rule for Connection Weights

$$\Delta w^{(l)} = -\eta \frac{\partial E(k)}{\partial w^l},$$

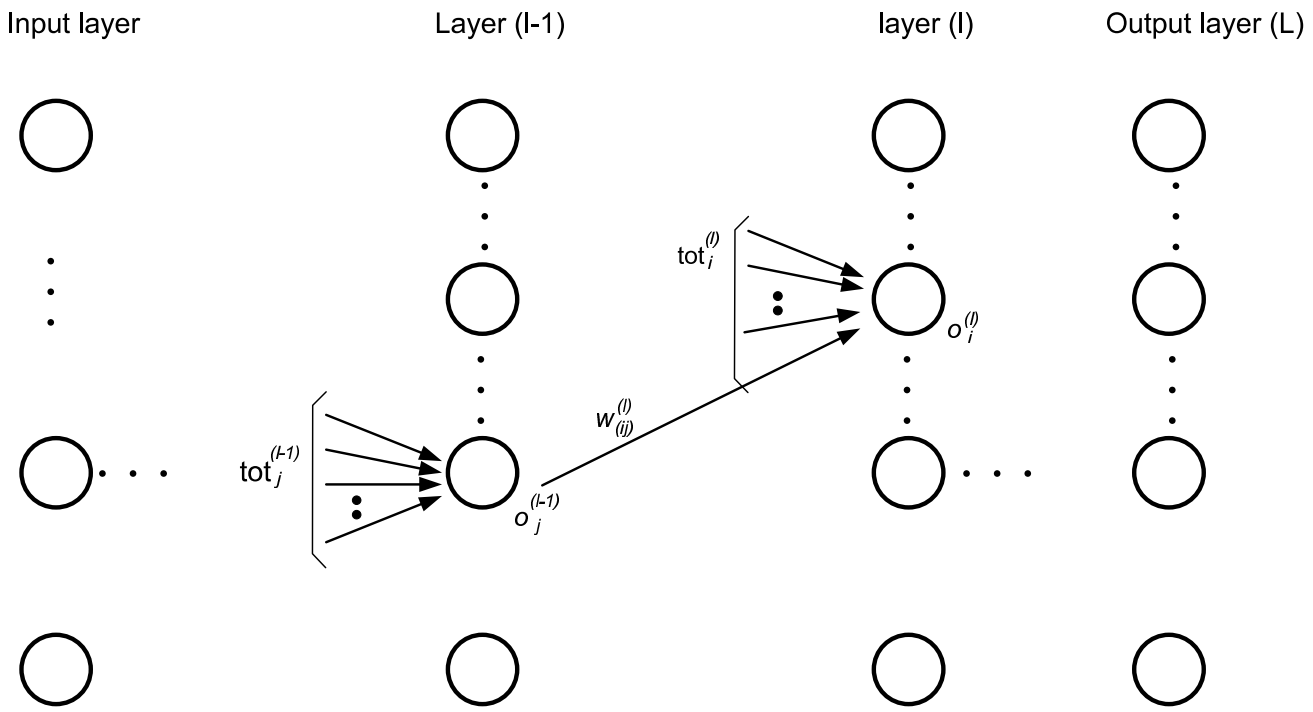
- l is layer (l-th) and η denotes the learning rate parameter,
- $\Delta w_{ij}^{(l)}$: the weight update for the connection linking the node j of layer $(l-1)$ to node i located at layer l.

BPL: On-Line Training (cont.)

Updating Rule for Connection Weights

- o_j^{l-1}: the output of the neuron j at layer $l-1$, the one located just before layer l,
- tot_i^l: the sum of all signals reaching node i at hidden layer l coming from previous layer $l-1$.

Illustration of Interconnection Between Layers of MLP



Interconnection Weights Updating Rules

- $\Delta w^{(l)} = \Delta w_{ij}^{(l)} = -\eta[\frac{\partial E(k)}{\partial o_i^{(l)}}][\frac{\partial o_i^{(l)}}{\partial tot_i^{(l)}}][\frac{\partial tot_i^{(l)}}{\partial w_{ij}^{(l)}}]$
- For the case where the layer (l) is the output layer (L):
  $\Delta w_{ij}^{(L)} = \eta[t_i - o_i^{(L)}][f'(tot)_i^{(L)}]o_j^{(L-1)}; \quad f'(tot)_i^{(l)} = \frac{\partial f(tot_i^{(l)})}{\partial tot_i^{(l)}}$
- By denoting $\delta_i^{(L)} = [t_i - o_i^{(L)}][f'(tot)_i^{(L)}]$ as being the **error signal** of the i-th node of the output layer, the weight update at layer (L) is as follows: $\Delta w_{ij}^{(L)} = \eta\delta_i^{(L)}o_j^{(L-1)}$
- In the case where f is the sigmoid function, the error signal becomes expressed as:
  $\delta_i^L = [(t_i - o_i^{(L)})o_i^{(L)}(1 - o_i^{(L)})]$

Interconnection Weights Updating Rules (cont.)

- Propagating the error backward now, and for the case where (l) represents a hidden layer ($l < L$), the expression of $\Delta w_{ij}^{(l)}$ becomes given by: $\Delta w_{ij}^{(l)} = \eta \delta_i^{(l)} o_j^{(l-1)}$, where $\delta_i^{(l)} = f'(tot)_i^{(l)} \sum_{p=1}^{n_l} \delta_p^{l+1} w_{pi}^{l+1}$.
- Again when f is taken as the sigmoid function, $\delta_i^{(l)}$ becomes expressed as: $\delta_i^{(l)} = o_i^{(l)}(1 - o_i^{(l)}) \sum_{p=1}^{n_l} \delta_p^{l+1} w_{pi}^{l+1}$.

Updating Rules: Off-Line Training

- The weight update rule:

$$\Delta w^{(l)} = -\eta \frac{\partial E_c}{\partial w^l}.$$

- All previous steps outlined for developing the on-line update rules are reproduced here with the exception that $E(k)$ becomes replaced with E_c.

- In both cases though, once the network weights have reached steady state values, the training algorithm is said to converge.

$\Delta w^L = \delta^L x^{L-1}$ This is the relationship between layers. These will propagate backwards untill it reaches the input.

$$\begin{aligned}
\Delta w_{ij}^L &= \eta \left[t_i - o_i^L \right] f'(tot)_i^L o_J^{L-1} \\
f(tot)_i^L &= \frac{\delta f(tot)^L}{\delta tot_i^L} \\
\delta_i^L &= \text{Error of the system}
\end{aligned}$$

Here f is the sigmoid function $f(x) = \frac{1}{1+e^{-x}}$. We like this activation function because it is infinitely differentiable $\frac{\delta f}{\delta x} = f(1-f)$. This is almost universally used.

$$\begin{aligned}
\Delta w_{ig}^L &= \eta \delta_i^L o_j^{L-1} \\
\delta_i^L &= (t_i - o_i^L)(o_i^L)(1 - o_i^L)
\end{aligned}$$

All of the above processes are online training. An alternative is batch/offline training where you put all of the input in at the same time. Usually online returns better results but there are places where offline training is helpful.

