Neural Network Lectures

ME 60033 IMS 2014-15

Radial Basis Functions

Dr C S Kumar, Robotics and Intelligent Systems Lab

Introduction to Radial Basis Functions

The idea of ***Radial Basis Function (RBF) Networks*** derives from the theory of function approximation. We have already seen how Multi-Layer Perceptron (MLP) networks with a hidden layer of sigmoidal units can learn to approximate functions. RBF Networks take a slightly different approach. Their main features are:

1. They are two-layer feed-forward networks.
2. The hidden nodes implement a set of radial basis functions (e.g. Gaussian functions).
3. The output nodes implement linear summation functions as in an MLP.
4. The network training is divided into two stages: first the weights from the input to hidden layer are determined, and then the weights from the hidden to output layer.
5. The training/learning is very fast.
6. The networks are very good at interpolation.

Exact Interpolation

The ***exact interpolation*** of a set of N data points in a multi-dimensional space requires every one of the D dimensional input vectors $\mathbf{x}^p = \{x_i^p : i = 1, \ldots, D\}$ to be mapped onto the corresponding target output t^p. The goal is to find a function $f(\mathbf{x})$ such that

$$f(\mathbf{x}^p) = t^p \qquad \forall\, p = 1, \ldots, N$$

The radial basis function approach introduces a set of N ***basis functions***, one for each data point, which take the form $\phi(\|\mathbf{x} - \mathbf{x}^p\|)$ where $\phi(\cdot)$ is some non-linear function whose form will be discussed shortly. Thus the pth such function depends on the distance $\|\mathbf{x} - \mathbf{x}^p\|$, usually taken to be Euclidean, between $\mathbf{x}$ and $\mathbf{x}^p$. The output of the mapping is then taken to be a linear combination of the basis functions, i.e.

$$f(\mathbf{x}) = \sum_{p=1}^{N} w_p \phi(\|\mathbf{x} - \mathbf{x}^p\|)$$

The idea is to find the "weights" w_p such that the function goes through the data points.

Determining the Weights

It is easy to determine equations for the weights by combining the above equations:

$$f(\mathbf{x}^q) = \sum_{p=1}^{N} w_p \phi\left(\left\|\mathbf{x}^q - \mathbf{x}^p\right\|\right) = t^q$$

We can write this in matrix form by defining the vectors $\mathbf{t} = \{t^p\}$ and $\mathbf{w} = \{w_p\}$, and the matrix $\mathbf{\Phi} = \left\{\Phi_{pq} = \phi\left(\left\|\mathbf{x}^q - \mathbf{x}^p\right\|\right)\right\}$. This simplifies the equation to $\mathbf{\Phi}\,\mathbf{w} = \mathbf{t}$. Then, provided the inverse of $\mathbf{\Phi}$ exists, we can use any standard matrix inversion techniques to give

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

It can be shown that, for a large class of basis functions $\phi(\cdot)$, the matrix $\mathbf{\Phi}$ is indeed non-singular (and hence invertable) providing the data points are distinct.

Once we have the weights, we have a function $f(\mathbf{x})$ that represents a continuous differentiable surface that passes exactly through each data point.

Commonly Used Radial Basis Functions

A range of theoretical and empirical studies have indicated that many properties of the interpolating function are relatively insensitive to the precise form of the basis functions $\phi(r)$. Some of the most commonly used basis functions are:

1. **Gaussian Functions:**

$$\phi(r) = \exp\left(-\frac{r^2}{2\sigma^2}\right) \qquad \text{width parameter } \sigma > 0$$

2. **Multi-Quadric Functions**:

$$\phi(r) = \left(r^2 + \sigma^2\right)^{1/2} \qquad \text{parameter } \sigma > 0$$

3. **Generalized Multi-Quadric Functions**:

$$\phi(r) = \left(r^2 + \sigma^2\right)^{\beta} \qquad \text{parameters } \sigma > 0,\ 1 > \beta > 0$$

4. **Inverse Multi-Quadric Functions**:

$$\phi(r) = \left(r^2 + \sigma^2\right)^{-1/2} \qquad \text{parameter } \sigma > 0$$

5. **Generalized Inverse Multi-Quadric Functions**:

$$\phi(r) = \left(r^2 + \sigma^2\right)^{-\alpha} \qquad \text{parameters } \sigma > 0, \alpha > 0$$

6. **Thin Plate Spline Function**:

$$\phi(r) = r^2 \ln(r)$$

7. **Cubic Function**:

$$\phi(r) = r^3$$

8. **Linear Function**:

$$\phi(r) = r$$

Properties of the Radial Basis Functions

The Gaussian and Inverse Multi-Quadric Functions are ***localised*** in the sense that

$$\phi(r) \to 0 \quad \text{as} \quad |\mathrm{r}| \to \infty$$

but this is not strictly necessary. All the other functions above have the property

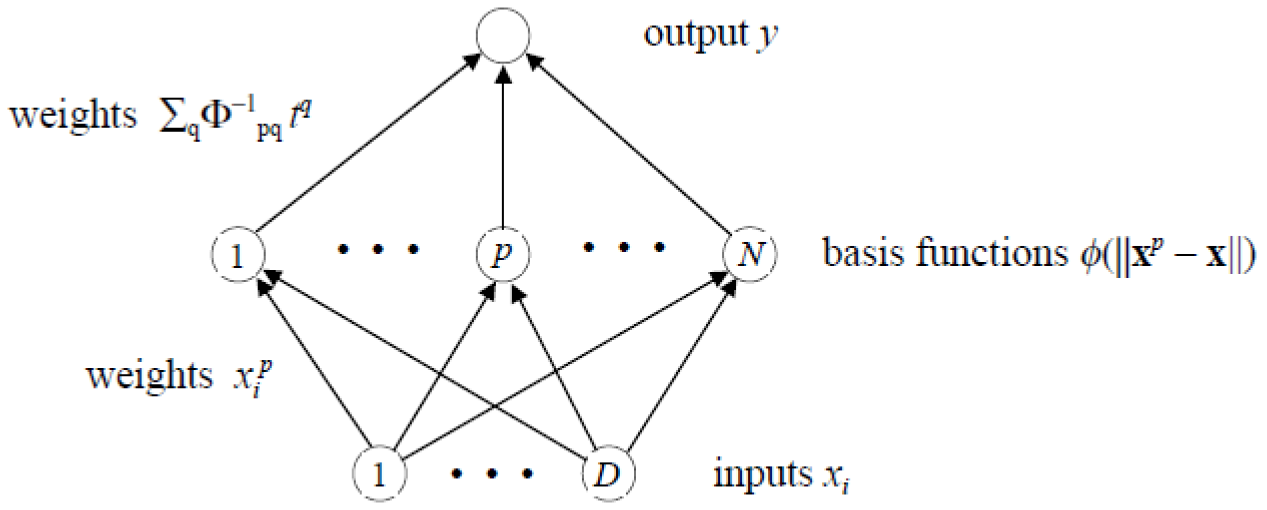
$$\phi(r) \to \infty \quad \text{as} \quad |\mathrm{r}| \to \infty$$

