Table of Contents

Chapter V- Function Approximation with MLPs, Radial Basis Functions, and Support Vector Machines

Version 2.0

This Chapter is Part of:

Neural and Adaptive Systems: Fundamentals Through Simulation© *by*

Jose C. Principe
Neil R. Euliano
W. Curt Lefebvre

This chapter provides an unifying perspective of adaptive systems by linking the concepts of function approximation, classification, regression and density approximation. We will introduce the radial basis functions (RBFs) as an alternate topology to implement classifiers or function approximators. Finally we will present the structural risk minimization principle and its implementation as support vector machines.

Go to the next section

1. Introduction

In Chapter I and III we presented two of the most common applications of adaptive systems which are respectively linear regression utilizing a linear adaptive system (the adaline), and classification using the multilayer perceptron (MLP). We saw that the nature of the applications was different since in regression the problem was one of representing the relationship between the input and the output data, while in classification the input data was assumed multi-class and the purpose was to separate them as accurately as possible. We also verified that the machinery developed for regression, i.e. gradient descent on a cost function, could be applied to classification. When properly extended the gradient descent procedure gave rise to the backpropagation algorithm developed to train the MLP.

The purpose of this chapter is to unify more formally the two applications of regression and classification. What we will be demonstrating is that both problems are in fact aspects of the more general problem of function approximation . Linear regression becomes function approximation with linear topologies, and classification becomes function approximation for a special type of functions called indicator functions. What we gain is a very broad perspective of the use of adaptive systems: they are systems that seek to represent an input-output relationship by changing at the same time the basis and the projections. This is unlike the most common function approximation schemes where the basis are fixed and only the projections change from signal to signal.

The MLP was utilized so far solely as a classifier but with this perspective becomes a general purpose nonlinear function approximation tool extending the adaline. This is a powerful perspective and will provide a lot of practical applications beyond classification ranging from system identification to data modeling, and will motivate the study of the MLP as a nonlinear regressor. The study of the MLP as a function approximator leads us to analyze the fundamental building blocks for function approximation, i.e. which are the

basis used by the MLP. It will also raise the question of alternate basis functions and what other neural topologies are universal function approximators. We will study the radial basis functions (RBFs) as another universal approximator and show that it can also be used as a classifier. In order to achieve this unifying view we have to present the basic concepts of function approximation, which will have the advantage of addressing other more well known basis functions and contrast them with the MLP and the RBFs.

1.1. The discovery of the input-output map as function approximation

We have demonstrated in Chapter I and III that a neural network combines a set of inputs to obtain an output that mimics the desired response. Given a set of input vectors ***x***, and a set of desired responses *d* the learning system must find the parameters that meet these specifications. This problem can be framed as function approximation, if one assumes that the desired response *d* is an unknown but fixed function of the input *d=f(**x**)* (Figure 1).

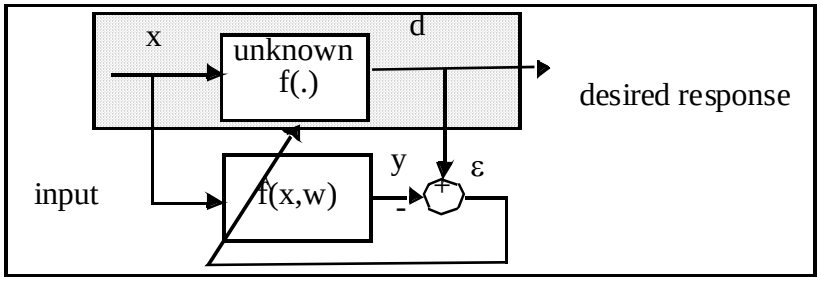


Figure 1. Supervised training as function approximation

The goal of the learning system is to discover the function *f(.)* given a finite number (hopefully small) of input-output pairs *(**x**,d)*. The learning machine output $y = \hat{f}(x, w)$ depends on a set of parameters ***w***, which can be modified to minimize the discrepancy between the system output *y* and the desired response *d*. When the network approximates *d* with *y* it is effectively approximating the unknown function *f(x)* by its input-output map $\hat{f}(x, w)$.

The nature of *f(.)* and the error criterion define the learning problem. As studied in

Chapter I, *linear regression* is obtained when the error criterion is the mean square error (MSE) and *f(.)* is linear. Classification, studied in Chapter II, specifies functions *f(.)* that produce 1, -1 (or 0) which are called *indicator functions*.

The problem of *generalization* already briefly discussed in Chapter IV can also be treated mathematically with this view of function approximation. This means that the ideas embodied in Figure 1 are rather relevant for the design of learning machines, specifically neural networks. Neural networks are in fact *nonlinear parametric function approximators*, so we should not think of them simply as classifiers.

ANNs are interesting to function approximation because:

- they are universal approximators
- they are efficient approximators
- and can be implemented as *learning machines*.

We already alluded in Chapter III to the universal approximation property of the MLP. It basically says that any function can be approximated by the MLP topology provided that enough PEs are available in the hidden layer. Here we will present more precisely these concepts.

With neural networks, the *coefficients* of the function decomposition are *automatically* obtained from the input-output data pairs and the specified topology using systematic procedures called the learning rules. So there is no need for tedious calculations to obtain analytically the parameters of the approximation. Once trained, the neural network becomes not only a parametric description of the function *but also its implementation*. Neural networks can be implemented in computers or analog hardware and trained on-line. This means that engineers and scientists have now means to solve function approximation problems involving real world data. The impact of this advance is to take function approximation out of the mathematician notebook and bring it to industrial applications.

Finally, we would like to argue that neural networks and learning are bringing focus to a

very important problem in the scientific method called *induction*. Induction is with *deduction* the only known systematic procedure to build scientific knowledge. Deduction applies general principles to specific situations. Deduction is pretty well understood, and has had enormous impact in all the fabric of mathematics, engineering computer science and science in general. For instance, deductive reasoning is the core of artificial intelligence. On the other hand induction is poorly understood and less applied. Induction is the principle of abstracting general rules from specific cases. As we all know from real life, this principle is much harder to apply with validity than deduction. Sometimes, true statements in a small set of cases do not generalize. Mathematically, induction is also much less formalized than deduction.

It turns out that a *neural network is using an inductive principle* when it learns from examples. Examples are specific instances of a general rule (the function that created the examples), and the goal of neural network learning is to seek the general principle that created the examples. Theoretically these issues are studied in learning theory . The difficulties we face in training appropriately a neural network are related to the difficulties of inducing general principles from examples. In practice, not always the ANN is able to capture the rule, and the pre-requisites (neural network architecture, training data, stopping criterion) to extrapolate from examples need to be carefully checked as we saw in Chapter IV.