Now we look at propagating backwards: $l < L$

$$\begin{aligned}
\Delta w_{ij}^l &= \eta \delta^l o_j^{l-1} \\
\delta_i^l &= f'(tot_i)^l \sum_{p=1}^{nl} \delta_p^{l+1} w_{pi}^{l+1} \\
nl &= \text{number of hidden layers} \\
\delta_i^l &= o_i^l (1 - o_i^l) \sum_{p=1}^{nl} \delta_p^{l+1} w_{pi}^{l+1} \\
\Delta w_{ij}^l &= \eta o_i^l (1 - o_i^l) \sum_{p=1}^{nl} \delta_p^{l+1} w_{pi}^{l+1}
\end{aligned}$$

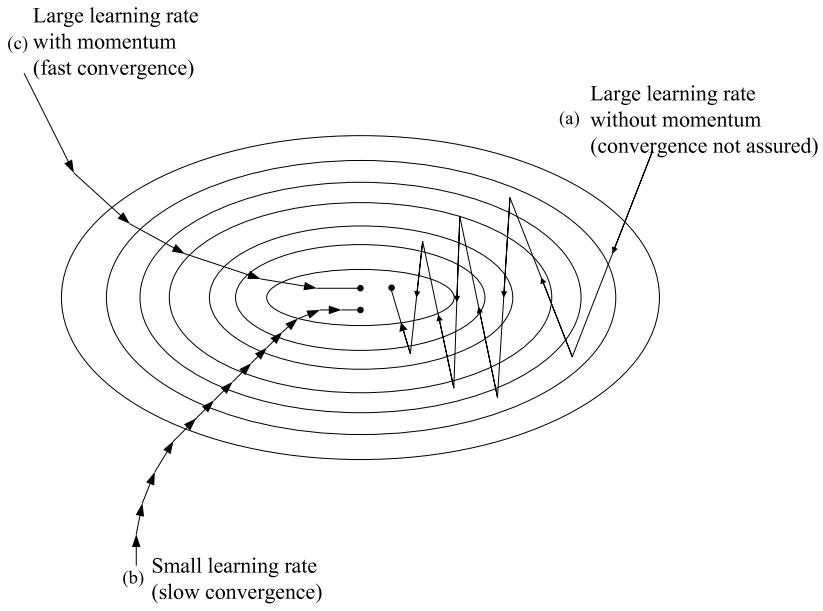
These equations areht emost equations used in back propagation.

Momentum

- The gradient descent requires by nature infinitesimal differentiation steps.
- For small values of the learning parameter η, this leads most often to a very slow convergence rate of the algorithm.
- Larger learning parameters have been known to lead to unwanted oscillations in the weight space.
- To avoid these issues, the concept of momentum has been introduced.

Momentum (cont.)

The modified weight update formulae including momentum term given as: $\Delta w^{(l)}(t+1) = -\eta \frac{\partial E_c(t)}{\partial w^l} + \gamma \Delta w^l(t)$.



Momentum helps us from oscillating like crazy, and encourages convergence.

So now we update our weights with the learning curve times the gradient descent plus some constant times the previous value of delta omega. This gives the system a bit of memory. It is now improving on earlier values of weight. The gradient descent is still small when compared to the previous value of omega. This means that delta omega is going to be very close to its previous value.

This method is very commonly used.

Bellow is an example, he chose not to go over it in class. In this example the bias is kept separate from the terms. This way it can be injected at certain areas. Remember that it is not always there. We see here that the output is scalar which is a bit new. Nothing really changes though.

Remember, you refresh the error between epochs. This is called online learning. Online learning has a smaller memory footprint than batch learning.

NOTE: this will likely be an example of a kind of question that could be on the exam. know it.

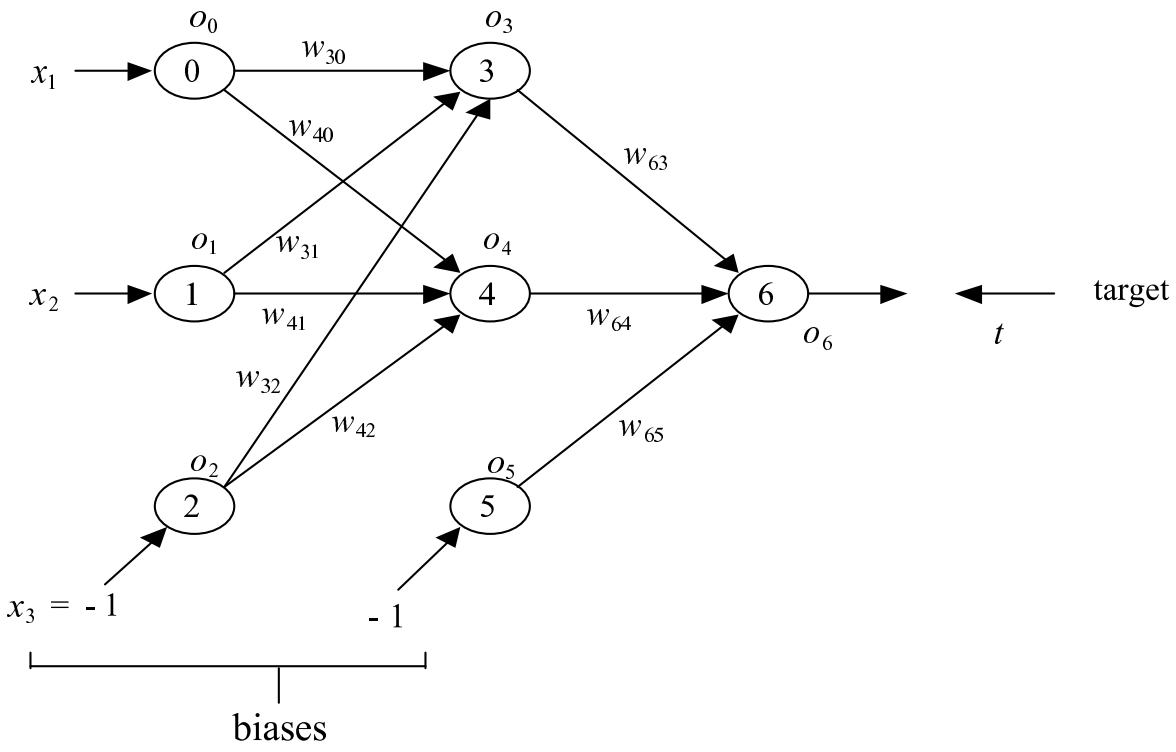
There are some typos:

- slide 29, lowest line should be $\delta_6 =$ the rest of the line

Example 1

- To illustrate this powerful algorithm, we apply it for the training of the following network shown in the next page.
- x : training patterns, and t : output data
  $x^{(1)} = (0.3, 0.4),\ \ t(1) = 0.88$
  $x^{(2)} = (0.1, 0.6),\ \ t(2) = 0.82$
  $x^{(3)} = (0.9, 0.4),\ \ t(3) = 0.57$
- Biases: -1
- Sigmoid activation function: $f(tot) = \frac{1}{1+e^{-\lambda tot}}$, using $\lambda = 1$, then $f'(tot) = f(tot)(1 - f(tot))$.

Example 1: Structure of the Network



Example 1: Training Loop (1)

- Step (1) Initialization
  - Initialize the weights to 0.2, set learning rate to $\eta = 0.2$; set maximum tolerable error to $E_{max} = 0.01$ (i.e. 1% error), set $E = 0$ and $k = 1$.
- Step (2) - Apply input pattern
  - Apply the 1^{st} input pattern to the input layer. $x^{(1)} = (0.3, 0.4)$, $t(1) = 0.88$, then,

    $o_0 = x_1 = 0.3$; $o_1 = x_2 = 0.4$; $o_2 = x_3 = -1$;

Example 1: Training Loop (1)

- Step (3) - Forward propagation
  - Propagate the signal forward through the network

$$o_3 = f(w_{30}o_0 + w_{31}o_1 + w_{32}o_2) = 0.485$$

$$o_4 = f(w_{40}o_0 + w_{41}o_1 + w_{42}o_2) = 0.485$$

$$o_5 = -1$$

$$o_6 = f(w_{63}o_3 + w_{64}o_4 + w_{65}o_5) = 0.4985$$

Example 1: Training Loop (1)

- Step (4) - Output error measure
  - Compute the error value E

$$E = \frac{1}{2}(t - o_6)^2 + E = 0.0728$$

  - Compute the error signal δ_6 of the output layer

$$\begin{aligned}\delta_6 &= f'(tot_6)(t - o_6) \\ &= o_6(1 - o_6)(t - o_6) \\ &= 0.0945\end{aligned}$$