Note that even the Linear Function $\phi(r) = r = \left\| \mathbf{x} - \mathbf{x}^p \right\|$ is still non-linear in the components of $\mathbf{x}$. In one dimension, this leads to a piecewise-linear interpolating function which performs the simplest form of exact interpolation.

For neural network mappings, there are good reasons for preferring localised basis functions. We shall focus our attention on ***Gaussian basis functions*** since, as well as being localised, they have a number of other useful analytic properties. We can also see intuitively how to set their widths σ and build up function approximations with them.

Radial Basis Function Networks

Is it a Neural Network?



output y
weights $\sum_q \Phi^{-1}_{pq} t^q$
1
p
N
basis functions $\phi(\|\mathbf{x}^p - \mathbf{x}\|)$
weights x_i^p
1
D
inputs x_i

Note that the N training patterns $\{ x_i^p, t^p \}$ determine the weights directly. The hidden layer to output weights multiply the hidden unit activations in the conventional manner, but the input to hidden layer weights are used in a very different fashion.

The Radial Basis Function (RBF) Mapping

We are working within the standard framework of function approximation. We have a set of N data points in a multi-dimensional space such that every D dimensional input vector $\mathbf{x}^p = \{x_i^p : i = 1, \ldots, D\}$ has a corresponding K dimensional target output $\mathbf{t}^p = \{t_k^p : k = 1, \ldots, K\}$. The target outputs will generally be generated by some underlying functions $g_k(\mathbf{x})$ plus random noise. The goal is to approximate the $g_k(\mathbf{x})$ with functions $y_k(\mathbf{x})$ of the form

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

We shall concentrate on the case of Gaussian basis functions

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

in which we have basis centres $\{\boldsymbol{\mu}_j\}$ and widths $\{\sigma_j\}$. Naturally, the way to proceed is to develop a process for finding the appropriate values for M, $\{w_{kj}\}$, $\{\mu_{ij}\}$ and $\{\sigma_j\}$.

Computational Power of RBF Networks

Intuitively, we can easily understand why linear superpositions of localised basis functions are capable of universal approximation. More formally:

Hartman, Keeler & Kowalski (1990, *Neural Computation*, vol.2, pp.210-215) provided a formal proof of this property for networks with Gaussian basis functions in which the widths $\{\sigma_j\}$ are treated as adjustable parameters.

Park & Sandberg (1991, *Neural Computation*, vol.3, pp.246-257; and 1993, *Neural Computation*, vol.5, pp.305-316) showed that with only mild restrictions on the basis functions, the universal function approximation property still holds.

As with the corresponding proofs for MLPs, these are existence proofs which rely on the availability of an arbitrarily large number of hidden units (i.e. basis functions). However, they do provide a theoretical foundation on which practical applications can be based with confidence.

Training RBF Networks

The proofs about computational power tell us what an RBF Network can do, but nothing about how to find all its parameters/weights $\{w_{kj}, \mu_{ij}, \sigma_j\}$.

Unlike in MLPs, in RBF networks the hidden and output layers play very different roles, and the corresponding "weights" have very different meanings and properties. It is therefore appropriate to use different learning algorithms for them.

The input to hidden "weights" (i.e. basis function parameters $\{\mu_{ij}, \sigma_j\}$) can be trained (or set) using any of a number of unsupervised learning techniques.

Then, after the input to hidden "weights" are found, they are kept fixed while the hidden to output weights are learned. Since this second stage of training involves just a single layer of weights $\{w_{jk}\}$ and linear output activation functions, the weights can easily be found analytically by solving a set of linear equations. This can be done very quickly, without the need for a set of iterative weight updates as in gradient descent learning.

Basis Function Optimization

One major advantage of RBF networks is the possibility of choosing suitable hidden unit/basis function parameters without having to perform a full non-linear optimization of the whole network. We shall now look at several ways of doing this:

1. Fixed centres selected at random
2. Orthogonal least squares
3. K-means clustering

Unsupervised Learning

With either approach, determining a good value for M remains a problem. It will generally be appropriate to compare the results for a range of different values, following the same kind of validation/cross validation methodology used for optimizing MLPs.

Fixed Centres Selected At Random

The simplest and quickest approach to setting the RBF parameters is to have their centres fixed at M points selected at ***random*** from the N data points, and to set all their widths to be equal and fixed at an appropriate size for the distribution of data points.

Specifically, we can use normalised RBFs centred at $\{\boldsymbol{\mu}_j\}$ defined by

$$\phi_j(\mathbf{x}) = \exp\left(-\frac{\|\mathbf{x}-\boldsymbol{\mu}_j\|^2}{2\sigma_j{}^2}\right) \qquad \text{where} \quad \{\boldsymbol{\mu}_j\} \subset \{\mathbf{x}^p\}$$

and the σ_j are all related in the same way to the maximum or average distance between the chosen centres $\boldsymbol{\mu}_j$. Common choices are

$$\sigma_j = \frac{d_{\max}}{\sqrt{2M}} \qquad \text{or} \qquad \sigma_j = 2d_{\text{ave}}$$

which ensure that the individual RBFs are neither too wide, nor too narrow, for the given training data. For large training sets, this approach gives reasonable results.

Orthogonal Least Squares

A more principled approach to selecting a sub-set of data points as the basis function centres is based on the technique of ***orthogonal least squares***.

This involves the sequential addition of new basis functions, each centred on one of the data points. At each stage, we try out each potential Lth basis function by using the $N–L$ other data points to determine the networks output weights. The potential Lth basis function which leaves the smallest residual output sum squared output error is used, and we move on to choose which $L+1$th basis function to add.