Go to the next section

2. Function Approximation

Function approximation seeks to describe the behavior of very complicated functions by ensembles of simpler functions. Very important results have been established in this branch of mathematics. Here we will only name a few that bear a direct relation with our goal of better understanding neural networks. Legendre (and Gauss) used polynomials to approximate functions. Chebychev developed the concept of best uniform approximation. Weierstrass proved that polynomials can approximate arbitrarily well any continuous real

function in an interval. Series expansions (i.e. Taylor series) have been utilized for many years to compute approximately the value of a function in a neighborhood of the operating point. The core advantage is that only multiplications and additions are necessary to implement a series approximation. Trigonometric polynomials are also widely used as function approximators, but their computation is a bit more involved. We will formalize next the concept of function approximation.

Let ***f(x)*** be a real function of a real valued vector $\mathbf{x} = \begin{bmatrix} x_1 & x_2 & \dots & x_d \end{bmatrix}^T$ that is square integrable (over the real numbers). Most real world data can be modeled by such conditions. We are also going to restrict this study to the linear projection theorem. The goal of function approximation using the projection theorem is to describe the behavior of ***f(x)***, in a compact area S of the input space, by a combination of simpler functions $\varphi_i(\mathbf{x})$, i.e.

$$\hat{f}(\mathbf{x}, \mathbf{w}) = \sum_{i=1}^{N} w_i \varphi_i(\mathbf{x})$$

Equation 1

where w_i are real valued constants such that

$$\left| f(\mathbf{x}) - \hat{f}(\mathbf{x}, \mathbf{w}) \right| < \varepsilon$$

Equation 2

and where ε can be made arbitrarily small. The function $\hat{f}$ (**x**,**w**) is called an *approximant* to ***f(x)***. The block diagram of Figure 2 describes well this formulation.

Let us examine Eq. 1 and 2. A real function is a map from the input domain to the real numbers. So this expression states that one can obtain the value of the function when **x** is in S by using an intermediate set of simpler functions, $\{\varphi_i(\mathbf{x})\}$ called the *elementary functions* and then linearly combining them (Figure 2).

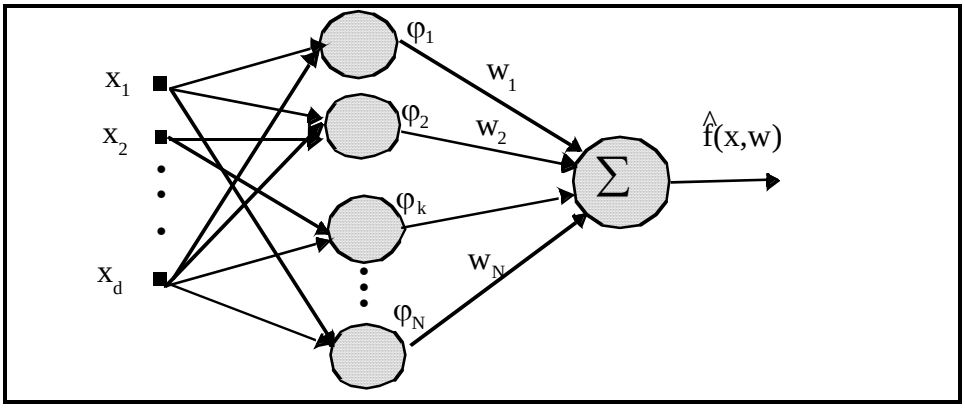


Figure 2. Implementation of the projection theorem

When one can find coefficients w_i that make ε arbitrarily small for any function *f(.)* over the domain of interest, we say that the elementary function set $\{\varphi_i(.)\}$ has the *property of universal approximation* over the class of functions *f(.),* or that the *set of elementary functions* $\varphi_i(\mathbf{x})$ *is complete*. From Eq. 1 we see that there are 3 basic decisions in function approximation:

- the choice of $\varphi_i(.)$,
- how to compute the w_i,
- how to select N.

The first problem is very rich because there are many possible elementary functions that can be used. We will illustrate this later, and we will show that the hidden PEs of a single hidden layer MLPs implement one possible choice for the elementary functions $\varphi_i(.)$.

The second problem is how to compute the coefficients w_i, which depends on how the difference or discrepancy between *f(x*) and $\hat{f}(x,w)$ is measured. In Chapter I we have already presented one possible machinery to solve this problem for the case of the minimization of the power of the error between $\hat{f}(x,w)$ and *f(x).* Least squares can be utilized also here to analytically compute the values for w_i. If the number of input vectors $\mathbf{x}_i$ is made equal to the number of elementary functions $\varphi_i(.)$, the normal equations can be written as

$$\begin{bmatrix} \varphi_1(x_1) & \varphi_2(x_1) & \varphi_N(x_1) \\ \varphi_1(x_N) & \varphi_2(x_N) & \varphi_N(x_N) \end{bmatrix} \begin{bmatrix} w_1 \\ w_N \end{bmatrix} = \begin{bmatrix} f(x_1) \\ f(x_N) \end{bmatrix}$$ **Equation 3**

and the solution becomes

$$\mathbf{w} = \Phi^{-1}\mathbf{f}$$ **Equation 4**

where ***w*** becomes a vector with the coefficients, ***f*** is a vector composed of the values of the function at the N points, and Φ the matrix with entries given by values of the elementary functions at each of the N points in the domain. An important condition that must be placed in the elementary functions is that the inverse of Φ must exist.

In general, there are many sets *{φi(.)}* with the property of universal approximation for a class of functions. We would prefer a set *{φi(.)}* over another *{γi(.)}* if *{φi(.)}* provides a smaller error ε for a pre-set value of N . This means that the speed of convergence of the approximation (i.e. how fast the approximation error ε decreases with N) is also an important factor in the selection of the basis. Other considerations may be imposed by the computer implementation.

2.1. Geometric Interpretation of the projection theorem

Let us provide a geometric interpretation for this decomposition because it exemplifies what is going on and what we try to accomplish. As long as the function *f(.)* is square integrable and N is finite, this geometric representation is accurate. Consider ***x*** as a given point in a N dimensional space. Its transformation by *f(.)* is assumed also to be another point in the same N dimensional space. We can alternatively think of ***x*** as a vector, with end points *0* and *x*. Likewise for ***f(x)***. For illustration purposes let us make N=3 and assume that we only have two elementary functions.

Eq.1 and 2 describe the projection of the vector ***f(x)*** into a set of *basis functions* *φi(x).* These basis functions can also be considered vectors and they define a manifold (i.e. a projection space) in M ($M \leq N$) dimensions, which is linear in our formulation. $\hat{f}$

(**x**,**w**) is the *image* or projection of ***f(x)*** in this manifold. In this example the projection manifold is a plane (M=2) depicted as the horizontal plane, and $\hat{f}$ (**x**,**w**) will be a vector that exists in the horizontal plane. We can interpret w_i as the magnitude of (or proportional to) $\hat{f}$ (**x**,**w**) along each one of the axis of the manifold.

