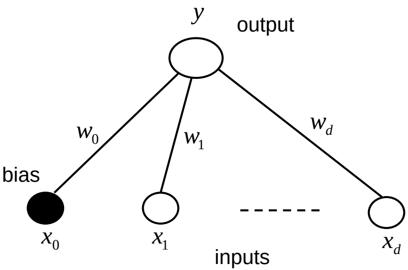
Artificial Neural Networks

Mingmin Chi

SCS Fudan University, Shanghai, China

Outline

Single-Layer Networks



$$y(\mathbf{x}) = f(\mathbf{w}^\top \Phi(\mathbf{x}))$$

The nonlinear activation function $f(\cdot)$ is given by a step function of the form

$$f(a) = \begin{cases} +1, & a \geq 0 \\ -1, & a < 0 \end{cases}$$

Single-Layer Networks (cont'd)

Advantages

- Easy to setup and train
- Outputs are weighted sum of inputs: interpretable representation

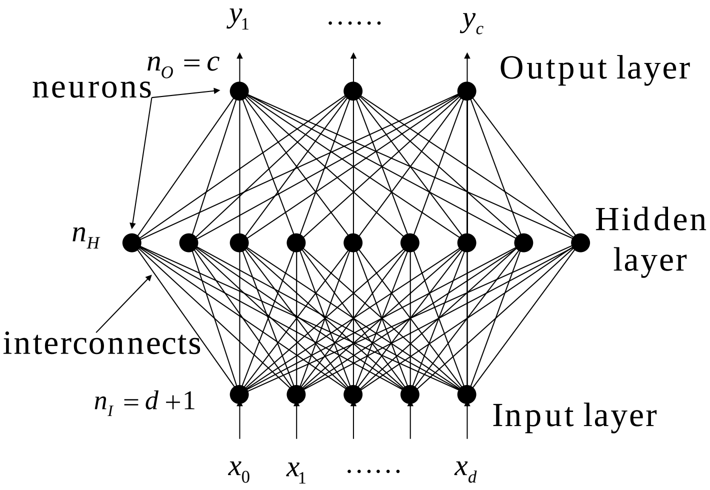
Limitations

- Can only represent a limited set of functions
- Decision boundaries must be hyperplane
- Can only perfectly separate linearly separable data

Relaxation to Other Neural Networks?

- Can we extend to three- or four-layer nets to overcome those drawbacks?
- How much multilayer networks can, at least in principle, provide the optimal solution to an arbitrary classification problem?
- They implement linear discriminants, nonlinear mapping

Topology of Two-layer Perceptron NNs



Following the same form of the linear models for classification

- Each d-dimensional input vector is presented to the input layer
- Each hidden unit emits an output to the output layer or the next hidden layer
  - Each hidden unit computes the weighted sum of its inputs to form its scalar *net activation* which we denote simply as *net*:

$$\text{net}_j = \sum_{i=1}^{d} x_i w_{ji} + w_{j0}$$

Following the same form of the linear models for classification

- Each d-dimensional input vector is presented to the input layer
- Each hidden unit emits an output to the output layer or the next hidden layer
  - Each hidden unit computes the weighted sum of its inputs to form its scalar *net activation* which we denote simply as *net*:

$$\text{net}_j = \sum_{i=1}^{d} x_i w_{ji} + w_{j0} = \sum_{i=0}^{d} x_i w_{ji}$$

Following the same form of the linear models for classification

- Each d-dimensional input vector is presented to the input layer
- Each hidden unit emits an output to the output layer or the next hidden layer
  - Each hidden unit computes the weighted sum of its inputs to form its scalar *net activation* which we denote simply as *net*:

$$\text{net}_j = \sum_{i=1}^{d} x_i w_{ji} + w_{j0} = \sum_{i=0}^{d} x_i w_{ji} = \mathbf{w}_j^\top \mathbf{x}$$

    w_{ji} denotes the input-to-hidden layer weights at the hidden unit j.
  - Each hidden unit emits an output that is a nonlinear function of its activation, $f(\text{net}_j)$, i.e.,

$$z_j = f(\text{net}_j)$$

- Each output unit emits the category
  - Each output unit similarly computes its net activation based on the hidden unit signals as

$$\text{net}_k = \sum_{j=1}^{n_H} z_j w_{kj} + w_{k0}$$

- Each output unit emits the category
  - Each output unit similarly computes its net activation based on the hidden unit signals as

$$\text{net}_k = \sum_{j=1}^{n_H} z_j w_{kj} + w_{k0} = \sum_{j=0}^{n_H} z_j w_{kj}$$