This sounds wasteful, but if we construct a set of orthogonal vectors in the space S spanned by the vectors of hidden unit activations for each pattern in the training set, we can calculate directly which data point should be chosen as the next basis function.

To get good generalization we generally use validation/cross validation to stop the process when an appropriate number of data points have been selected as centres.

K-Means Clustering

A potentially even better approach is to use clustering techniques to find a set of centres which more accurately reflects the distribution of the data points.

The ***K-Means Clustering Algorithm*** picks the number K of centres in advance, and then follows a simple re-estimation procedure to partition the data points $\{\mathbf{x}^p\}$ into K disjoint sub-sets S_j containing N_j data points to minimize the sum squared clustering function

$$J = \sum_{j=1}^{K} \sum_{p \in S_j} \left\| \mathbf{x}^p - \boldsymbol{\mu}_j \right\|^2$$

where $\boldsymbol{\mu}_j$ is the mean/centroid of the data points in set S_j given by

$$\boldsymbol{\mu}_j = \frac{1}{N_j} \sum_{p \in S_j} \mathbf{x}^p$$

Once the basis centres have been determined in this way, the widths can then be set according to the variances of the points in the corresponding cluster.

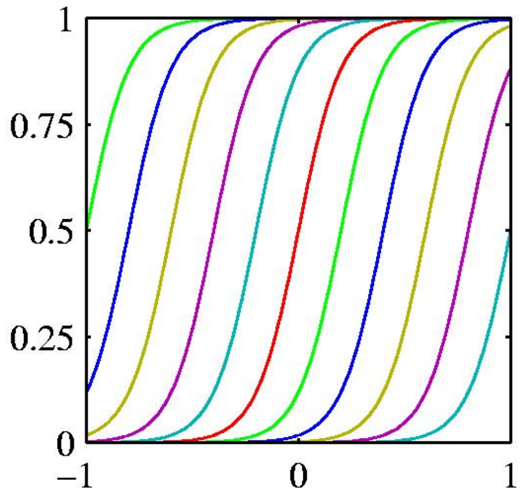
References

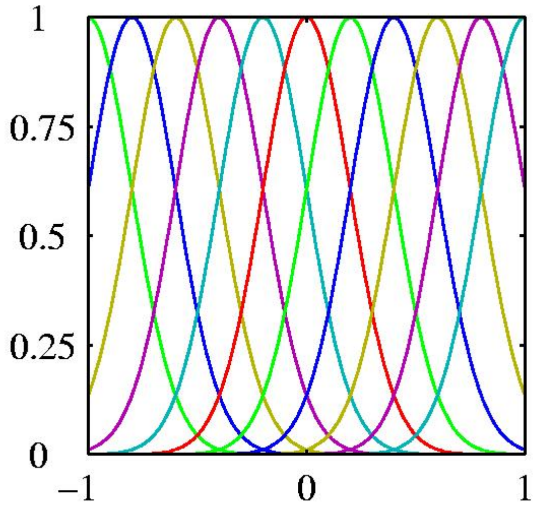
1. Simon Haykins – Neural Networks and Learning Machines
2. Christopher Bishop – Neural Networks in Pattern Recognition
3. Kevin Gurney – Introduction to Neural Networks
4. Hertz, Krogh, Palmer – Introduction to theory of Neural computation (Santa Fe Institute Series)
5. Handbook of Brain Sciences & Neural Networks

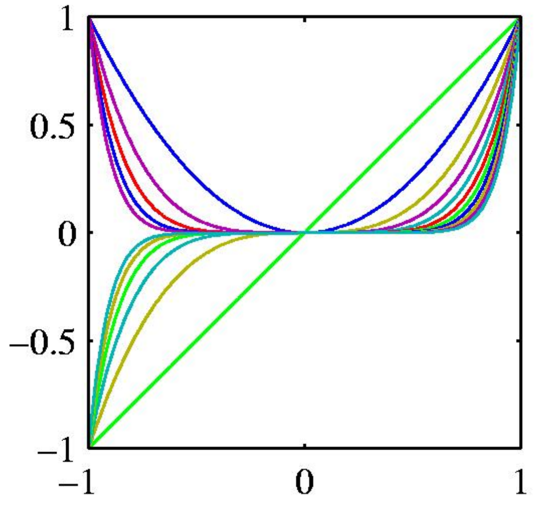
Linear models

- It is mathematically easy to fit linear models to data.
  - We can learn a lot about model-fitting in this relatively simple case.
- There are many ways to make linear models more powerful while retaining their nice mathematical properties:
  - By using non-linear, non-adaptive basis functions, we can get generalised linear models that learn non-linear mappings from input to output but are linear in their parameters – only the linear part of the model learns.
  - By using kernel methods we can handle expansions of the raw data that use a huge number of non-linear, non-adaptive basis functions.
  - By using large margin kernel methods we can avoid overfitting even when we use huge numbers of basis functions.
- But linear methods will not solve most AI problems.
  - They have fundamental limitations.

Some types of basis function in 1-D







Sigmoids Gaussians Polynomials

Sigmoid and Gaussian basis functions can also be used in multilayer neural networks, but neural networks learn the parameters of the basis functions. This is much more powerful but also much harder and much messier.

Two types of linear model that are equivalent with respect to learning

$$y(\mathbf{x},\mathbf{w}) = \overset{\text{bias}}{\overset{\downarrow}{w_0}} + w_1 x_1 + w_2 x_2 + \ldots = \mathbf{w}^T \mathbf{x}$$

$$y(\mathbf{x},\mathbf{w}) = w_0 + w_1 \phi_1(\mathbf{x}) + w_2 \phi_2(\mathbf{x}) + \ldots = \mathbf{w}^T \Phi(\mathbf{x})$$

- The first model has the same number of adaptive coefficients as the dimensionality of the data +1.
- The second model has the same number of adaptive coefficients as the number of basis functions +1.
- Once we have replaced the data by the outputs of the basis functions, fitting the second model is exactly the same problem as fitting the first model (unless we use the kernel trick)
  - So its silly to clutter up the math with basis functions