If ***f(x)*** belongs to the manifold, then there is always a set of constants w_i that will make $\hat{f}$ (**x,w**) exactly equal to ***f(x).*** Figure 3 represents this in case A. If ***f(x)*** does not belong to the manifold created by the basis $\{\varphi_i(\boldsymbol{x})\}$, then there will always be an error between $\hat{f}$ (**x,w**) and ***f(x)*** (case B). The best solution (least possible error) found in Eq. 4 is the *orthogonal projection* of ***f(x)*** onto the manifold. As we saw in Chapter I this is exactly the solution that the least squares provide, since the error becomes orthogonal to all the basis $\{\varphi_i(\boldsymbol{x})\}$.

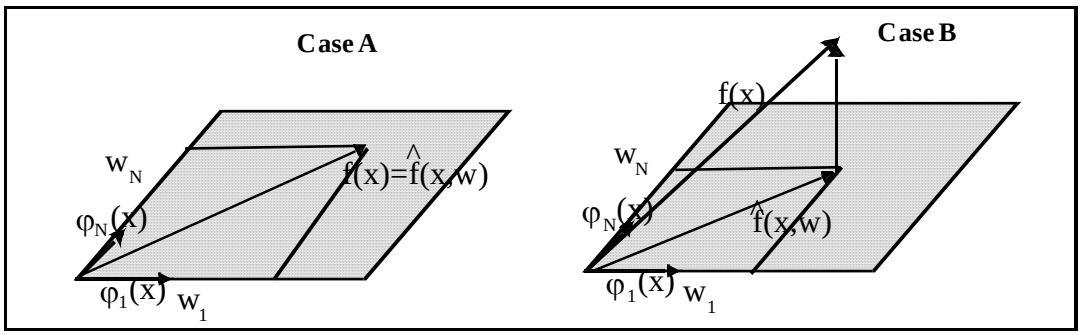


Figure 3 Approximation as a projection: A) vector is in the manifold. B) vector is outside de manifold.

When ***f(x)*** is external to the projection manifold, decreasing the error means making $\hat{f}$ (**x**,**w**) closer to ***f(x).*** This can be accomplished by increasing the number of elementary functions (i.e. the dimension M of the manifold) because the manifold will fill more and more of the available signal space. This view is correct provided that the basis set is complete, i.e. in the limit of large M the projection manifold will fill all the available signal space.

Let us now study in more detail each one of the steps in function approximation. We will see that throughout this study we will obtain a very different view of what the MLP is, and will tie this topology with other very well known basis functions.

Go to the next section

3. Choices for the elementary functions

One decisive step in function approximation is the choice of the elementary functions $\varphi_i(.)$ because they will impact how close $\hat{f}(\mathbf{x},\mathbf{w})$ can be made to $f(\mathbf{x})$. If the choice is not appropriate there will be a non vanishing error between $\hat{f}(\mathbf{x},\mathbf{w})$ and $f(\mathbf{x})$, no matter how big N is. The search for sets of elementary functions $\{\varphi_i(.)\}$ that are universal approximators of a class of functions $f(.)$ is therefore very important. Moreover, we would like the elementary functions $\varphi_i(.)$ to have nice mathematical properties and to be easy to work with.

One requirement for the usefulness of elementary functions in function approximation is that $\Phi^{-1}(\mathbf{x})$ must exist (Eq. 4). This condition is met if the elementary functions constitute a *basis*, i.e. if they are *linearly independent* or

$$w_1\varphi_1(x)+\ldots+w_N\varphi_N(x)=0 \quad iff \quad (w_1,\ldots,w_N)=0 \qquad \textbf{Equation 5}$$

A simplifying assumption that is often imposed on the elementary functions is that the basis be *orthonormal*, i.e. that

$$\int_s \varphi_i(x)\varphi_j(x)dx=\delta_{ij}(x) \qquad \textbf{Equation 6}$$

where $\delta(x)$ is the Dirac delta function . This means that in orthogonal decompositions the projection of a basis in any other basis is always zero. An orthonormal basis is very

appealing because one can evaluate the projection on each basis independently of the projection on the other bases, and they provide a unique set of w_i for the projection of ***f(x)***. But many elementary functions obey the orthogonality conditions, and so different sets may provide different properties.

With complete orthonormal basis the weights of the decomposition become very simple to compute. One can show that calculation of the orthonormal weights

$$w_i = < f(x), \varphi_i(x) > \quad \textbf{Equation 7}$$

where <.> is the inner product of *f(x)* with the bases, given by

$$< f(x), \varphi(x) > = \int_D f(x)\varphi(x)dx \quad \textbf{Equation 8}$$

and *D* is the domain where *f(x)* is defined.

3.1. Examples of elementary functions

In engineering, many important function approximation results are commonly applied. The usefulness of digital signal processing lies on the sampling theorem . The sampling theorem shows that one can approximate any real smooth signal (i.e. a function with finite slope) in an interval (infinitely many points) by knowing the functional values only at a finite set of equally spaced points in the interval (called the samples). The value of the signal at any other point in the interval can be exactly reconstructed by using sums of sinc functions. In this case the bases are the sinc functions and the weights are the values of the signal at the sampling points.

Figure 4. Decomposition by sinc functions.

This result opened up the use of sampled representations to reproduce sound (the compact disk (CD) contains just a stream of numbers) and to reproduce images (the forthcoming digital TV). And is the basis for the very important field of digital signal processing. sinc decomposition

NeuroSolutions 1

5.1 Sinc interpolation

Here we will use NeuroSolutions to interpolate an input waveform to a higher frequency using the sinc function. This example is not as dramatic as the one that produces from a digital sequence a continuous representations as alluded above, but it is based on the same principles. We will start with a digital waveform representing a ramp, and will introduce between each two consecutive points two zero samples as shown in the input scope. As we can expect the ramp becomes distorted. The idea is to recreate the ramp by filling in the missing values. We will do this by designing an interpolator that implements a close approximation of the sinc function. We use a new component that is the delay line and will enter in the Synapse the values that correspond to a sampling of the sinc function.

NeuroSolutions Example

Another example of the power of function approximation is the Fourier series . Fourier series are an example of expansions with trigonometric polynomials. Everybody in engineering has heard of frequency representations (also called the spectrum) because of the following amazing property: any periodic function (even with discontinuities) can be approximated by sums of sinusoids (eventually with infinitely many terms). Moreover, there are simple formulas that allow us to compute the components in the frequency domain from any time signal. Fourier formulas

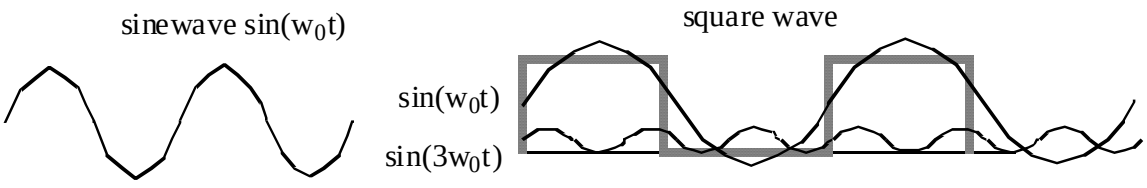


Figure 5. Decomposition by sinewaves

NeuroSolutions 2

5.2 Fourier decomposition

This example is a demonstration of how an addition of sinusoids does in fact produce a waveform that resembles a square wave. In order to compute the

coefficients we have to perform a Fourier series decomposition of the square wave, which is not difficult but is cumbersome and requires an infinite number of sinusoids. By including more and more terms of the Fourier series we make the composite waveform closer and closer to the square.