- Each output unit emits the category
  - Each output unit similarly computes its net activation based on the hidden unit signals as

$$\text{net}_k = \sum_{j=1}^{n_H} z_j w_{kj} + w_{k0} = \sum_{j=0}^{n_H} z_j w_{kj} = \mathbf{w}_k^\top \mathbf{z}$$

    at the output unit k.
  - Each output unit emits an output that is a nonlinear function of its activation, $f(\text{net}_k)$, i.e.,

$$y_k = f(\text{net}_k)$$

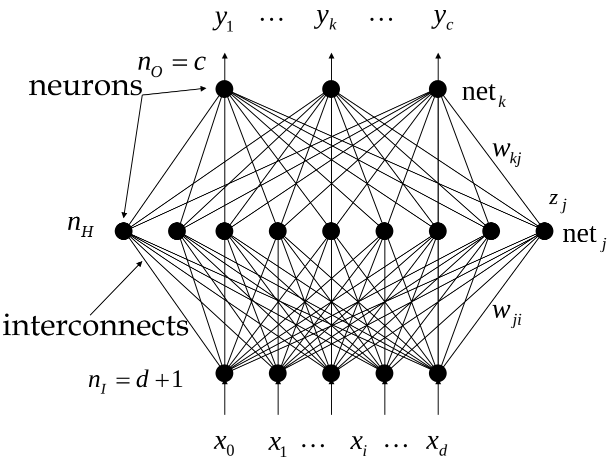
Feed-Forward Network Mappings

The mapping between inputs and outputs:

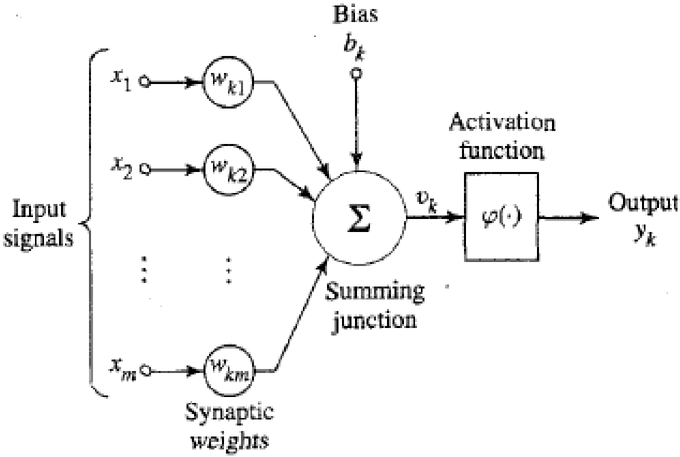
$$y_k = f\left(\underbrace{\sum_{j=0}^{n_H} f\left(\underbrace{\sum_{i=0}^{d} x_i w_{ji}}_{\text{net}_j}\right) w_{kj}}_{\text{net}_k}\right)$$

- Binary case, $y_k = \{+1, -1\}$
- Multi-category case, $y_k = f(\text{net}_k) = g_k(\mathbf{x})$

Topology of MLPNNs for Multiple Case Problem



Nonlinear Model of a Neuron



[Haykin, 2001]

Intuition

- The goal of the network training is to find an efficient technique for evaluating the gradient of an error function for a feed-forward neural network

Intuition

- The goal of the network training is to find an efficient technique for evaluating the gradient of an error function for a feed-forward neural network
- Could we calculate an effective error for each hidden unit, and thus derive a learning rule for the input-to-hidden weights?

In error backpropagation using a local message passing scheme information is sent alternately forwards and backwards through the network

Training Error

In general

- We consider the training error on a pattern to be the sum over output units of the squared difference between the desired output t_k and the actual output y_k:

$$E(\mathbf{w}) = \frac{1}{2}\sum_{k=1}^{c}(t_k - y_k)^2 = \frac{1}{2}||\mathbf{t} - \mathbf{y}||^2$$

Training Error

In general

- We consider the training error on a pattern to be the sum over output units of the squared difference between the desired output t_k and the actual output y_k:

$$E(\mathbf{w}) = \frac{1}{2}\sum_{k=1}^{c}(t_k - y_k)^2 = \frac{1}{2}||\mathbf{t} - \mathbf{y}||^2$$

- The back-propagation learning rule is based on gradient descent.