The loss function

- Fitting a model to data is typically done by finding the parameter values that minimize some loss function.
- There are many possible loss functions. What criterion should we use for choosing one?
  - Choose one that makes the math easy (squared error)
  - Choose one that makes the fitting correspond to maximizing the likelihood of the training data given some noise model for the observed outputs.
  - Choose one that makes it easy to interpret the learned coefficients (easy if mostly zeros)
  - Choose one that corresponds to the real loss on a practical application (losses are often asymmetric)

Minimizing squared error

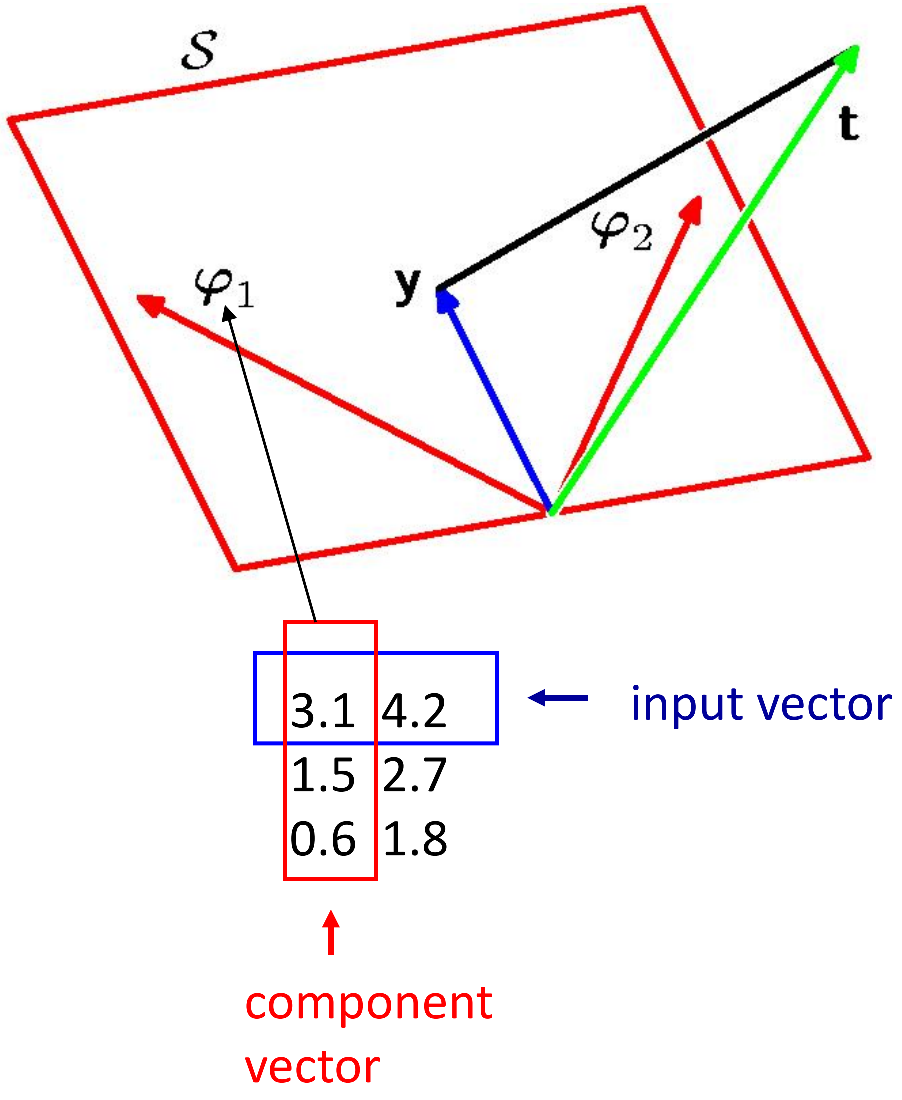
$$y = \mathbf{w}^T \mathbf{x}$$

$$error = \sum_n (t_n - \mathbf{w}^T \mathbf{x}_n)^2$$

$$\underset{\text{optimal weights}}{\mathbf{w}^*} = \underbrace{\boxed{(\mathbf{X}^T \mathbf{X})^{-1}}}_{\text{inverse of the covariance matrix of the input vectors}} \underbrace{\mathbf{X}^T}_{\text{the transposed design matrix has one input vector per column}} \boxed{\mathbf{t}} \leftarrow \text{vector of target values}$$

A geometrical view of the solution

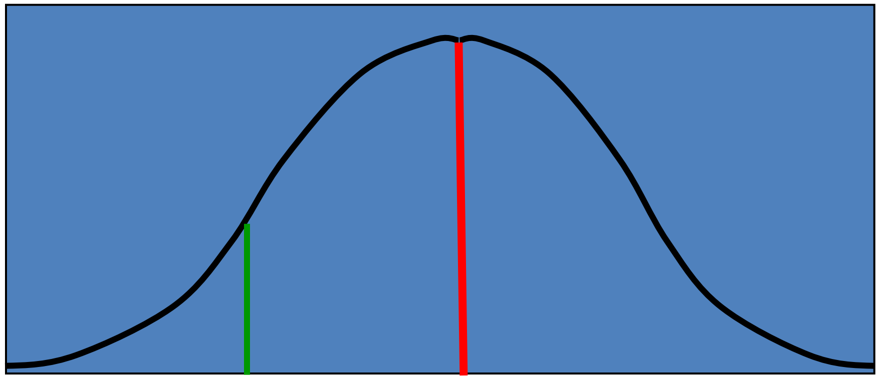
- The space has one axis for each training case.
- So the vector of target values is a point in the space.
- Each vector of the values of one component of the input is also a point in this space.
- The input component vectors span a subspace, S.
  - A weighted sum of the input component vectors must lie in S.
- The optimal solution is the orthogonal projection of the vector of target values onto S.



3.1	4.2
1.5	2.7
0.6	1.8

When is minimizing the squared error equivalent to Maximum Likelihood Learning?