NeuroSolutions Example

Still another example is the wavelets. One of the problems with the Fourier decomposition is that the sinewaves have infinite extent in time, i.e. they exist for all time. In many practical problems one would like to decompose signals that have a finite extent (transients) in which case the Fourier analysis is not very efficient. Wavelets provide such a decomposition for transients. The idea is to choose a wave shape that is appropriate to represent the signal of interest (the mother wavelet), and create many translations and scales (also called dilation) such that one can reconstruct the desired signal.

The wavelet expansion uses a two parameter decomposition

$$\hat{f}(x,w) = \sum_i \sum_j w_{i,j} \varphi_{i,j}(x)$$

Equation 9

where the $\varphi_{i,j}(x)$ are the wavelet bases. The interesting thing is that the bases are obtained from a single function (the mother wavelet $\varphi(x)$) by the operations of scaling and translation,

$$\varphi_{i,j}(x) = 2^{j/2} \varphi(2^j x - i)$$

Equation 10

hence the two indices. Figure 6 shows the scaling and translation operations.

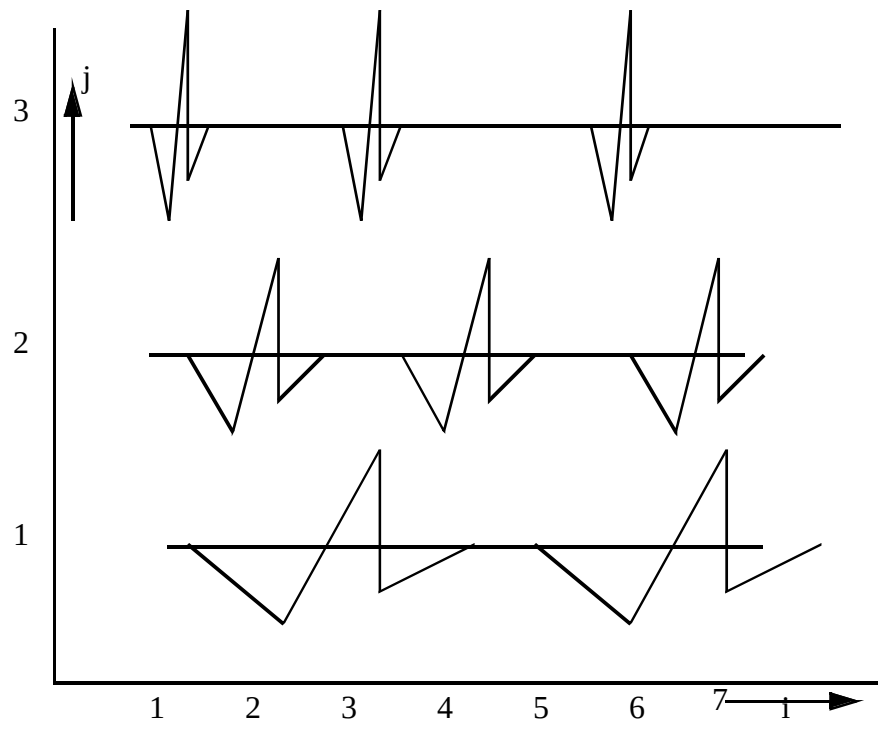


Figure 6. Translation and scaling for a wavelet

All the above methods construct arbitrary functions by weighting the contributions of predetermined elementary functions (sinewaves of different frequencies, translation of sincs or the dilation-translation of the mother wavelet). What varies are the weights in the decomposition. Normally there are close formula solutions to compute the weighting from the signal. In neurocomputing the problem is more complicated for two reasons: first, we want to find the coefficients through adaptation instead of through analytic formulas as in the Fourier case; second, because the basis themselves are dependent upon the data and the coefficients (adapted bases).

In some situations the basis can be chosen naturally from the type of problem being investigated, as in linear systems theory .

3.2. Bases for linear function approximation

When the function in Figure 1 is linear the ideas of linear regression explained in Chapter I can be immediately applied to construct an approximation.

5.3 Linear regression

Here we will show that through adaptation we can find the coefficients of a very simple linear transformation between x and d of Figure 1. The transformation is simply

d=2x+3

We will see that for this case a linear system constructed from a Synapse and a BiasAxon can solve the problem very easily. This is simply linear regression we studied in Chapter I. We will create the transformation by applying one of the function generators to the input of the system and using another function generator at the output producing the same wave shape but with twice the amplitude and with a bias of 3. Then we will let the system adapt using the LMS rule. We can see that very quickly the system finds the relationship and the synaptic weight becomes 2 and the bias becomes 3.

NeuroSolutions Example

However, there is a preferred choice for elementary functions when *f(x)* is linear with constant coefficients. Linear system theory shows that the natural bases are the complex exponentials e^{sx} because they are complete for square integrable functions and they are the eigenfunctions of linear shift-invariant operators. eigendecomposition .

The implication of this fact is thoroughly explored in linear systems, which are networks that implement a signal decomposition using complex exponentials. We will use eigendecompositions in Chapter IX when we study adaptive filters. But here we just would like to remark that eigendecompositions are the most efficient since we are constructing a function from its **"**elementary pieces**"** so the reconstruction error can be made equal to zero with small number of bases. Sometimes other considerations such as easy of implementation may overshadow the use of complex exponentials.

3.3. Bases for nonlinear system approximation - The MLP network

When the function *f(x)* in Figure 1 is nonlinear there is in general no natural choice of

basis. Many have been attempted such as the Volterra expansions , the splines , and the polynomials. Weierstrass proved that polynomials are universal approximators. Weierstrass Theorem The problem is that either many terms are necessary or the approximations are not very well behaved. One of our requirements is that the basis have to be powerful and easy to work with.