Gradient Descent

The back-propagation learning rule is based on gradient descent

- The weights are initialized with random values

Gradient Descent

The back-propagation learning rule is based on gradient descent

- The weights are initialized with random values
- The weights are changed in a direction that will reduce the error:

$$\Delta \mathbf{w} = -\eta \frac{\partial E}{\partial \mathbf{w}}$$

or in component form $\Delta \mathbf{w}_{pq} = -\eta \frac{\partial E}{\partial w_{pq}}$

Gradient Descent

The back-propagation learning rule is based on gradient descent

- The weights are initialized with random values
- The weights are changed in a direction that will reduce the error:

$$\Delta \mathbf{w} = -\eta \frac{\partial E}{\partial \mathbf{w}}$$

or in component form $\Delta \mathbf{w}_{pq} = -\eta \frac{\partial E}{\partial w_{pq}}$

- This iterative algorithm requires taking a weight vector at iteration τ and updating it as

$$\mathbf{w}^{(\tau+1)} = \mathbf{w}^{(\tau)} + \Delta \mathbf{w}^{(\tau)}$$

The Chain Rule for Differentiation

$$\mathrm{net}_k = \sum_{j=1}^{n_H} z_j w_{kj} + w_{k0} = \sum_{j=0}^{n_H} z_j w_{kj}$$

- Since the error is not explicitly dependent upon w_{kj}, we must use the chain rule for differentiation:

$$\frac{\partial E}{\partial w_{kj}}$$

The Chain Rule for Differentiation

$$\text{net}_k = \sum_{j=1}^{n_H} z_j w_{kj} + w_{k0} = \sum_{j=0}^{n_H} z_j w_{kj}$$

- Since the error is not explicitly dependent upon w_{kj}, we must use the chain rule for differentiation:

$$\frac{\partial E}{\partial w_{kj}} = \frac{\partial E}{\partial \text{net}_k} \frac{\partial \text{net}_k}{\partial w_{kj}}$$

The Chain Rule for Differentiation

$$\mathrm{net}_k = \sum_{j=1}^{n_H} z_j w_{kj} + w_{k0} = \sum_{j=0}^{n_H} z_j w_{kj}$$

- Since the error is not explicitly dependent upon w_{kj}, we must use the chain rule for differentiation:

$$\frac{\partial E}{\partial w_{kj}} = \frac{\partial E}{\partial \mathrm{net}_k} \frac{\partial \mathrm{net}_k}{\partial w_{kj}} = -\delta_k \frac{\partial \mathrm{net}_k}{\partial w_{kj}}$$

The Chain Rule for Differentiation

$$\text{net}_k = \sum_{j=1}^{n_H} z_j w_{kj} + w_{k0} = \sum_{j=0}^{n_H} z_j w_{kj}$$

- Since the error is not explicitly dependent upon w_{kj}, we must use the chain rule for differentiation:

$$\frac{\partial E}{\partial w_{kj}} = \frac{\partial E}{\partial \text{net}_k} \frac{\partial \text{net}_k}{\partial w_{kj}} = -\delta_k \frac{\partial \text{net}_k}{\partial w_{kj}}$$

- We define δ_k as the *sensitivity* of the unit k:

$$\delta_k = -\frac{\partial E}{\partial \text{net}_k}$$

The Chain Rule for Differentiation (cont'd)

- Assuming that the activation function $f(\cdot)$ is differentiable, δ_k can be simplified as follows:

$$\delta_k = -\frac{\partial E}{\partial \mathrm{net}_k}$$

The Chain Rule for Differentiation (cont'd)

- Assuming that the activation function $f(\cdot)$ is differentiable, δ_k can be simplified as follows:

$$\delta_k = -\frac{\partial E}{\partial \mathrm{net}_k} = -\frac{\partial E}{\partial y_k}\frac{\partial y_k}{\partial \mathrm{net}_k}$$

The Chain Rule for Differentiation (cont'd)

- Assuming that the activation function $f(\cdot)$ is differentiable, δ_k can be simplified as follows:

$$\delta_k = -\frac{\partial E}{\partial \text{net}_k} = -\frac{\partial E}{\partial y_k}\frac{\partial y_k}{\partial \text{net}_k} = (t_k - y_k)f'(\text{net}_k)$$