- Minimizing the squared residuals is equivalent to maximizing the log probability of the correct answer under a Gaussian centered at the model's guess.



t = the correct answer

y = model's estimate of most probable value

$$y_n = y(\mathbf{x}_n, \mathbf{w})$$

$$p(t_n \mid y_n) = p(y_n + noise = t_n \mid \mathbf{x}_n, \mathbf{w}) = \frac{1}{\sqrt{2\pi}\sigma} e^{-\frac{(t_n - y_n)^2}{2\sigma^2}}$$

$$-\log p(t_n \mid y_n) = \log\sqrt{2\pi} + \underset{\uparrow}{\log\sigma} + \frac{(t_n - y_n)^2}{2\sigma^2}$$

can be ignored if sigma is fixed

can be ignored if sigma is same for every case

Multiple outputs

- If there are multiple outputs we can often treat the learning problem as a set of independent problems, one per output.
  - Not true if the output noise is correlated and changes from case to case.
- Even though they are independent problems we can save work by only multiplying the input vectors by the inverse covariance of the input components once. For output k we have:

$$\mathbf{w}_k^* = \underbrace{\boxed{(\mathbf{X}^T\,\mathbf{X})^{-1}\;\mathbf{X}^T}}_{\uparrow}\,\mathbf{t}_k$$

does not depend on a

Least mean squares: An alternative approach for really big datasets

$$\underset{\substack{\uparrow \\ \text{weights after seeing training case tau+1}}}{\mathbf{w}^{\tau+1}} = \mathbf{w}^{\tau} - \underset{\substack{\uparrow \\ \text{learning rate}}}{\eta} \underset{\substack{\uparrow \\ \text{vector of derivatives of the squared error w.r.t. the weights on the training case presented at time tau.}}}{\boxed{\nabla E_{n(\tau)}}}$$

- This is called “online“ learning. It can be more efficient if the dataset is very redundant and it is simple to implement in hardware.
  - It is also called stochastic gradient descent if the training cases are picked at random.
  - Care must be taken with the learning rate to prevent divergent oscillations, and the rate must decrease at the end to get a good fit.

Regularized least squares

$$\widetilde{E}(\mathbf{w}) = \frac{1}{2}\sum_{n=1}^{N}\{y(\mathbf{x}_n, \mathbf{w}) - t_n\}^2 + \frac{\lambda}{2}\|\mathbf{w}\|^2$$

The penalty on the squared weights is mathematically compatible with the squared error function, so we get a nice closed form for the optimal weights with this regularizer:

$$\mathbf{w}^* = (\underset{\substack{\uparrow \\ \text{identity matrix}}}{\lambda \mathbf{I}} + \mathbf{X}^T \mathbf{X})^{-1} \mathbf{X}^T \mathbf{t}$$

Commonly Used Radial Basis Functions

A range of theoretical and empirical studies have indicated that many properties of the interpolating function are relatively insensitive to the precise form of the basis functions $\phi(r)$. Some of the most commonly used basis functions are:

1. **Gaussian Functions:**

$$\phi(r) = \exp\left(-\frac{r^2}{2\sigma^2}\right) \qquad \text{width parameter } \sigma > 0$$

2. **Multi-Quadric Functions:**

$$\phi(r) = \left(r^2 + \sigma^2\right)^{1/2} \qquad \text{parameter } \sigma > 0$$

3. **Generalized Multi-Quadric Functions:**

$$\phi(r) = \left(r^2 + \sigma^2\right)^{\beta} \qquad \text{parameters } \sigma > 0,\ 1 > \beta > 0$$

4. **Inverse Multi-Quadric Functions**:

$$\phi(r) = \left(r^2 + \sigma^2\right)^{-1/2} \qquad \text{parameter } \sigma > 0$$

5. **Generalized Inverse Multi-Quadric Functions**:

$$\phi(r) = \left(r^2 + \sigma^2\right)^{-\alpha} \qquad \text{parameters } \sigma > 0, \alpha > 0$$

6. **Thin Plate Spline Function**:

$$\phi(r) = r^2 \ln(r)$$

7. **Cubic Function**:

$$\phi(r) = r^3$$

8. **Linear Function**:

$$\phi(r) = r$$

Radial Basis Functions

Let $\varphi: R^{+} \to R$ be a continous function with $\varphi(0) \geq 0$. If $\mathbf{x}_i \in \Omega$, let

$$\varphi_i\left(\mathbf{x}_i\right) = \varphi\left(\left\|\mathbf{x} - \mathbf{x}_i\right\|\right),$$

where $\|\bullet\|$ is the Euclidean norm. Then φ_i is called the RBF.

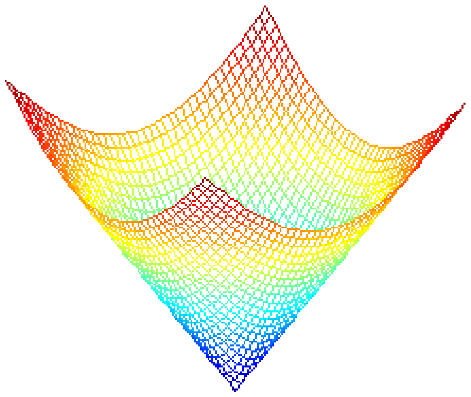
Linear:	r
Cubic:	r^3
Multiquadrics:	$\sqrt{r^2 + c^2}$ where c is a shape parameter.
Polyharmonic Spines:	$\begin{cases} r^{2n} \log r, \quad n \geq 1, & \text{in 2D}, \\ r^{2n-1}, \ n \geq 1, & \text{in 3D}. \end{cases}$
Gaussian:	e^{-cr^2}