In neurocomputing there are two basic choices for the elementary functions that build the approximant $\hat{f}$ (**x**,**w**), which are called *local and global elementary functions*. An elementary function is global when it responds to the full input space, while local elementary functions respond primarily to a limited area of the input space. Going back to Figure 2, it is easy to link the operation of function approximation to a neural topology, in this case to a one hidden layer perceptron with a linear output, where $\varphi_i(\boldsymbol{x})$ is

$$\varphi_i(x) = \sigma\left(\sum_k a_{ik} x_k + b_i\right)$$

Equation 11

and σ is one of the sigmoid nonlinearities (logistic or tanh). The system output is given by $y = \sum_i w_i \varphi_i$. Note that the first layer weights are denoted by a_{ik} and they change the value of $\varphi_i(\boldsymbol{x})$. So, the one hidden layer MLP with a linear output PE can be thought of as an implementation of a system for function approximation (Eq. 1), where the bases are exactly the outputs of the hidden PEs. Note that the sigmoid PE responds to the full input space x with a non zero value (1, -1 (or 0), or intermediate values) so the MLP implements an approximation with global elementary functions.

The interpretation is that the MLP is performing function approximation with a set of *adaptive bases* that are determined from the input-output data. This means that the bases are not predefined as in the sinc, wavelet, or Fourier analysis, *but depend upon the first layer weights and on the input*. In this respect the MLP is much closer to the function approximation implemented by some linear systems. So function approximation

with adaptive bases is slightly different from the previous picture we gave (Figure 3). First, because the bases change with the data. This means that the projection manifold is changing with the data. Second, because the weights in the network have different functions. The input layer weights change the bases by orienting the manifold, while the output layer weights find the best projection within the manifold. Training will find the set of weights (a_{ij}) that best orient the manifold (first layer weights) and that determine the best projection (w_{ij}). *Therefore the training is more difficult because not only the projection but also the basis are changing*. However we can obtain much more compact representations.

This view should be compared with what we described in Chapter III about MLPs for classification. Each PE in the hidden layer creates a discriminant function with a shape defined by the PE nonlinearity with an orientation and position defined by the first layer weights. So the views agree but in function approximation the PEs are less prone to saturate. Due to the highly connected topology and the global nature of the elementary functions, good fitting is obtained with reasonably few bases (i.e. few PEs). However, the training is difficult because the basis functions are far from orthogonal. multi-hidden-layer MLPs

In terms of function approximation the one layer MLP is deciding the orientation, where to place, and what is the relative amplitude of a set of multidimensional sigmoid functions (one per PE). This function decomposition resembles the approximation obtained with step functions well known in linear systems (Figure 7).

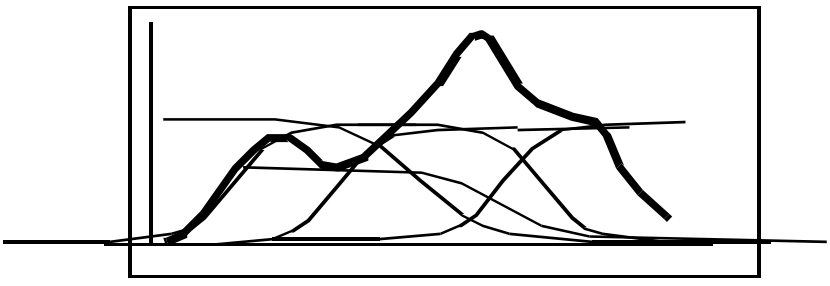
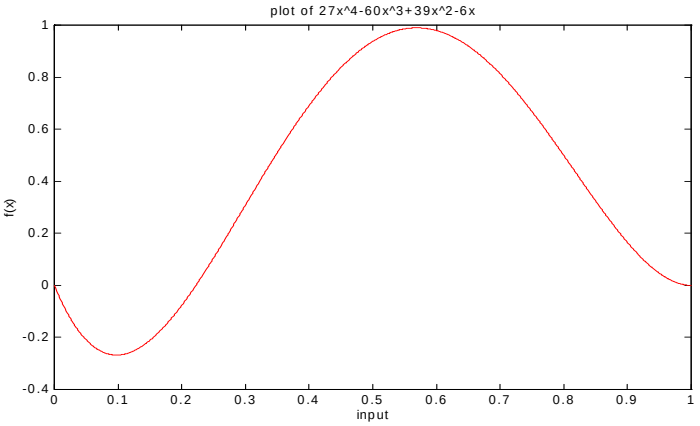


Figure 7. Function approximation with logistic functions.

NeuroSolutions 4

5.4 Function approximation with the MLP

Starting here, many of the examples will be studying the function approximation abilities of various networks. To illustrate this point, we have chosen a fourth order polynomial to try to approximate. The polynomial was chosen to give an interesting shape over the input range of 0..1 and has the equation 27x4 - 60x3 + 39x2 - 6x. The graph of the polynomial from 0..1 is:



In this example we will use an MLP with a linear output to approximate the above function. In our case the MLP will approximate the function with tanh bases (the hidden layer PEs are tanh). These elementary functions are stretched and moved over the range and then summed together to approximate the polynomial.

NeuroSolutions Example

MLPs are universal approximators as we stated in Chapter III. The proof is based on an extension of the Weierstrass theorem outline of proof . But as we stated above, another important characteristic is to study how the error decreases with the order or/and the dimension of the problem. The importance of MLPs for function approximation was recently reinforced by the work of Barron . He showed that the *asymptotic accuracy of the approximation with MLPs is independent of the dimension of the input space*. This is unlike the approximation with polynomials where the error convergence rate is a function of the number of dimensions of the input (the error decreases exponentially slower with

the dimension of the input space). *This means that MLPs become much more efficient than polynomials for approximating functions in high dimensional spaces.* The better approximation properties of MLPs explain why MLPs are more efficient than other methodologies for classification, and why they are key players in identification of nonlinear systems as we will see in Chapter X and XI.

What changes when we use a MLP for function approximation and for classification? The obvious answer is to look at the output PE and say that for function approximation the output is linear while for classification the output PE must be also nonlinear. In fact, we can use also a nonlinear PE for function approximation if we carefully set the dynamic range of the output. So the difference is not solely in the output PE, but also in the nature of the problem. In function approximation the operating point of the hidden PEs is normally far away from saturation since the mappings tend to be smooth. In classification, where the outputs are 1,0, the operating point of the hidden PEs is normally driven to saturation. This is easily observed when we use a square wave as the desired signal, because this choice implements exactly an indicator function.

NeuroSolutions 5

5.5 MLP to approximate a squarewave (classification)

In this example we use an MLP with a tanh output to approximate a square wave. Notice, that since a square wave is either on or off, this function approximation problem is identical to a classification problem. Thus, classification is a subset of function approximation with the desired signal having on/off characteristics. The important point to show here is that when doing classification, the PEs become saturated and the weights increase greatly. This allows the tanh or logistic function to approximate the on/off characteristics of the desired signal. Thus for classification, the MLP tends to operate in the saturated regions of the hidden PEs (on/off) while for general function approximation the hidden PEs tend to operate in the linear region.