Example 1: Training Loop (1)

Step (5) - Error back-propagation

- Third layer weight updates:

$$\Delta w_{63} = \eta \delta_6 o_3 = 0.0093 \qquad w_{63}^{new} = w_{63}^{old} + \Delta w_{63} = 0.2093$$

$$\Delta w_{64} = \eta \delta_6 o_4 = 0.0093 \qquad w_{64}^{new} = w_{64}^{old} + \Delta w_{64} = 0.2093$$

$$\Delta w_{65} = \eta \delta_6 o_5 = 0.0191 \qquad w_{65}^{new} = w_{65}^{old} + \Delta w_{65} = 0.1809$$

- Second layer error signals:

$$\delta_3 = f_3'(tot_3) \sum_{i=6}^{6} w_{i3}\delta_i = o_3(1 - o_3) w_{63}\delta_6 = 0.0048$$

$$\delta_4 = f_4'(tot_4) \sum_{i=6}^{6} w_{i4}\delta_i = o_4(1 - o_4) w_{64}\delta_6 = 0.0048$$

Example 1: Training Loop (1)

Step (5) - Error back-propagation (cont.)

- Second layer weight updates:

$\Delta w_{30} = \eta\delta_3 o_0 = 0.00028586 \quad w_{30}^{new} = w_{30}^{old} + \Delta w_{30} = 0.2003$

$\Delta w_{31} = \eta\delta_3 o_1 = 0.00038115 \quad w_{31}^{new} = w_{31}^{old} + \Delta w_{31} = 0.2004$

$\Delta w_{32} = \eta\delta_3 o_2 = -0.00095288 \quad w_{32}^{new} = w_{32}^{old} + \Delta w_{32} = 0.199$

$\Delta w_{40} = \eta\delta_4 o_0 = 0.00028586 \quad w_{40}^{new} = w_{40}^{old} + \Delta w_{40} = 0.2003$

$\Delta w_{41} = \eta\delta_4 o_1 = 0.00038115 \quad w_{41}^{new} = w_{41}^{old} + \Delta w_{41} = 0.2004$

$\Delta w_{42} = \eta\delta_4 o_2 = -0.00095288 \quad w_{42}^{new} = w_{42}^{old} + \Delta w_{42} = 0.199$

Example 1: Training Loop (2)

- Step (2) - Apply the 2^{nd} input pattern
  $x^{(2)} = (0.1, 0.6),\ t(2) = 0.82$, then,
  $o_0 = 0.1;\ o_1 = 0.6;\ o_2 = -1;$
- Step (3) - Forward propagation
  $o_3 = f(w_{30}o_0 + w_{31}o_1 + w_{32}o_2) = 0.4853$
  $o_4 = f(w_{40}o_0 + w_{41}o_1 + w_{42}o_2) = 0.4853$
  $o_5 = -1$
  $o_6 = f(w_{63}o_3 + w_{64}o_4 + w_{65}o_5) = 0.5055$
- Step (4) - Output error measure
  $E = \frac{1}{2}(t - o_6)^2 + E = 0.1222$
  $= o_6(1 - o_6)(t - o_6) = 0.0786$

Training Loop - Loop (2)

Step (5) - Error back-propagation

- Third layer weight updates:

$$\Delta w_{63} = \eta \delta_6 o_3 = 0.0076 \qquad w_{63}^{new} = w_{63}^{old} + \Delta w_{63} = 0.2169$$

$$\Delta w_{64} = \eta \delta_6 o_4 = 0.0076 \qquad w_{64}^{new} = w_{64}^{old} + \Delta w_{64} = 0.2169$$

$$\Delta w_{65} = \eta \delta_6 o_5 = 0.0157 \qquad w_{65}^{new} = w_{65}^{old} + \Delta w_{65} = 0.1652$$

- Second layer error signals:

$$\delta_3 = f_3'(tot_3) \sum_{i=6}^{6} w_{i3}\delta_i = o_3(1 - o_3) w_{63}\delta_6 = 0.0041$$

$$\delta_4 = f_4'(tot_4) \sum_{i=6}^{6} w_{i4}\delta_i = o_4(1 - o_4) w_{64}\delta_6 = 0.0041$$

Example 1: Training Loop (2)

Step (5) - Error back-propagation (cont.)

- Second layer weight updates:

$\Delta w_{30} = \eta\delta_3 o_0 = 0.000082169 \quad w_{30}^{new} = w_{30}^{old} + \Delta w_{30} = 0.2004$

$\Delta w_{31} = \eta\delta_3 o_1 = 0.00049302 \quad w_{31}^{new} = w_{31}^{old} + \Delta w_{31} = 0.2009$

$\Delta w_{32} = \eta\delta_3 o_2 = -0.00082169 \quad w_{32}^{new} = w_{32}^{old} + \Delta w_{32} = 0.1982$

$\Delta w_{40} = \eta\delta_4 o_0 = 0.000082169 \quad w_{40}^{new} = w_{40}^{old} + \Delta w_{40} = 0.2004$

$\Delta w_{41} = \eta\delta_4 o_1 = 0.00049302 \quad w_{41}^{new} = w_{41}^{old} + \Delta w_{41} = 0.2009$

$\Delta w_{42} = \eta\delta_4 o_2 = -0.00082169 \quad w_{42}^{new} = w_{42}^{old} + \Delta w_{42} = 0.1982$

Example 1: Training Loop (3)

- Step (2) - Apply the 2^{nd} input pattern
  $x^{(3)} = (0.9, 0.4),\ t(3) = 0.57$, then,
  $o_0 = 0.9;\ o_1 = 0.4;\ o_2 = -1;$
- Step (3) - Forward propagation

  $$o_3 = f(w_{30}o_0 + w_{31}o_1 + w_{32}o_2) = 0.5156$$

  $$o_4 = f(w_{40}o_0 + w_{41}o_1 + w_{42}o_2) = 0.5156$$

  $$o_5 = -1$$

  $$o_6 = f(w_{63}o_3 + w_{64}o_4 + w_{65}o_5) = 0.5146$$
- Step (4) - Output error measure

  $$E = \frac{1}{2}(t - o_6)^2 + E = 0.1237$$
  $$= o_6(1 - o_6)(t - o_6) = 0.0138$$

Example 1: Training Loop (3)

Step (5) - Error back-propagation

- Third layer weight updates:

$$\Delta w_{63} = \eta \delta_6 o_3 = 0.0014 \qquad w_{63}^{new} = w_{63}^{old} + \Delta w_{63} = 0.2183$$

$$\Delta w_{64} = \eta \delta_6 o_4 = 0.0014 \qquad w_{64}^{new} = w_{64}^{old} + \Delta w_{64} = 0.2183$$

$$\Delta w_{65} = \eta \delta_6 o_5 = -0.0028 \qquad w_{65}^{new} = w_{65}^{old} + \Delta w_{65} = 0.1624$$

- Second layer error signals:

$$\delta_3 = f_3'(tot_3) \sum_{i=6}^{6} w_{i3}\delta_i = o_3(1 - o_3) w_{63}\delta_6 = 0.00074948$$

$$\delta_4 = f_4'(tot_4) \sum_{i=6}^{6} w_{i4}\delta_i = o_4(1 - o_4) w_{64}\delta_6 = 0.00074948$$

Example 1: Training Loop (3)

Step (5) - Error back-propagation (cont.)