Globally Supported RBFs

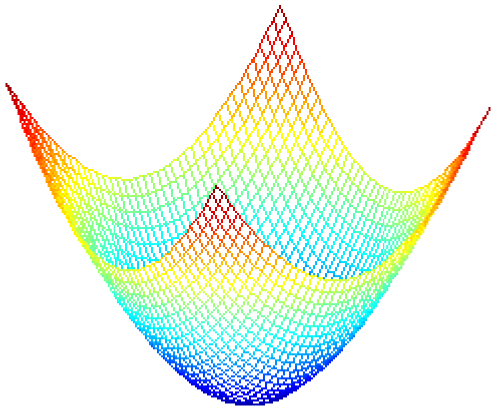
$\varphi = 1+r$

$$\varphi = \sqrt{r^2 + c^2}$$

1+r



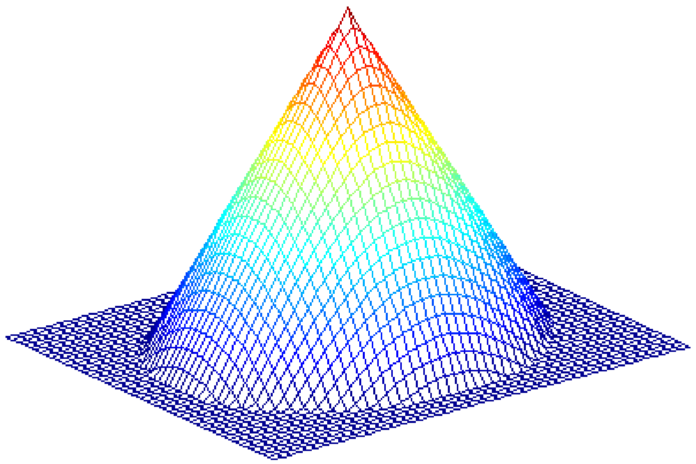
Classical rbf: MQ



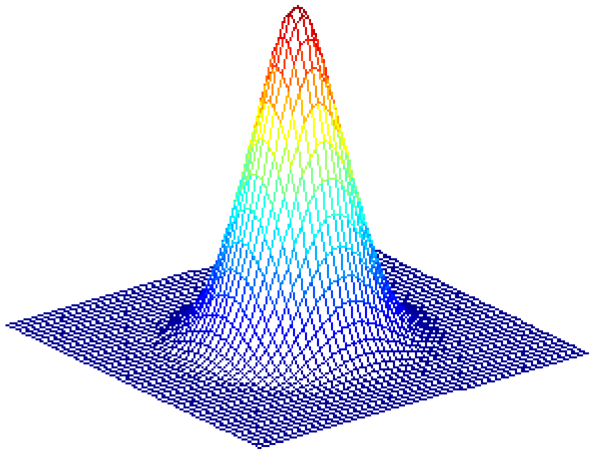
Compactly Supported RBFs

$$\varphi = (1-r)_+^2$$

$$\varphi = (1-r)_+^4(4r+1)$$



CS-PD-RBF $(4r+1)(r-1)^4$



Computing the Output Weights

Our equations for the weights are most conveniently written in matrix form by defining matrices with components $(\mathbf{W})_{kj} = w_{kj}$, $(\mathbf{\Phi})_{pj} = \phi_j(\mathbf{x}^p)$, and $(\mathbf{T})_{pk} = \{t_k^p\}$. This gives

$$\mathbf{\Phi}^{\mathsf{T}}\left(\mathbf{\Phi}\mathbf{W}^{\mathsf{T}} - \mathbf{T}\right) = 0$$

and the formal solution for the weights is

$$\mathbf{W}^{\mathsf{T}} = \mathbf{\Phi}^{\dagger}\mathbf{T}$$

in which we have the standard ***pseudo inverse*** of $\mathbf{\Phi}$

$$\mathbf{\Phi}^{\dagger} \equiv (\mathbf{\Phi}^{\mathsf{T}}\mathbf{\Phi})^{-1}\mathbf{\Phi}^{\mathsf{T}}$$

which can be seen to have the property $\mathbf{\Phi}^{\dagger}\mathbf{\Phi} = \mathbf{I}$. We see that the network weights can be computed by fast linear matrix inversion techniques. In practice we tend to use singular value decomposition (SVD) to avoid possible ill-conditioning of $\mathbf{\Phi}$, i.e. $\mathbf{\Phi}^{\mathsf{T}}\mathbf{\Phi}$ being singular or near singular.

Supervised RBF Network Training

Supervised training of the basis function parameters will generally give better results than unsupervised procedures, but the computational costs are usually enormous.

The obvious approach is to perform gradient descent on a sum squared output error function as we did for our MLPs. The error function would be

$$E = \sum_p \sum_k (y_k(\mathbf{x}^p) - t_k^p)^2 = \sum_p \sum_k (\sum_{j=0}^{M} w_{kj}\phi_j\left(\mathbf{x}^p, \boldsymbol{\mu}_j, \sigma_j\right) - t_k^p)^2$$

and we would iteratively update the weights/basis function parameters using

$$\Delta w_{jk} = -\eta_w \frac{\partial E}{\partial w_{jk}} \qquad \Delta \mu_{ij} = -\eta_\mu \frac{\partial E}{\partial \mu_{ij}} \qquad \Delta \sigma_j = -\eta_\sigma \frac{\partial E}{\partial \sigma_i}$$

We have all the problems of choosing the learning rates η, avoiding local minima and so on, that we had for training MLPs by gradient descent. Also, there is a tendency for the basis function widths to grow large leaving non-localised basis functions.

Regularization Theory for RBF Networks

Instead of restricting the number of hidden units, an alternative approach for preventing overfitting in RBF networks comes from the theory of ***regularization***, which we saw previously was a method of controlling the smoothness of mapping functions.

We can have one basis function centred on each training data point as in the case of exact interpolation, but add an extra term to the error measure which penalizes mappings which are not smooth. If we have network outputs $y_k(\mathbf{x}^p)$ and sum squared error measure, we can introduce some appropriate differential operator $\mathbf{P}$ and write

$$E = \frac{1}{2}\sum_p \sum_k (y_k(\mathbf{x}^p) - t_k^p)^2 + \lambda \sum_k \int \left|\mathbf{P} y_k(\mathbf{x})\right|^2 d\mathbf{x}$$

where λ is the regularization parameter which determines the relative importance of smoothness compared with error. There are many possible forms for $\mathbf{P}$, but the general idea is that mapping functions $y_k(\mathbf{x})$ which have large curvature should have large values of $|\mathbf{P}y_k(\mathbf{x})|^2$ and hence contribute a large penalty in the total error function.