The Chain Rule for Differentiation (cont'd)

- Assuming that the activation function $f(\cdot)$ is differentiable, δ_k can be simplified as follows:

$$\delta_k = -\frac{\partial E}{\partial \text{net}_k} = -\frac{\partial E}{\partial y_k}\frac{\partial y_k}{\partial \text{net}_k} = (t_k - y_k)f'(\text{net}_k)$$

- As $\text{net}_k = \sum_{j=0}^{n_H} w_{kj}z_j$, we have $\frac{\partial \text{net}_k}{\partial w_{kj}} = z_j$
- Learning rule for the hidden-to-output weights:

$$\Delta w_{kj} = \eta \delta_k z_j$$

The Chain Rule for Differentiation (cont'd)

- Assuming that the activation function $f(\cdot)$ is differentiable, δ_k can be simplified as follows:

$$\delta_k = -\frac{\partial E}{\partial \text{net}_k} = -\frac{\partial E}{\partial y_k}\frac{\partial y_k}{\partial \text{net}_k} = (t_k - y_k)f'(\text{net}_k)$$

- As $\text{net}_k = \sum_{j=0}^{n_H} w_{kj} z_j$, we have $\frac{\partial \text{net}_k}{\partial w_{kj}} = z_j$
- Learning rule for the hidden-to-output weights:

$$\Delta w_{kj} = \eta \delta_k z_j = \eta \underbrace{(t_k - y_k)f'(\text{net}_k)}_{\delta_k} z_j$$

- In the case of the output unit is linear, i.e., $f(\text{net}_k) = \text{net}_k$ and $f'(\text{net}_k) = 1$, then the above equation is simply the LMS rule

The Chain Rule for Differentiation

The learning rule for the input-to-hidden units is more subtle, and it is the crux of the solution to the credit assignment problem

The Chain Rule for Differentiation

The learning rule for the input-to-hidden units is more subtle, and it is the crux of the solution to the credit assignment problem

$$\text{net}_j = \sum_{i=1}^{d} x_i w_{ji} + w_{j0} = \sum_{i=0}^{d} x_i w_{ji}$$

With the chain rule, we calculate

$$\frac{\partial E}{\partial w_{ji}}$$

The Chain Rule for Differentiation

The learning rule for the input-to-hidden units is more subtle, and it is the crux of the solution to the credit assignment problem

$$\text{net}_j = \sum_{i=1}^{d} x_i w_{ji} + w_{j0} = \sum_{i=0}^{d} x_i w_{ji}$$

With the chain rule, we calculate

$$\frac{\partial E}{\partial w_{ji}} = \frac{\partial E}{\partial z_j} \frac{\partial z_j}{\partial \text{net}_j} \frac{\partial \text{net}_j}{\partial w_{ji}}$$

The Chain Rule for Differentiation(cont'd)

$$\mathrm{net}_j = \sum_{i=0}^{d} x_i w_{ji}, \quad \frac{\partial E}{\partial w_{ji}} = \frac{\partial E}{\partial z_j} \frac{\partial z_j}{\partial \mathrm{net}_j} \frac{\partial \mathrm{net}_j}{\partial w_{ji}}$$

- The last term on the right-hand side involves all of the weights w_{ji}:

$$\frac{\partial \mathrm{net}_j}{\partial w_{ji}} = x_i$$

- The second term on the right-hand side:

$$\frac{\partial z_j}{\partial \mathrm{net}_j} = f'(\mathrm{net}_j)$$

The Chain Rule for Differentiation(cont'd)

- The first term on the right-hand side involves all of the weights w_{kj}:

$$\begin{aligned}
\frac{\partial E}{\partial z_j} &= \frac{\partial}{\partial z_j}\left[\frac{1}{2}\sum_{k=1}^{c}(t_k - y_k)^2\right] \\
&= -\sum_{k=1}^{c}(t_k - y_k)\frac{\partial y_k}{\partial z_j} \\
&= -\sum_{k=1}^{c}(t_k - y_k)\frac{\partial y_k}{\partial \text{net}_k}\frac{\partial \text{net}_k}{\partial z_j} \\
&= -\sum_{k=1}^{c}\underbrace{(t_k - y_k)f'(\text{net}_k)}_{\delta_k} w_{kj}
\end{aligned}$$

The Chain Rule for Differentiation(cont'd)

- In analogy with the case for hidden-to-output, we can also define the sensitivity for a hidden unit as