- Second layer weight updates:

$$\Delta w_{30} = \eta\delta_3 o_0 = 0.00013491 \quad w_{30}^{new} = w_{30}^{old} + \Delta w_{30} = 0.2005$$

$$\Delta w_{31} = \eta\delta_3 o_1 = 0.000059958 \quad w_{31}^{new} = w_{31}^{old} + \Delta w_{31} = 0.2009$$

$$\Delta w_{32} = \eta\delta_3 o_2 = -0.0001499 \quad w_{32}^{new} = w_{32}^{old} + \Delta w_{32} = 0.1981$$

$$\Delta w_{40} = \eta\delta_4 o_0 = 0.00013491 \quad w_{40}^{new} = w_{40}^{old} + \Delta w_{40} = 0.2005$$

$$\Delta w_{41} = \eta\delta_4 o_1 = 0.000059958 \quad w_{41}^{new} = w_{41}^{old} + \Delta w_{41} = 0.2009$$

$$\Delta w_{42} = \eta\delta_4 o_2 = -0.0001499 \quad w_{42}^{new} = w_{42}^{old} + \Delta w_{42} = 0.1981$$

Example 1: Final Decision

- Step (6) - One epoch looping

  The training patterns have been cycled one epoch.

- Step (7) - Total error checking

  $E = 0.1237$ and $E_{max} = 0.01$, which means that we have to continue with the next epoch by cycling the training data again.

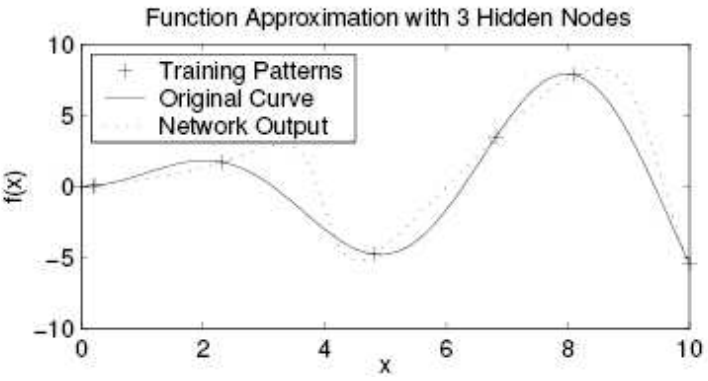
Now we have a system with three nodes

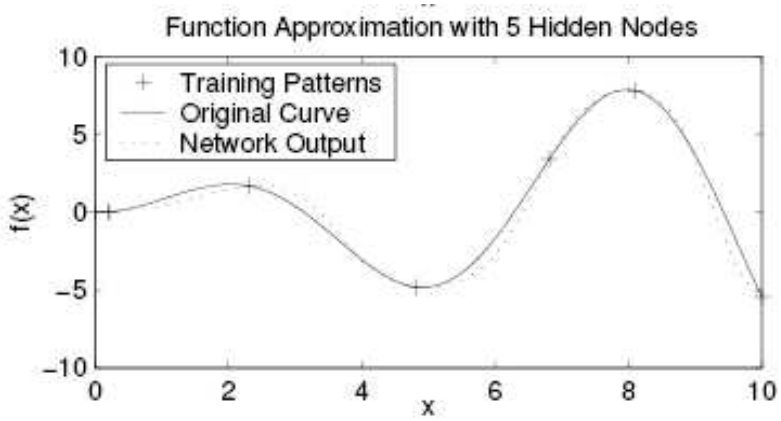
Example 2

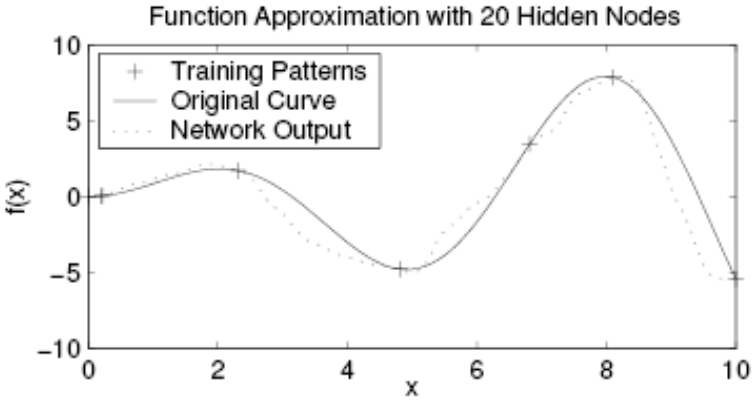
Effect of Hidden Nodes on Function Approximation

- Consider this function $f(x) = x\sin(x)$
- Six input/output samples were selected from the range $[0, 10]$ of the variable x
- The first run was made for a network with 3 hidden nodes
- Another run was made for a network with 5 and 20 nodes, respectively.

Example 2: Different Hidden Nodes







Example 2: Remarks

- A higher number of nodes is not always better. It may overtrain the network.
- This happens when the network starts to memorize the patterns instead of interpolating between them.
- A smaller number of nodes was not able to approximate faithfully the function given the nonlinearities induced by the network was not enough to interpolate well in between the samples.
- It seems here that this network (with five nodes) was able to interpolate quite well the nonlinear behavior of the curve.

Example 3

Effect of Training Patterns on Function Approximation

- Consider this function $f(x) = x\sin(x)$
- Assume a network with a fixed number of nodes (taken as five here), but with a variable number of training patterns
- The first run was made for a network with 3 three samples
- Another run was made for a network with 10 and 20 samples, respectively.

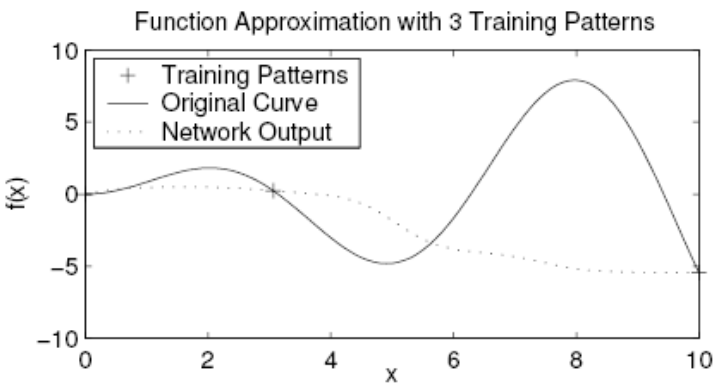
We can accidentally have too many nodes, we say that this causes the system to be **overtrained**. We do lots of cross validation, data changing, and reconfiguring of the system to see which form converges best. Basically you use a bunch of data to tune the system before you apply the testing data.

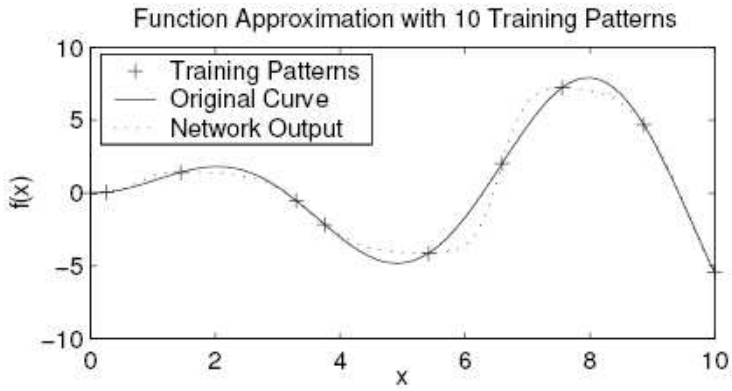
Example 3

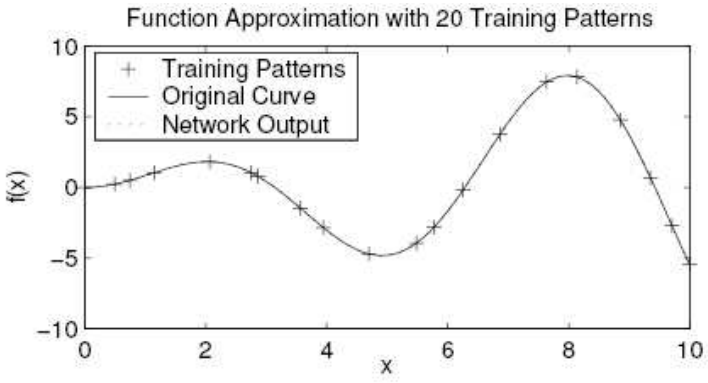
Effect of Training Patterns on Function Approximation

- Consider this function $f(x) = x\sin(x)$
- Assume a network with a fixed number of nodes (taken as five here), but with a variable number of training patterns
- The first run was made for a network with 3 three samples
- Another run was made for a network with 10 and 20 samples, respectively.