Computing the Regularized Weights

Provided the regularization term is quadratic in $y_k(\mathbf{x})$, the second layer weights can still be found by solving a set of linear equations. For example, the regularizer

$$\lambda \sum_k \left[\sum_p \sum_i \frac{1}{2} \left(\frac{\partial^2 y_k(\mathbf{x}^p)}{\partial x_i^{\,2}} \right)^2 \right]$$

certainly penalizes large output curvature, and minimizing the error function E now leads to linear equations for the output weights that are no harder to solve than we had before

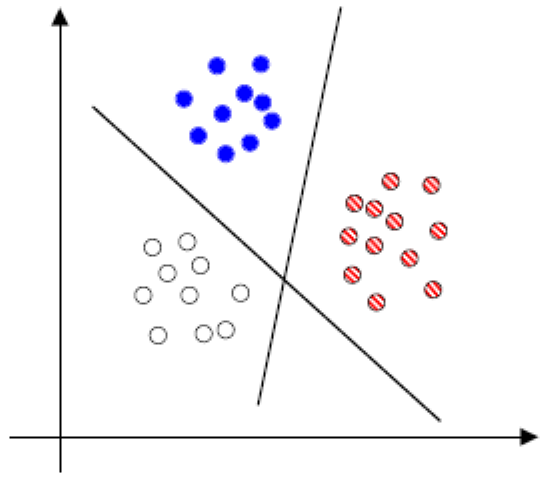
$$\mathbf{M}\mathbf{W}^{\mathsf{T}} - \mathbf{\Phi}^{\mathsf{T}}\mathbf{T} = 0$$

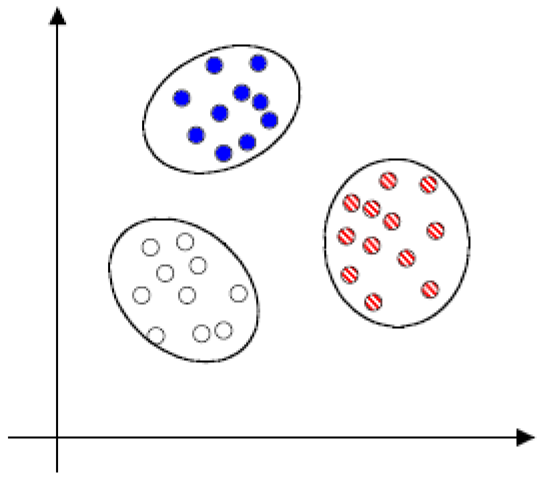
In this we have defined the same matrices with components $(\mathbf{W})_{kj} = w_{kj}$, $(\mathbf{\Phi})_{pj} = \phi_j(\mathbf{x}^p)$, and $(\mathbf{T})_{pk} = \{t_k^p\}$ as before, and also the regularized version of $\mathbf{\Phi}^{\mathsf{T}}\mathbf{\Phi}$

$$\mathbf{M} = \mathbf{\Phi}^{\mathsf{T}}\mathbf{\Phi} + \lambda \sum_i \frac{\partial^2 \mathbf{\Phi}^{\mathsf{T}}}{\partial x_i^{\,2}} \frac{\partial^2 \mathbf{\Phi}}{\partial x_i^{\,2}}$$

RBF Networks for Classification

So far we have concentrated on RBF networks for function approximation. They are also useful for classification problems. Consider a data set that falls into three classes:





An MLP would naturally separate the classes with hyper-planes in the input space (as on the left). An alternative approach would be to model the separate class distributions by localised radial basis functions (as on the right).

Implementing RBF Classification Networks

In principle, it is easy to have an RBF network perform classification – we simply need to have an output function $y_k(\mathbf{x})$ for each class k with appropriate targets

$$t_k^p = \begin{cases} 1 & \text{if pattern } p \text{ belongs to class } k \\ 0 & \text{otherwise} \end{cases}$$

and, when the network is trained, it will automatically classify new patterns.

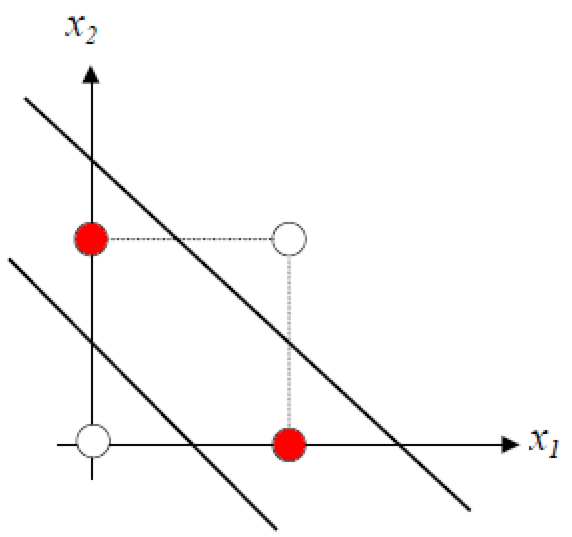
The underlying justification is found in ***Cover's theorem*** which states that "A complex pattern classification problem cast in a high dimensional space non-linearly is more likely to be linearly separable than in a low dimensional space". We know that once we have linear separable patterns, the classification problem is easy to solve.

In addition to the RBF network outputting good classifications, it can be shown that the outputs of such a regularized RBF network classifier will also provide estimates of the *posterior class probabilities*.

The XOR Problem Revisited

We are already familiar with the non-linearly separable XOR problem:

p	x_1	x_2	t
1	0	0	0
2	0	1	1
3	1	0	1
4	1	1	0



We know that Single Layer Perceptrons with step or sigmoidal activation functions cannot generate the right outputs, because they are only able to form a single decision boundary. To deal with this problem using Perceptrons we needed to either change the activation function, or introduce a non-linear hidden layer to give an Multi Layer Perceptron (MLP).

The XOR Problem in RBF Form

Recall that sensible RBFs are M Gaussians $\phi_j(\mathbf{x})$ centred at random training data points:

$$\phi_j(\mathbf{x}) = \exp\left(-\frac{M}{d_{\max}^2} \left\| \mathbf{x} - \boldsymbol{\mu}_j \right\|^2 \right) \qquad \text{where} \quad \{\boldsymbol{\mu}_j\} \subset \{\mathbf{x}^p\}$$

To perform the XOR classification in an RBF network, we start by deciding how many basis functions we need. Given there are four training patterns and two classes, $M = 2$ seems a reasonable first guess. We then need to decide on the basis function centres. The two separated zero targets seem a good bet, so we can set $\boldsymbol{\mu}_1 = (0,0)$ and $\boldsymbol{\mu}_2 = (1,1)$ and the distance between them is $d_{max} = \sqrt{2}$. We thus have the two basis functions

$$\phi_1(\mathbf{x}) = \exp\left(-\left\| \mathbf{x} - \boldsymbol{\mu}_1 \right\|^2 \right) \qquad \text{with} \quad \boldsymbol{\mu}_1 = (0,0)$$

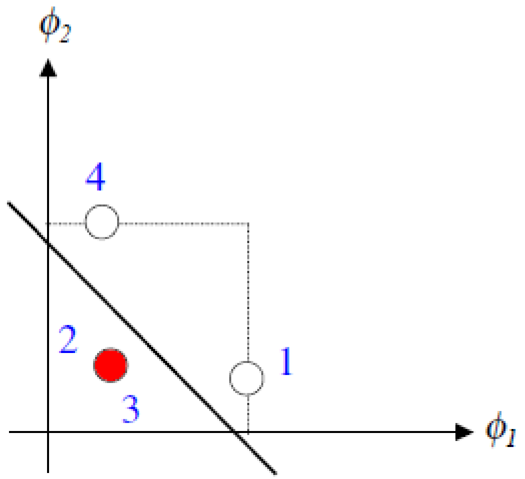
$$\phi_2(\mathbf{x}) = \exp\left(-\left\| \mathbf{x} - \boldsymbol{\mu}_2 \right\|^2 \right) \qquad \text{with} \quad \boldsymbol{\mu}_2 = (1,1)$$

This will hopefully transform the problem into a linearly separable form.