NeuroSolutions Example

3.4. Alternate basis for nonlinear systems - the RBF network

In neurocomputing, the other popular choice for elementary functions is the *radial basis functions* (RBFs), where $\varphi_i(\mathbf{x})$ becomes

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

Equation 12

where $\gamma(.)$ is normally a Gaussian function

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

Equation 13

with variance σ^2. Notice that the Gaussian is centered at $\mathbf{x}_i$ with variance σ^2, so its maximum response is concentrated in the neighborhood of the input $\mathbf{x}_i$, falling off exponentially with the square of the distance. The Gaussians are then an example of local elementary functions. If we plug Eq. 13 in Eq. 1 we obtain the following implementation for the approximant to the function f(**x**)

$$\hat{f}(\mathbf{x}, \mathbf{w}) = \sum_i w_i G\left(\left|\mathbf{x} - \mathbf{x}_i\right|\right)$$

Equation 14

which implements the input-output map of the RBF network.

Let us think of an arbitrary function and of a set of localized windows (of the Gaussian shape). Function approximation in a limited area of the input space requires (see Figure 8): the placement of the localized windows to cover the space; the control of the window width; and a way to set the window amplitude (the height). So it is plausible that in fact we can approximate arbitrary continuous functions with a RBF network. approximation properties of RBF

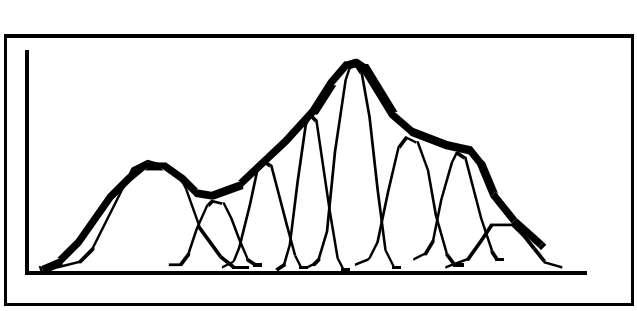


Figure 8. Approximation by RBFs in 1-D.

NeuroSolutions 6

5.6 Function approximation with RBFs

Now we will solve the same polynomial approximation problem with a Radial Basis Function. We can vary the number of RBFs and see how it affects the power of the network to approximate the given polynomial.

NeuroSolutions Example

Go to Next Section

4. Probabilistic Interpretation of the mappings-Nonlinear regression

So far we have assumed a deterministic framework to study the input-output mapping. It enhances our understanding to look now at the mappings discovered by MLPs and RBFs from a statistical perspective. The result we will enunciate below is valid as long as the mean square error (MSE) criterion is utilized in the training.

We will assume that the input data is a random variable *x*, and the desired response *t* is also a random variable, not necessarily Gaussian distributed. The topology is a MLP or a RBF with a linear output PE as we have been discussing. The important result is the following: a network with weights obtained by minimizing the MSE has an output which approximates the conditional average of the desired response data tk, i.e. the regression of *t* conditioned on *x*

$$y_k(x, w^*) = << t_k \mid x >>$$ **Equation 15**

where w* means the optimal weights, and <<.>> refers to the conditional average defined by

$$<< t_k \mid x >>= \int t_k \, p(t_k \mid x) dt_k$$ **Equation 16**

derivation of the conditional average

. So the MLP and RBF networks are effectively nonlinear regressors, extending the adaline for cases where the input-output map is nonlinear. They will be able to **"**discover **"** any deterministic input-output relationship corrupted by additive zero-mean noise, since the network output will approximate the average of the desired response. The only requirements are that the network has converged to the global minimum, that the number of degrees of freedom in the network topology is large enough and that there is enough data to train the system. These are non trivial issues but we have learned ways to cope with them in Chapter III and IV.

NeuroSolutions 7

5.7 Nonlinear regressors

We will illustrate this important point by creating a nonlinear mapping problem corrupted by additive noise. We again use the polynomial approximation case and add noise to the desired signal. Since the network output can be thought of as the average of d with respect to the distribution p(d|xi) at a given point xi of the domain, the network should clean the noise and produce the polynomial. This clearly shows that the MLP is doing regression but now with nonlinear mappings. You can also use the RBF to produce the same result, since it is due to the use of the MSE criterion, and it is independent of the topology.

NeuroSolutions Example

Go to next section

5. Training Neural Networks for Function Approximation

5.1. Training MLPs for function approximation

The second problem that needs to be solved in applying neural networks for function approximation is a procedure to automatically find the coefficients from the data. Notice that the backpropagation algorithm studied in Chapter III solves exactly this problem. In fact, straight backpropagation minimizes the error power between the desired response and the system output (the L_2 norm). This algorithm is totally transparent to the fact that in function approximation we have a linear output and we use the absolute value of the error instead of the error power. In fact we saw how to integrate backpropagation with arbitrary norms in Chapter IV, so we can use backpropagation with the L_1 norm to exactly solve Eq. 2 . L1 versus L2

NeuroSolutions 8

5.8 MLPs for function approximation with L1 norm

We again show the MLP network approximating the function of Example 4 except that this time the L1 criterion is utilized. In theory, this should produce a better fit to the data but may train slower.

NeuroSolutions Example

5.2. Adapting the Centers and variances of Gaussians in RBFs

Backpropagation can be applied to arbitrary topologies made up of smooth nonlinearities, so it can train also the newly introduced RBFs. However, there are other procedures to adapt RBF networks that are worth describing. One simple (but sometimes wasteful in classification) approach to assign the Gaussians is simply to uniformly distribute their centers in the input space. This was the method used in Example 3. Although this may be a reasonable idea for approximation of complicated functions that cover the full input space, it is not recommended in cases where the data clusters in certain areas of the input space. There are basically two ways to select the positioning and width of the Gaussians in RBFs: the supervised method and using self-organization.

The supervised method is a simple extension of the backpropagation idea for the RBF network. In fact the Gaussian is a differentiable function, so errors can be

backpropagated through it to adapt μ and σ in the same way as done for tanh or sigmoid nonlinearities. The backpropagation algorithm can theoretically be used to *simultaneously* adapt the centers, the variance and the weights of RBF networks. The problem is that the method may provide suboptimal solutions due to local minima (the optimization is nonlinear for the centers and variances).local minima for Gaussian adaptation

The self-organizing idea is very different. It divides the training phase in the independent adaptation of the first layer (i.e. the location and width of the Gaussians), followed by a second step that only adapts the output weights in a supervised mode keeping the first layer frozen. The idea is appealing because it treats the adaptation of the centers and variances as a resource allocation step that does not require external labels. This means that only the input data is required in this step. Since the training of the hidden layer is the most time consuming with gradient methods, the self-organizing method is more efficient.

The clusters of the data samples in the input space should work as attractors for the Gaussian centers. If there is data in an area of the space, the system needs to allocate resources to represent the data cluster. The variances can also be estimated to cover the input data distribution given the number of Gaussians available. This reasoning means that there is no need for supervised learning at this stage. The shortcoming is that a good coverage of the input data distribution does not necessarily mean that a good classification will result.