Example 3: Different Samples







Example 3: Remarks

- The first run with three samples was not able to provide a good mach with the original curve.
- This can be explained by the fact that the three patterns, in the case of a nonlinear function such as this, are not able to reproduce the relatively high nonlinearities of the function.
- A higher number of training points provided better results.
- The best result was obtained for the case of 20 training patterns. This is due to the fact that a network with five hidden nodes interpolates extremely well in between close training patterns.

The more varied your training data the better your system, but the more nodes you have might not improve things.

Applications of MLP

- Multilayer perceptrons are currently among the most used connectionist models.
- This stems from the relative ease for training and implementing, either in hardware or software forms.

Applications

- Signal processing
- Pattern recognition
- Financial market prediction
- Weather forecasting
- Signal compression

Limitations of MLP

- Among the well-known problems that may hinder the generalization or approximation capabilities of MLP is the one related to the convergence behavior of the connection weights during the learning stage.
- In fact, the gradient descent based algorithm used to update the network weights may never converge to the global minima.
- This is particularly true in the case of highly nonlinear behavior of the system being approximated by the network.

Limitations of MLP

- Many remedies have been proposed to tackle this issue either by retraining the network a number of times or by using optimization techniques such as those based on:
  - Genetic algorithms,
  - Simulated annealing.

If the system is not converging you have to reshuffle your data or the structure of your network.

NOTE: know how the different configurations and values can effect the system.

Below is another example he didnt go through

MLP NN: Case Study

Function Estimation (Regression)

MLP NN: Case Study

- Use a feedforward backpropagation neural network that contains a single hidden layer.
- Each of hidden nodes has an activation function of the logistic form.
- Investigate the outcome of the neural network for the following mapping.

$$f(x) = exp(-x^2), \quad x \in [0\ 2]$$

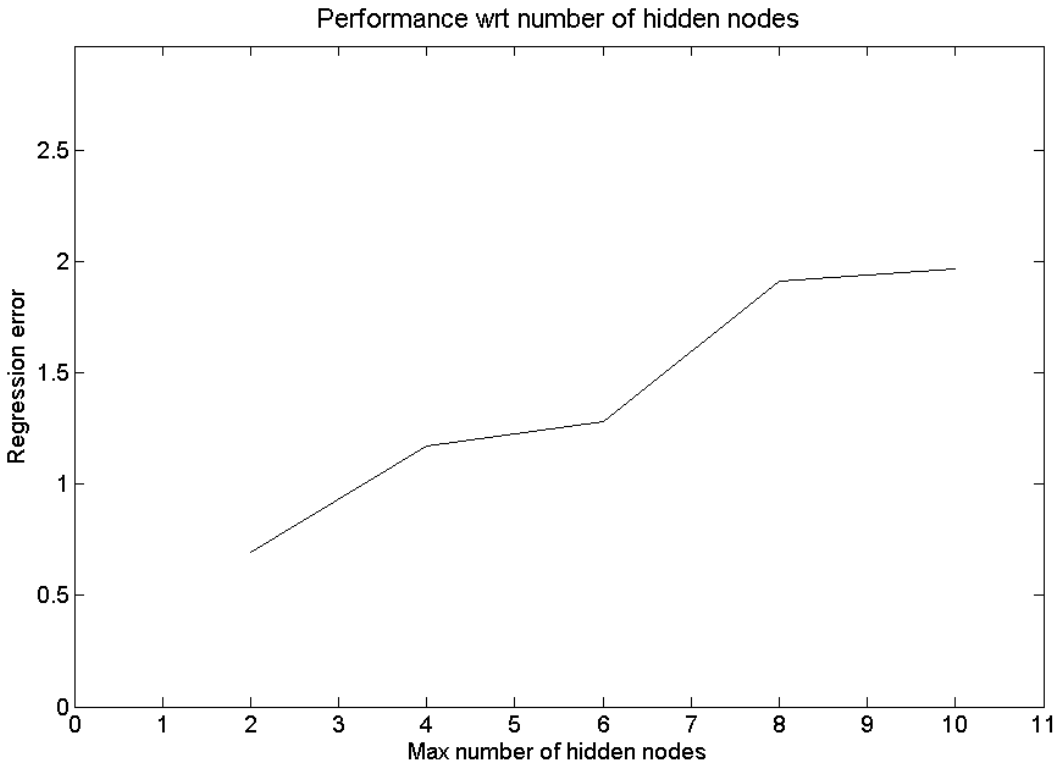
- Experiment with different number of training samples and hidden layer nodes

MLP NN: Case Study

Experiment 1: Vary Number of Hidden Nodes

- Uniformly pick six sample points from [0 2], use half of them for training and the rest for testing
- Evaluate regression performance increasing the number of hidden nodes
- Use sum of regression error (i.e. $\sum_{i \in \text{test samples}} (Output(i) - True\_output(i))$) as performance measure
- Repeat each test 20 times and compute average results, compensating for potential local minima

MLP NN: Case Study

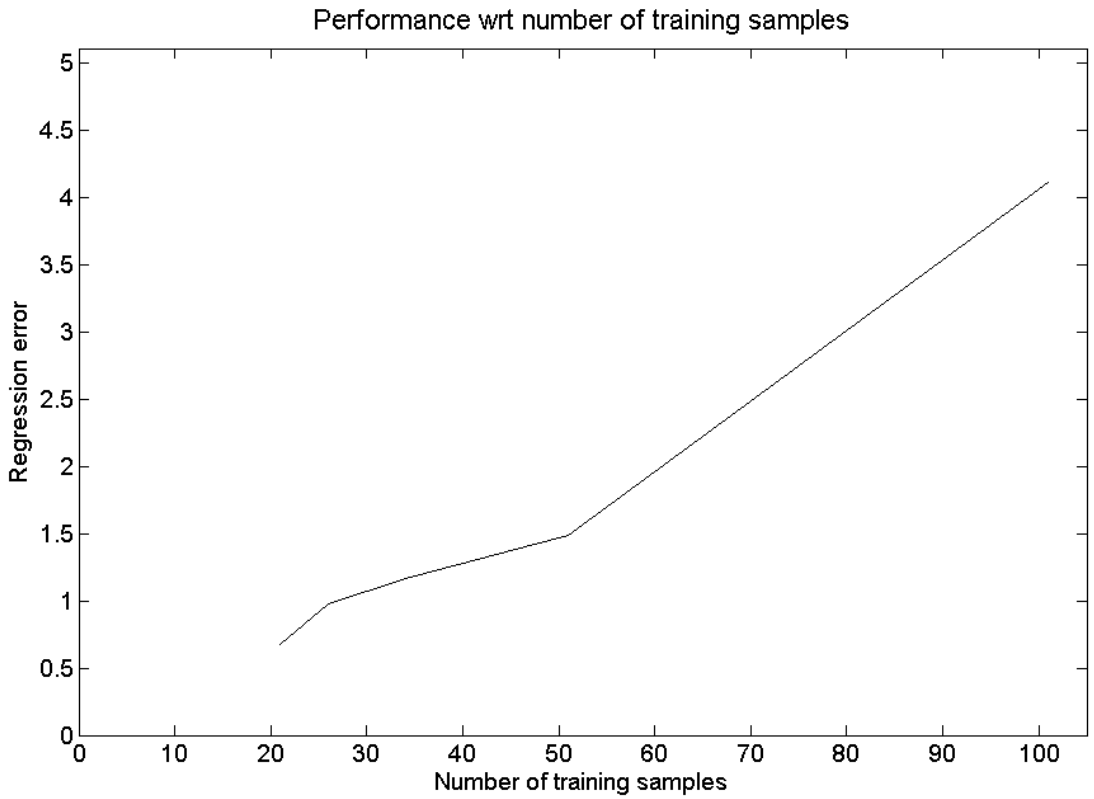


MLP NN: Case Study

Experiment 2: Vary Number of Training Samples

- Construct neural network using three hidden nodes
- Uniformly pick sample points from [0 2], increasing their number for each test
- Use half of sample data points for training and the rest for testing
- Use the same performance measure as experiment 1, i.e. sum of regression error
- Repeat each test 50 times and compute average results