The Chain Rule for Differentiation(cont'd)

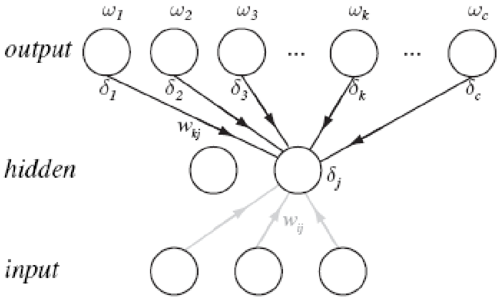
- In analogy with the case for hidden-to-output, we can also define the sensitivity for a hidden unit as

$$\delta_j \equiv f'(\text{net}_j) \sum_{k=1}^{c} w_{kj}\delta_k$$

  This is the core of the solution to the credit assignment problem
- The learning rule for the input-to-hidden weights is

$$\Delta w_{ji} = \eta \delta_j x_i = \eta \underbrace{f'(\text{net}_j) \left[\sum_{k=1}^{c} w_{kj}\delta_k \right]}_{\delta_j} x_i$$

BP Algorithm



$$\Delta w_{kj} = \eta \underbrace{(t_k - y_k) f'(\text{net}_k)}_{\delta_k} z_j$$

$$\Delta w_{ji} = \eta \underbrace{f'(\text{net}_j) \left[\sum_{k=1}^{c} w_{kj} \delta_k \right]}_{\delta_j} x_i$$

BP Algorithm

begin initialize network topology (# hidden units), $\mathbf{w}$, criterion $\theta, \eta, r \leftarrow 0$
    do $r \leftarrow r + 1$ (increment epoch)
        $m \leftarrow 0;\ \Delta w_{ij} \leftarrow 0;\ \Delta w_{jk} \leftarrow 0$
        do $m \leftarrow m + 1$
            $\mathbf{x}^m \leftarrow$ select pattern
            $\Delta w_{ij} \leftarrow \Delta w_{ij} + \eta \delta_j x_i;\quad \Delta w_{jk} \leftarrow \Delta w_{jk} + \eta \delta_k y_j$
        until $m = n$
        $w_{ij} \leftarrow w_{ij} + \Delta w_{ij};\quad w_{jk} \leftarrow w_{jk} + \Delta w_{jk}$
    until $\nabla J(\mathbf{w}) < \theta$
return $\mathbf{w}$
end

More General Cases

- Input units include a bias unit

More General Cases

- Input units include a bias unit
- Input units are connected directly to output units as well as hidden units

More General Cases

- Input units include a bias unit
- Input units are connected directly to output units as well as hidden units
- Three are more than two-layer of units

More General Cases

- Input units include a bias unit
- Input units are connected directly to output units as well as hidden units
- Three are more than two-layer of units
- There are different nonlinearities for different layers

More General Cases

- Input units include a bias unit
- Input units are connected directly to output units as well as hidden units
- Three are more than two-layer of units
- There are different nonlinearities for different layers
- Each unit has its own nonlinearity

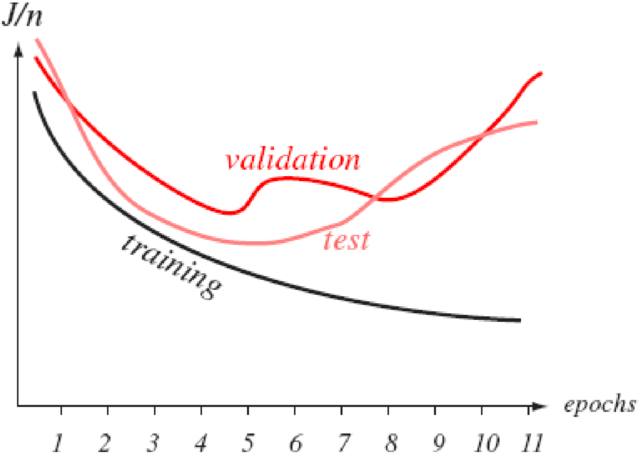
More General Cases

- Input units include a bias unit
- Input units are connected directly to output units as well as hidden units
- Three are more than two-layer of units
- There are different nonlinearities for different layers
- Each unit has its own nonlinearity
- Each unit has a different learning rate