Once the centers and variances are determined, then the simple LMS algorithm presented in Chapter I (or the analytic method of the least squares) can be utilized to adapt the output weights since the adaptation problem is linear in the weights. So let us see what are the algorithms to adapt the centers of the Gaussians and their variances.

Gaussian centers

The goal is to place the Gaussians centered on data clusters. There are many well known algorithms to accomplish this task (see Haykin). Here we will only address the K-means and its on-line implementation, the competitive learning algorithm.

In K-means the goal is to divide the N input samples into K clusters. The clusters are defined by their centers c_i. First a random data assignment is made, and then the goal is to partition the data in sets S_i to minimize the Euclidean distance between the data partition N_i and the cluster centers c_i, i.e.

$$J = \sum_{i=1} \sum_{n \in S_i} \left| x_n - c_i \right|$$

Equation 17

where the data centers c_i are defined by

$$c_i = \frac{1}{N_i} \sum_{n \in S_i} x_n$$

Equation 18

K means clustering requires a batch operation where the samples are moved from cluster to cluster such as to minimize Eq. 17 . An on-line version of this algorithm starts by asking which center is closest to the current pattern x_n. The center that is closest, denoted by c^*_j, wins the competition and it is simply incrementally moved towards the present pattern x_n, i.e.

$$\Delta c^*_j = \eta \left(x_n - c^*_j \right)$$

Equation 19

where η is a step size parameter. We recommend that an annealing schedule be incorporated in the step size. The c* are the weights of the layer preceding the RBFs. This method will be fully described in Chapter VII.

Variance computation

In order to set the variance, the distance to the neighboring centers have to be estimated. The idea is to set the variance of the Gaussian to be a fraction (¼) of the distance among clusters. The simplest procedure is to estimate the distance to the closest cluster,

$$\sigma^2_i = \left\| w_{ij} - w_{kj} \right\|^2$$

Equation 20

where w_{kj} represent the weights of the k^{th} PE which is closest to the i^{th} PE. In general the distances to more neighbors (P) provides a more reliable estimate in high

dimensional spaces so the expression becomes

$$\sigma_i^2 = \frac{1}{P}\sum_{k=1}^{P}\left\|w_{ij} - w_{kj}\right\|^2$$ **Equation 21**

where the P nearest neighbors to the i^{th} PE are chosen.

NeuroSolutions 9

5.9 Training RBFs for classification

We will train a RBF network using the competitive learning approach. We will use a new Synapse called the Competitive Synapse, which will cluster the centers of the RBFs where most of the data resides. Notice that the GaussianAxon will be "cracked" meaning that the dataflow is interrupted. This is done because there is no point to adapt the top layer weights until the centers are placed over the input data. The controller enables full selection of the number of iterations to adapt the centers using competitive learning.

NeuroSolutions Example

Go to next section

6. How to select the number of bases

The selection of the number of bases is rather important. If not enough bases are used, then the approximation suffers throughout the domain. At first one might think that for better approximation more and more bases are needed, but in fact this is not so. In particular if the bases are orthogonal, more bases mean that the network has the potential to represent a larger and larger space. If the data does not fill the input space but is corrupted by white noise (white noise always fills the available space), then the network starts to represent also the noise which is wasteful and provides sub-optimal results. Let us illustrate this with NeuroSolutions.

NeuroSolutions 10

5.10 Overfitting

This example demonstrates that when the data is noisy too many basis will distort the underlying noiseless input-output relationship. We will use a RBF to approximate the polynomial. But instead of doing it in the noiseless case as before we are going to add random noise to the desired signal. We then will change the number of basis and the width of the Gaussians. We will see that for larger networks the noise becomes more apparent. We will also see that if the network doesn't have enough degrees of freedom, then the approximation is also not good.

NeuroSolutions Example

Experience shows that the problem is not just one of the pure size of the network, but the values of the coefficients are also very important. So learning, complicates the matter of selecting the number of bases. Effectively, this is the same problem that was encountered in selecting the size of the MLP for classification. Here we will revisit the problem presenting a statistical view, and then offering two approaches to deal with it: penalizing the training error, and using regularization. Although this problem was already briefly treated in Chapter IV here we will provide a more precise view of the problem and will relate the findings with the previous techniques.

6.1. The bias-variance dilemma

The optimal size of a learning machine can be framed as a compromise between bias and variance of a model. We will address this view fully in the next section, so here we will just motivate the arguments with a simple analogy. Let us use polynomial curve fitting to exemplify the problem faced by the learning machine. A polynomial of order N can exactly pass through N+1 points, so when a polynomial fits a set of points (fiducial points) two things can happen. If the polynomial degree is smaller than the number of points, the fitting will be bad (*model bias*) because there are not enough degrees of freedom to pass the polynomial through every point (left panel of Figure 9). So errors will exist all over the domain. For example, the linear regressor, which is a first order polynomial, will produce errors at nearly every point of a quadratic curve (second order polynomial). On the other extreme, if the order of the polynomial is much larger than the number of

fiducial points, the polynomial can exactly pass through every point. The problem is that the polynomial was not constrained for the other points in the domain and thus its values can oscillate widely between the fiducial points (*model variance*) as illustrate in the right panel of Figure 9. The best solution is to find an *intermediate polynomial order that will provide low bias and low variance* across the domain.

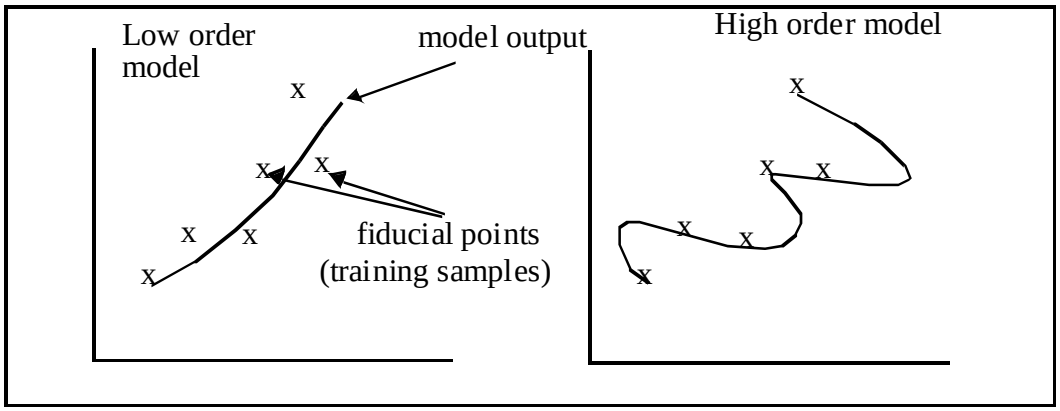


Figure 9. Under and over fitting polynomials to a set of points

This simple example provides a conceptual framework for the problem of generalization encountered in learning machines.