MLP NN: Case Study



Now lets look at topology

Radial Basis Function Network

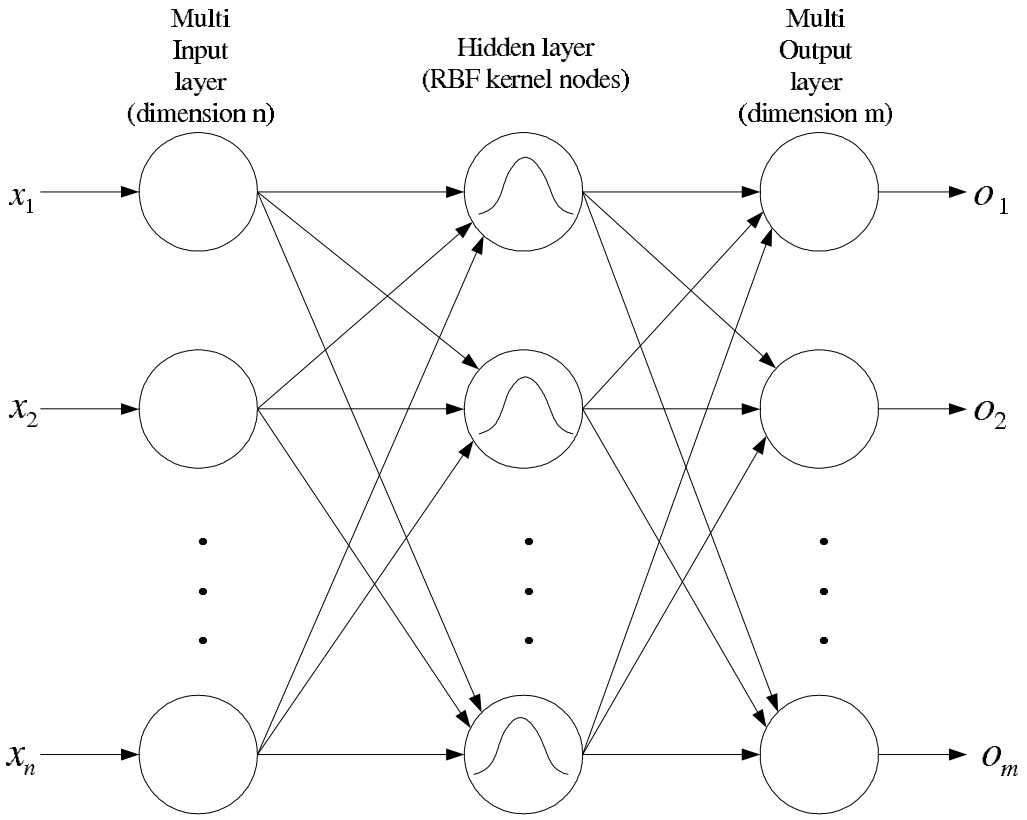
Topology

- Radial basis function network (RBFN) represent a special category of the **feedforward** neural networks architecture.
- Early researchers have developed this connectionist model for **mapping nonlinear behavior of static processes** and for **function approximation purposes**.
- The basic RBFN structure consists of **an input layer**, **a single hidden layer** with **radial activation function** and **an output layer**.

The typical activation for an MLP is tanh or sigmoid since they are infinitely differentiable.

We now consider closed/bounded activation functions that start high and get lower (a bit like converging). The best for the following system is the Gausian membership function.

Topology: Graphical Representation



Here we have central nodes that have maximum results and the edges less so. Here we only have weights between the hidden layer and the output layers. You can think of this as the weights between the input layer and the hidden layer as one.

We dont want a system to pass between all of the points, we want to it interpolate between the points. This is usually the result of **overfeeding**.

When we get our data set we should separate it into training, validation, and testing sets. Training and validation is used to create a system that will try to predict properly the results of the testing data. We can shuffle the training and validation data around to hone in the right amount of training for the system. The training is the one that tunes up the weights. The validation is used to validate the model that you produced. Do not go to the testing part if the validation is not good.

The behavior of the system depends on:

- the number of training data (you can control this)
  - larger is better
- architecture of the network, nodes and hidden layers (you can control this)
  - a medium value is needed, gotta tune this
- the behavior of the system or its linearity (you cannot control this)

this is probably going to be on the exam

The interpolator solution can create a very good approximation of your system even if it is nonlinear.

NOTE: I think his slides are borked again, so this might need syncing with them later

We can perfectly approximate a nonlinear function with an adequate number (that we need to find) if some other functions that we sum.

This is basically linear combination of bayes gaussian network. If you chose which of these functions you use your error can go to 0.

Topology (cont.)

- The network structure uses **nonlinear transformations** in its hidden layer (typical transfer functions for hidden functions are Gaussian curves).
- However, it uses **linear transformations** between the hidden and output layers.
- The rationale behind this is that input spaces, cast nonlinearly into high-dimensional domains, are more likely to be linearly separable than those cast into low-dimensional ones.

Here we are mapping a nonlinear function into higher dimensional domains. We want to extract from the system, all of the representatives of the overall system and cluster it into unique local systems.

Topology (cont.)

- Unlike most FF neural networks, the connection weights between the input layer and the neuron units of the hidden layer for an RBFN are all equal to **unity**.
- The nonlinear transformations at the hidden layer level have the main characteristics of being symmetrical.
- They also attain their maximum at the function center, and generate positive values that are rapidly decreasing with the distance from the center.

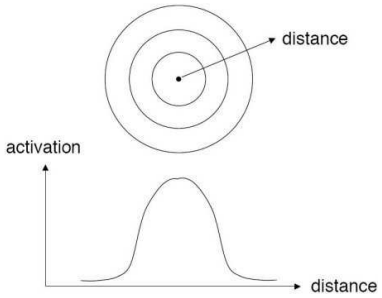
From the input layer to the middle layer there is the unity value (weights are 1) which is a unique character to radial systems.

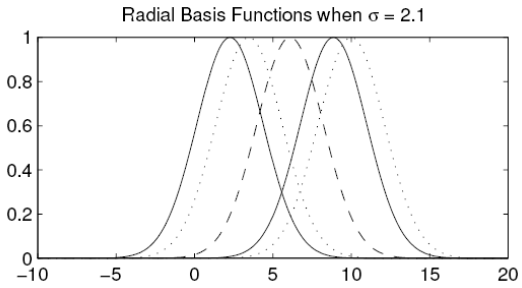
Topology (cont.)

- As such they produce radially activation signals that are bounded and localized.

Parameters of Each activation Function

- The center
- The width





Topology (cont.)

- For an optimal performance of the network, the hidden layer nodes should span the training data input space.
- Too sparse or too overlapping functions may cause the degradation of the network performance.

The generation of these functions required unsupervised learning. This is why it is called a hybrid system. One components is supervised and the other is through clustering.

We need the weights, the mean, and the standard deviations.

Weights are through supervised learning. The mean and standard deviation is through unsupervised learning.