Generalization

- Regularization - the complexity of the networks

Learning curve



Practical Issues

- Activation function: nonlinear, saturation, monotonicity and continuity and smoothness
- Scaling input
- Number of hidden units n_H, roughly, $n/10$
- Initializing weights
- Learning rate, $\eta = 0.1$
- Weight decay
- Number of hidden layers
- Criterion (or objective) function

- MLPNN performs a non-linear mapping from the input space to the output space
- The other major class of neural network model can be considered in which hidden units provides a set of “foundations" constitute an arbitrary “basis" for the input patterns when they are expanded to hidden space [haykin, 2001]
- Eps. the activation of a hidden unit is determined by the distance between the input vector and a prototype vector, known as Radial Basis Function neural networks, RBF-NN with only one hidden layer
- The training for RBF networks can be substantially faster than the method used to train multi-layer perceptron networks, i.e., the error-backpropagation algorithm based on the stochastic gradient descent

Cover's Theorem on the Separability of Patterns

[Cover 1965]

A complex pattern-classification problem cast in a high-dimensional nonlinear is more likely to be linearly separable than in a low-dimensional space

two ingredients

- Nonlinear formulation of the hidden functions defined by $\phi_i(\mathbf{x})$, where $\mathbf{x}$ is the input vector
- High dimensionality of the hidden space compared to the input space

In some cases, the use of nonlinear mapping is sufficient to produce the linear separability without having to increase the dimensionality of the hidden space

Exact Interpolation

- Origins in techniques for performing exact interpolation of a set of data points in a multi-dimensional space
- The exact interpolation problem requires every input vector to be mapped exactly onto the corresponding target vector, e.g., if the data set $(\mathbf{x}_i, t_i), \mathbf{x}_i \in \mathbb{R}^d, t^i \in \mathbb{R}, , i = 1, \cdots, n$, the goal is to find a function $h(\mathbf{x})$ such that $h(\mathbf{x}_i) = t_i$

Exact Interpolation

- Origins in techniques for performing exact interpolation of a set of data points in a multi-dimensional space
- The exact interpolation problem requires every input vector to be mapped exactly onto the corresponding target vector, e.g., if the data set $(\mathbf{x}_i, t_i), \mathbf{x}_i \in \mathbb{R}^d, t^i \in \mathbb{R}, , i = 1, \cdots, n$, the goal is to find a function $h(\mathbf{x})$ such that $h(\mathbf{x}_i) = t_i$
- The radial basis function approach (Powell, 1987) introduces a set of n basis functions, one for each point, which take the form $\phi(||\mathbf{x} - \mathbf{x}_i||)$
- The ith such function depends on the distance $||\mathbf{x} - \mathbf{x}_i||$
- The output of the mapping:

$$h(\mathbf{x}) = \sum_{i=1}^{n} w_i \phi(||\mathbf{x} - \mathbf{x}_i||)$$

Exact Interpolation (cont)

- The interpolation conditions can then be written in matrix form as

$$\Phi \mathbf{w} = \mathbf{t}$$

- Provided the inverse matrix Φ^{-1} exists we can solve the above function to give

$$\mathbf{w} = \Phi^{-1}\mathbf{t}$$

  (where every pattern should be different)

- The most commonly used the function is Gaussian:

$$\phi(\mathbf{x}) = \exp\left(-\frac{\mathbf{x}^2}{2\sigma^2}\right)$$

RBF networks

- Limitation of the exact interpolation
- By introducing a number of modifications to the exact interpolation procedure we obtain the radial basis function neural network model (Broomhead and Lowe, 1988; Moody and Darken, 1989)

RBF networks

- Limitation of the exact interpolation
- By introducing a number of modifications to the exact interpolation procedure we obtain the radial basis function neural network model (Broomhead and Lowe, 1988; Moody and Darken, 1989)
- This provides a smooth interpolating function in which the number of basis functions is determined by the complexity of the mapping to be represented rather than by the size of the data set. The modification which are required are as follows:

RBF networks

- Limitation of the exact interpolation
- By introducing a number of modifications to the exact interpolation procedure we obtain the radial basis function neural network model (Broomhead and Lowe, 1988; Moody and Darken, 1989)
- This provides a smooth interpolating function in which the number of basis functions is determined by the complexity of the mapping to be represented rather than by the size of the data set. The modification which are required are as follows:
  1. the number M of basis functions need not equal to the number n of data points, typically $M \ll n$