The XOR Problem Basis Functions

Since the hidden unit activation space is only two dimensional we can easily plot how the input patterns have been transformed:

p	x_1	x_2	ϕ_1	ϕ_2
1	0	0	1.0000	0.1353
2	0	1	0.3678	0.3678
3	1	0	0.3678	0.3678
4	1	1	0.1353	1.0000



We can see that the patterns are now linearly separable. Note that in this case we did not have to increase the dimensionality from the input space to the hidden unit/basis function space – the non-linearity of the mapping was sufficient. **Exercise**: check what happens if you chose two different basis function centres.

The XOR Problem Output Weights

In this case we just have one output $y(\mathbf{x})$, with one weight w_j to each hidden unit j, and one bias $-\theta$. This gives us the network's input-output relation for each input pattern $\mathbf{x}$

$$y(\mathbf{x}) = w_1\phi_1(\mathbf{x}) + w_2\phi_2(\mathbf{x}) - \theta$$

Then, if we want the outputs $y(\mathbf{x}^p)$ to equal the targets t^p, we get the four equations

$$\begin{aligned}
1.0000w_1 + 0.1353w_2 - 1.0000\theta &= 0\\
0.3678w_1 + 0.3678w_2 - 1.0000\theta &= 1\\
0.3678w_1 + 0.3678w_2 - 1.0000\theta &= 1\\
0.1353w_1 + 1.0000w_2 - 1.0000\theta &= 0
\end{aligned}$$

Three are different, and we have three variables, so we can easily solve them to give

$$w_1 = w_2 = -2.5018 \quad , \quad \theta = -2.8404$$

This completes our "training" of the RBF network for the XOR problem.

Comparison of RBF Networks with MLPs

There are clearly a number of similarities between RBF networks and MLPs:

Similarities

1. They are both non-linear feed-forward networks
2. They are both universal approximators
3. They are used in similar application areas

It is not surprising, then, to find that there always exists an RBF network capable of accurately mimicking a specified MLP, or vice versa. However the two networks do differ from each other in a number of important respects:

Differences

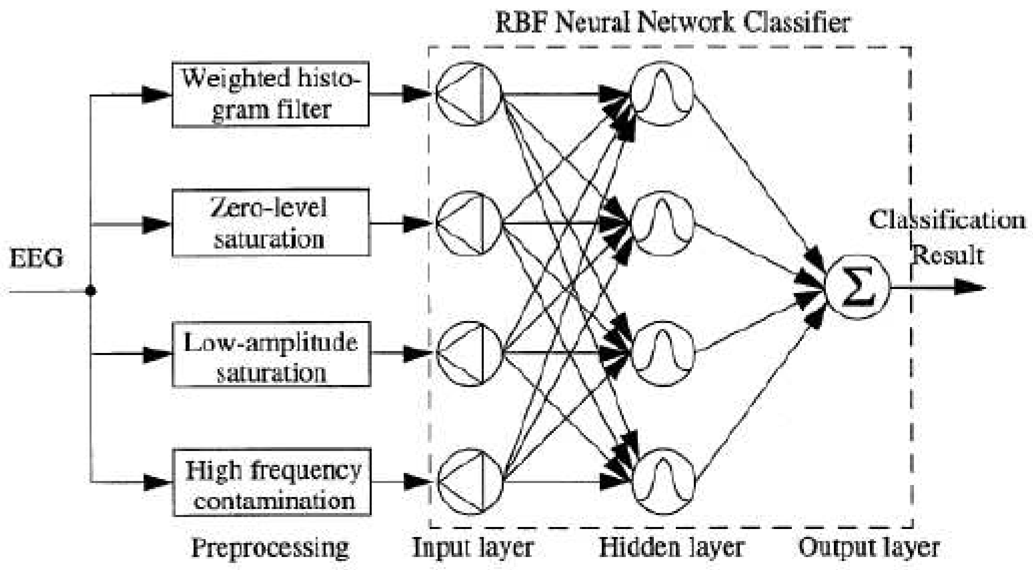
1. An RBF network (in its natural form) has a single hidden layer, whereas MLPs can have any number of hidden layers.

2. RBF networks are usually fully connected, whereas it is common for MLPs to be only partially connected.
3. In MLPs the computation nodes (processing units) in different layers share a common neuronal model, though not necessarily the same activation function. In RBF networks the hidden nodes (basis functions) operate very differently, and have a very different purpose, to the output nodes.
4. In RBF networks, the argument of each hidden unit activation function is the ***distance*** between the input and the "weights" (RBF centres), whereas in MLPs it is the ***inner product*** of the input and the weights.
5. MLPs are usually trained with a single global supervised algorithm, whereas RBF networks are usually trained one layer at a time with the first layer unsupervised.
6. MLPs construct ***global*** approximations to non-linear input-output mappings with ***distributed*** hidden representations, whereas RBF networks tend to use ***localised*** non-linearities (Gaussians) at the hidden layer to construct ***local*** approximations.

Although, for approximating non-linear input-output mappings, the RBF networks can be trained much faster, MLPs may require a smaller number of parameters.

Real World Application – EEG Analysis

One successful RBF network detects epileptiform artefacts in EEG recordings:



For full details see the original paper by: A. Saastamoinen, T. Pietilä, A. Värri, M. Lehtokangas, & J. Saarinen, (1998). Waveform detection with RBF network – Application to automated EEG analysis. *Neurocomputing,* vol. **20**, pp. 1-13