Radial Function or Kernel Function

- In general the form taken by an RBF function is given as:

$$g_i(x) = r_i\left(\frac{\| x - v_i \|}{\sigma_i}\right)$$

  - where x is the input vector,
  - v_i is the vector denoting the center of the radial function g_i,
  - σ_i is width parameter.

Usually the center of the network has the same number of inputs as outputs.

So its the euclidean distance between the input vector and the center of the radial function, divided by some width parameter all of which fed into the activation function (I think r is the activation, needs to be confirmed).

Famous Radial Functions

- The Gaussian kernel function is the most widely used form of RBF given by:

$$g_i(x) = exp(\frac{- \parallel x - v_i \parallel^2}{2\sigma_i^2})$$

- The logistic function has also been used as a possible RBF candidate:

$$g_i(x) = \frac{1}{1 + exp(\frac{\|x - v_i\|^2}{\sigma_i^2})}$$

both of these variations of g are used as possible functions. The first is far more common, but some situations use the second with justification.

Output of an RBF Network

- A typical output of an RBF network having n units in the hidden layer and r output units is given by:

$$o_j(x) = \sum_{i=1}^{n} w_{ij} g_i(x), \; j = 1, \cdots, r.$$

  - where w_{ij} is the connection weight between the i-th receptive field unit and the j-th output,
  - g_i is the i-th receptive field unit (radial function).

We still consider ω as the weights between nodes.

The receptive field is another name for the radial function.

Learning Algorithm

Two-Stage Learning Strategy

- At first, an unsupervised clustering algorithm is used to extract the parameters of the radial basis functions, namely the width and the centers.
- This is followed by the computation of the weights of the connections between the output nodes and the kernel functions using a supervised least mean square algorithm.

First we use an unsupervised training to get the center vector (aka mean) and a scalar for the spread (aka standard deviation). Then we do supervised training to get the weights.

Learning Algorithm: Hybrid Approach

- The standard technique used to train an RBF network is the **hybrid approach**.

Hybrid Approach

- Step 1: Train the RBF layer to get the adaptation of centers and scaling parameters using the **unsupervised training**.
- Step 2: Adapt the weights of the output layer using **supervised training algorithm**.

Learning Algorithm: Step 1

- To determine the centers for the RBF networks, typically **unsupervised** training procedures of **clustering** are used:
  - K-means method,
  - "Maximum likelihood estimate" technique,
  - Self-organizing map method.
- This step is very important in the training of RBFN, as the accurate knowledge of v_i and σ_i has a major impact on the performance of the network.

Learning Algorithm: Step 2

- Once the centers and the widths of radial basis functions are obtained, the next stage of the training begins.
- To update the weights between the hidden layer and the output layer, the supervised learning based techniques such as are used:
  - Least-squares method,
  - Gradient method.
- Because the weights exist only between the hidden layer and the output layer, it is easy to compute the weight matrix for the RBFN.

For the interpolation problem the number of training data is equal to the hidden layer nodes. If you have 20 nodes you have 20 training data. Remember this, called **interpolation**.

Learning Algorithm: Step 2 (cont.)

- In the case where the RBFN is used for interpolation purposes, we can use the **inverse** or **pseudo-inverse method** to calculate the **weight matrix**.
- If we use Gaussian kernel as the radial basis functions and there are n input data, we have:

$$G = [\{g_{ij}\}],$$

where

$$g_{ij} = exp(\frac{- \parallel x_i - v_j \parallel^2}{2\sigma_j^2}), \ \ i, j = 1, \cdots, n$$

Learning Algorithm: Step 2 (cont.)

- Now we have:

$$D = GW$$

where D is the desired output of the training data.

- If G^{-1} exists, we get:

$$W = G^{-1}D$$

- In practice however, G may be ill-conditioned (close to singularity) or may even be a non-square matrix (if the number of radial basis functions is less than the number of training data) then W is expressed as:

$$W = G^{+}D$$

D is the desired output and G^{-1}, if it exists, gives us the weights.

If G^{-1} doesn't exists, it might be **illconditioned**. This happens if the determinant is very small resulting in large numbers. If this occurs it we use a non-square matrix.

This happens if the number of training data is not equal to the number of hidden nodes as well.

Learning Algorithm: Step 2 (cont.)

- We had:

$$W = G^{+}D,$$

  - where G^{+} denotes the pseudo-inverse matrix of G, which can be defined as

$$G^{+} = (G^{T}G)^{-1}G^{T}$$

- Once the weight matrix has been obtained, all elements of the RBFN are now determined and the network could operate on the task it has been designed for.

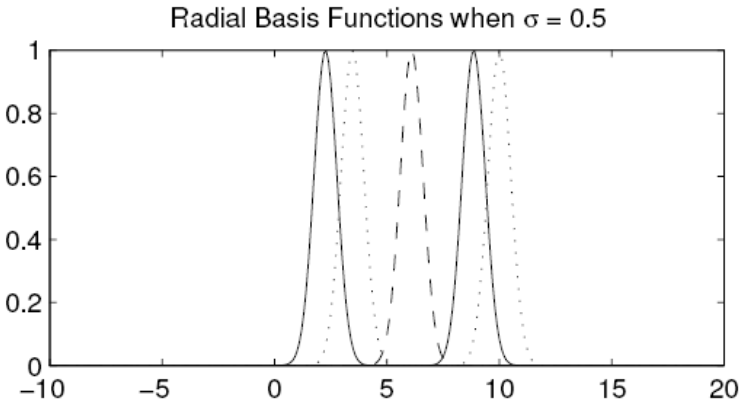
Example

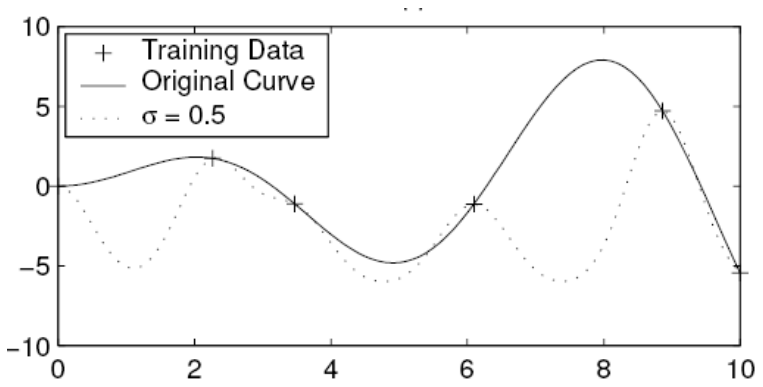
Approximation of Function $f(x)$ Using an RBFN

- We use here the same function as the one used in the MLP section, $f(x) = x\sin(x)$.
- The RBF network is composed here of five radial functions.
- Each radial function has its center at a training input data.
- Three width parameters are used here: 0.5, 2.1, and 8.5.
- The results of simulation show that the width of the function plays a major importance.

Suppose we have a function. We have 5 radial function, the center for each one at a training data. We have have an interpolation problem because we have the same number of inputs as training data.