RBF networks

- Limitation of the exact interpolation
- By introducing a number of modifications to the exact interpolation procedure we obtain the radial basis function neural network model (Broomhead and Lowe, 1988; Moody and Darken, 1989)
- This provides a smooth interpolating function in which the number of basis functions is determined by the complexity of the mapping to be represented rather than by the size of the data set. The modification which are required are as follows:
    1. the number M of basis functions need not equal to the number n of data points, typically $M \ll n$
    2. The centers of the basis functions

RBF networks

- Limitation of the exact interpolation
- By introducing a number of modifications to the exact interpolation procedure we obtain the radial basis function neural network model (Broomhead and Lowe, 1988; Moody and Darken, 1989)
- This provides a smooth interpolating function in which the number of basis functions is determined by the complexity of the mapping to be represented rather than by the size of the data set. The modification which are required are as follows:
  1. the number M of basis functions need not equal to the number n of data points, typically $M \ll n$
  2. The centers of the basis functions
  3. The width σ_j of the basis functions

RBF networks

- Limitation of the exact interpolation
- By introducing a number of modifications to the exact interpolation procedure we obtain the radial basis function neural network model (Broomhead and Lowe, 1988; Moody and Darken, 1989)
- This provides a smooth interpolating function in which the number of basis functions is determined by the complexity of the mapping to be represented rather than by the size of the data set. The modification which are required are as follows:
  1. the number M of basis functions need not equal to the number n of data points, typically $M \ll n$
  2. The centers of the basis functions
  3. The width σ_j of the basis functions
  4. Bias parameters are included in the linear sum

RBF Networks

- When all the above change are made to the exact interpolation, we arrive at the following form for the RBF-NNs:

$$y_k(\mathbf{x}) = \sum_{j=1}^{M} w_{kj}\phi_j(\mathbf{x}) + w_{k0}$$

RBF Networks

- When all the above change are made to the exact interpolation, we arrive at the following form for the RBF-NNs:

$$y_k(\mathbf{x}) = \sum_{j=1}^{M} w_{kj}\phi_j(\mathbf{x}) + w_{k0}$$

- For the case of Gaussian basis functions we have

$$\phi_j(\mathbf{x}) = \exp\left(-\frac{||\mathbf{x} - \boldsymbol{\mu}_j||^2}{2\sigma_j^2}\right)$$

RBF Network Training

Two stage training procedure

1. The input dataset **X** alone is used to determine the parameters of the basis functions (e.g., μ_j and σ_j for the spherical Gaussian basis functions considered above)

RBF Network Training

Two stage training procedure

1. The input dataset **X** alone is used to determine the parameters of the basis functions (e.g., $\boldsymbol{\mu}_j$ and σ_j for the spherical Gaussian basis functions considered above)
2. The second stage training
   - We begin by absorbing the bias as

$$y_k(\mathbf{x}) = \sum_{j=0}^{M} w_{kj}\phi_j(\mathbf{x})$$

   - This can be written in matrix notation as

$$\mathbf{y} = \mathbf{W}\boldsymbol{\Phi}$$

The Second-Stage RBF Network Training

- We can optimize the weights by minimization of a suitable error function, conveniently, to consider a sum-of-squares error function given by

$$E = \frac{1}{2}\sum_i \sum_k \{(y_k(\mathbf{x}_i) - t_k^i)^2\}$$

The Second-Stage RBF Network Training

- We can optimize the weights by minimization of a suitable error function, conveniently, to consider a sum-of-squares error function given by

$$E = \frac{1}{2} \sum_i \sum_k \{(y_k(\mathbf{x}_i) - t_k^i)^2\}$$

- We can obtain the normal equation for the least squares problem:

$$(\Phi^\top \Phi)\mathbf{W}^\top = \Phi^\top \mathbf{T}$$
$$\Rightarrow \mathbf{W}^\top = \Phi^\dagger \mathbf{T}$$

  where $\Phi^\dagger$ denotes the pseudo-inverse of Φ.

Comparison between RBF and MLP

- RBF- single hidden layer
- MLP share the same model for each unit of hidden layer; RBF-different
- RBF- hidden is nonlinear but output is linear but in the MLP both are nonlinear
- The argument of activation function of RBF is Euclidean distance between the center unit and the input pattern; in MLP, activation function is inner product between the input vector and the weight vector of each unit
- MLP- global approximation to nonlinear input-output mapping, but RBF- local approximation to nonlinear input-output mapping