Example: Function Approximation with Gaussian Kernels ($\sigma = 0.5$)

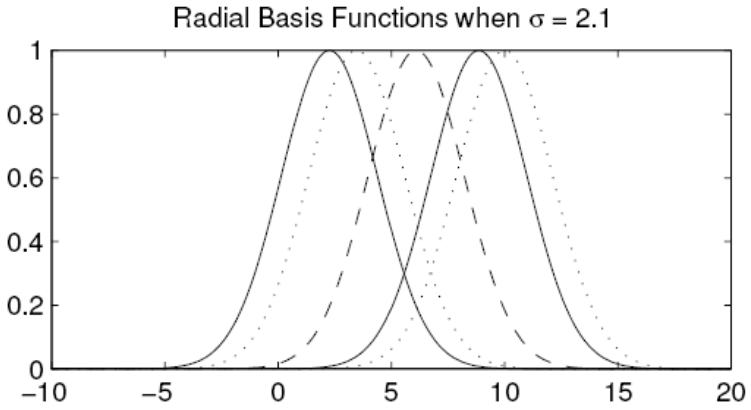


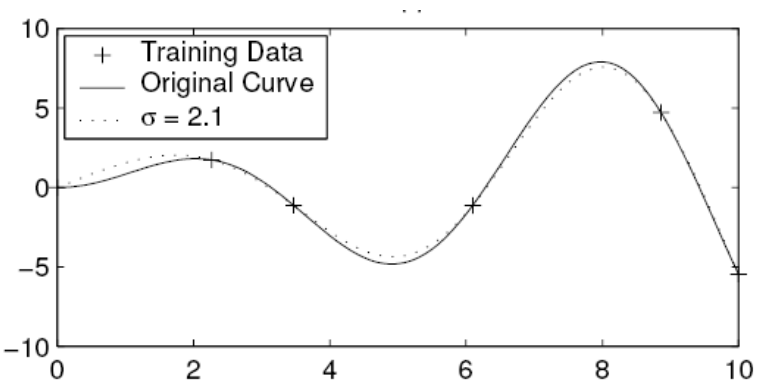


here are the funcitons we will use.

The one on the right uses a randomly chosen spread of 0.5. This is clearly incorrect.

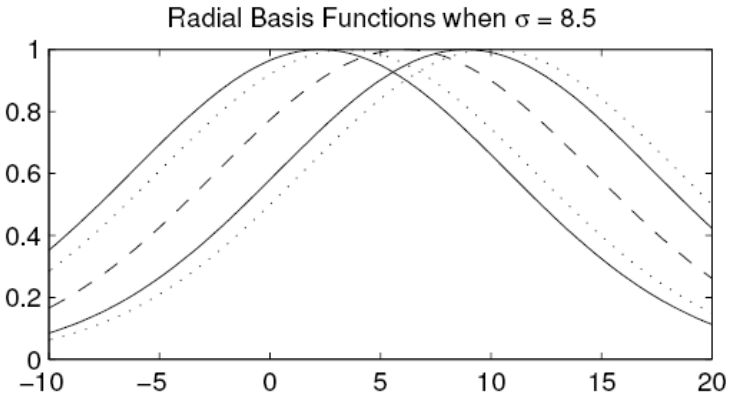
Example: Function Approximation with Gaussian Kernels ($\sigma = 2.1$)

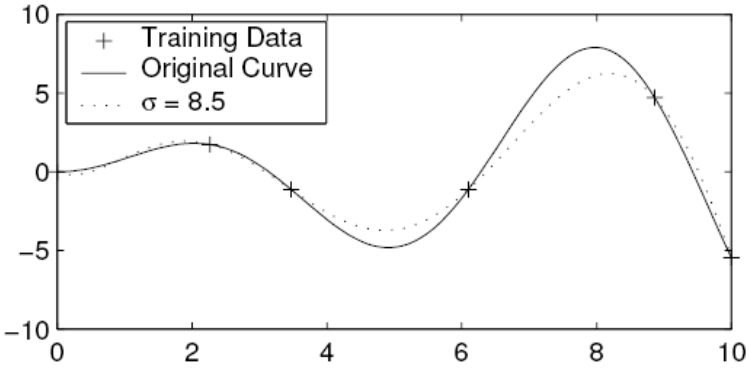




He we tried a different spread with much better results.

Example: Function Approximation with Gaussian Kernels ($\sigma = 8.5$)

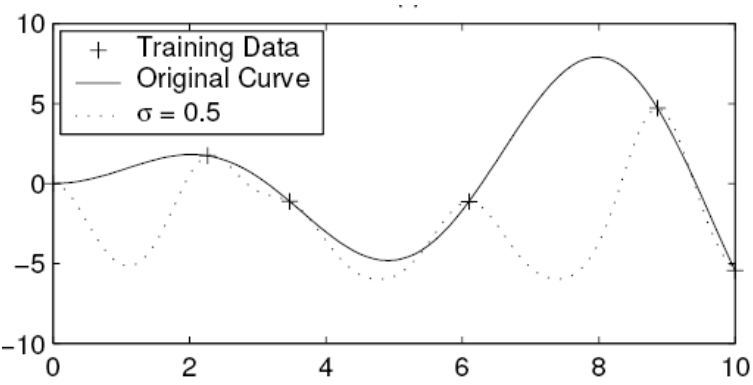


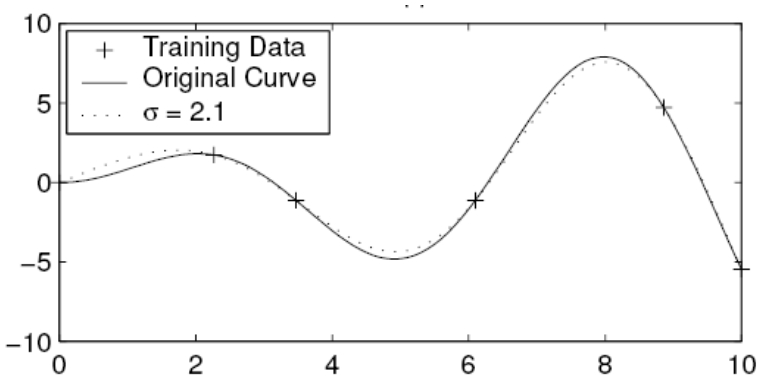


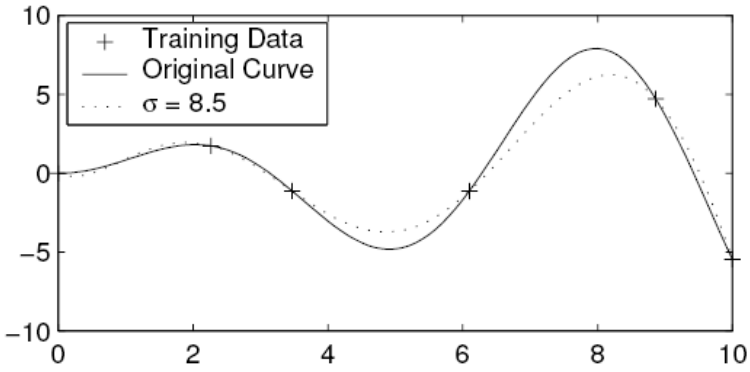
This one is overreaching on the spread.

It is important to extract this value from unsupervised training.

Example: Comparison







Example: Remarks

- A smaller width value 0.5 doesn't seem to provide for a good interpolation of the function in between sample data.
- A width value 2.1 provides a better result and the approximation by RBF is close to the original curve.
  - This particular width value seems to provide the network with the adequate interpolation property.
- A larger width value 8.5 seems to be inadequate for this particular case, given that a lot of information is being lost when the ranges of the radial functions are further away from the original range of the function.

Advantages/Disadvantages

- Unsupervised learning stage of an RBFN is not an easy task.
- RBF trains faster than a MLP.
- Another advantage that is claimed is that the hidden layer is easier to interpret than the hidden layer in an MLP.
- Although the RBF is quick to train, when training is finished and it is being used it is slower than a MLP, so where speed is a factor a MLP may be more appropriate.

RBF has been proven to be better than the multilayer perceptron.

Having a single hidden layer is much better and easier to work with. Its more transparent.

Which speed is faster depends on the complexity of the problem. For simple problems MLP is faster but RBF is better for larger systems.

Applications

- Known to have **universal approximation capabilities**, **good local structures** and **efficient training algorithms**, RBFN have been often used for nonlinear mapping of complex processes and for solving a wide range of **classification problems**.
- They have been used as well for control systems, audio and video signals processing, and pattern recognition.

Applications (cont.)

- They have also been recently used for **chaotic time series prediction**, with particular application to weather and power load forecasting.
- Generally, RBF networks have an undesirably high number of hidden nodes, but the dimension of the space can be reduced by careful planning of the network.