- The fiducial points are the training samples.
- The full domain represents all the possible test data that the learning machine will encounter.
- The polynomial becomes the input-output functional map created by the learning machine.
- The learning machine weights are equivalent to the coefficients of the polynomial.
- The size of the polynomial is the number of weights.

Therefore, we can see that for a good fit all over the domain, both the size of the network as well as the amount of training data are relevant.

The *model bias* is the error across the full data set, which can be approximated to a first degree by the error in the training set. Given a set of training samples the learning machine will try to approximate them (minimize training set classification error). If the complexity of the machine is low (few parameters) the error in the training set is high, and

performance in the test set will also suffer, meaning that the separation surfaces have not been properly placed (Figure 9a). If the machine complexity is increased, the training set error will decrease, showing a smaller model bias.

Too large a model produces an exact fit to the training set samples (*memorization of the training samples*) but may also produce large errors in the test set. The source of this test set error for larger machines (Figure 9b) differs from the small machine case. It is produced by the model variance, i.e. using parameters fine tuned for a specific subset of samples (training samples) that do not **"**generalize**"** to a different set of samples*. This is the reason the committees presented in Chapter IV which basically reduce the variance through weighted averaging improved the test set performance*.

The difference in performance between the training and the test set is a practical measure of the model variance. We can always expect that the error in the test set will be larger than in the training set. However, a large performance difference between the training and test sets should be a *red flag* indicating that *learning and/or model selection was not successful*.

This argument means that the *goal of learning should not be a zero error in the training set*. It also clearly indicates that information from both the training and test sets must be utilized to set appropriately a compromise between model bias and variance. This is the reasoning why in Chapter IV we presented crossvalidation as the best way to stop the training of a learning machine, since crossvalidation brings the information from the unseen samples to stop training at the point where the best generalization occurs.

6.2. The bias-variance dilemma treated mathematically

The problem of generalization can be studied mathematically in a statistical framework by interpreting the network as a regressor and decomposing the output error into its bias and variance.

A measure of how close the output is to the desired response is given by

$$\left(y-d\right)^2$$

Equation 22

But note that this error depends on the training set utilized. To remove this dependence we average over the training sets (TS) to yield

$$J_{TS}\left[\left(y-d\right)^2\right]$$

Equation 23

Now rewrite the expression inside the square brackets as

$$\left(y-d\right)^2=\left[y-J_{TS}\left(y\right)+J_{TS}\left(y\right)-d\right]^2$$

Equation 24

When we compute the expected value we obtain

$$J_{TS}\left[\left(y-d\right)^2\right]=J_{TS}\left[J_{TS}\left(y\right)-d\right]^2+J_{TS}\left\{\left[y-J_{TS}\left(y\right)\right]^2\right\}$$

$$bias^2 \qquad \text{var}\,iance$$

Equation 25

The first term is the (square of the) bias of the model because it measures how much in the average the output differs from the desired response. The second term is the variance because it measures how much each particular output y differs from its mean across the training sets.

Now let us assume that we add noise to the desired response, i.e.

$$d=f+\varepsilon$$

Equation 26

where *f* is the true input-output map and ε is the noise. One extreme is the case that the model is so small that the output is not dependent at all on the variability of the data (no free parameters, just an a priori chosen function *g*). So the model bias may be large (if the function *g* we picked is not the true function *f*), but the model variance is zero since *y* is the same across all training sets.

The other extreme is to have the model with so many parameters that it passes exactly by every training data point. In this case the first term which measures the model bias is zero, but the second term which measures the model variance is the power of the noise ε. A good size model is the one where both the model bias and variance are small. This

view however, does not tell us how to select the size of the model, but illustrates well what is going on.

6.3. Penalizing the training error

The problem is to find a general criterion to determine the model order for the problem under consideration. Generalization can also be formulated in this way. Many theories have addressed this issue. One that we would like to mention is Rissanen's minimum description length (MDL) criterion because it is central to extracting models from data (or composing complex functions from simpler ones).

The previous explanation shows that one can not look only at the fitting error as the criterion of optimality. We have to counterbalance it with the number of degrees of freedom of the model. Rissanen presented this idea very intuitively in terms of code lengths.

Our data can be thought as having an intrinsic code length in the following way: We may try to describe the data using a code we define. So the data set requires a certain number of bits to be described (Figure 10). If the data is random noise then every sample needs to be used to describe the data and the code length is the same as the data length. But the data may have been created by a linear system for which two numbers (slope and bias) are sufficient to describe all the samples.

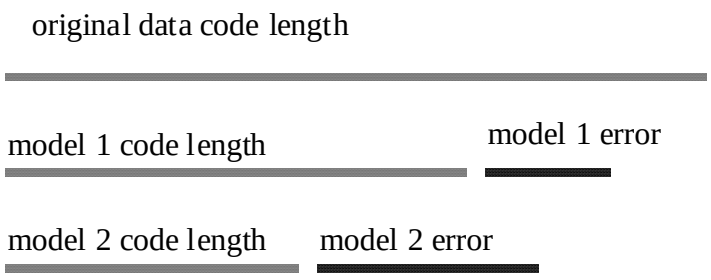


Figure 10. Code lengths of data and several models

When we model the data we are effectively describing it in a different way, by the topology and parameters of our model, and also by the fitting error. If we add the error to the model output then we again describe the original data exactly. Consider that we have

to assign bits to codify the error, C(Ei), and also to represent the parameters of our model, C(Mi). So the description of the data using a particular model *i* is

$$C(M_i, E_i) = C(M_i) + C(E_i)$$

The most efficient description of the data set is the one that minimizes the overall code length, i.e. the total number of bits to represent both the error and the model parameters (Figure 10),

$$\min_i \quad C(M_i, E_i)$$

Notice that this is a very interesting idea, because it couples the complexity (size) of the machine with the size of the fitting error. If we use small number of parameters, then the error will be larger, and we utilize many bits to represent the error. On the other hand, if we use too large a machine we use too many bits to describe the machine, although only a few are needed to represent the error*. The best compromise in terms of code length lies in the middle of smaller machines and manageable errors*. MDL and Bayesian theory

Possibly the simplest implementation of this idea is to penalize the mean square error obtained in the training by including a term that increases with the size of the model, as was first proposed by Akaike . Akaike's information criterion (AIC) reads

$$\min_k \quad AIC(k) = N \quad \ln J(k) + 2k$$

Equation 27

where *J(k)* is the MSE in the training set, *k* is the number of free parameters of the model, and *N* is the number of data samples. AIC has been extensively used in model based spectral analysis (Kay). This expression shows that even if the error decreases with the size of the model *k*, there is a linear penalty with *k* so the minimum value is obtained at some intermediate value of model order *k* (Figure 11).

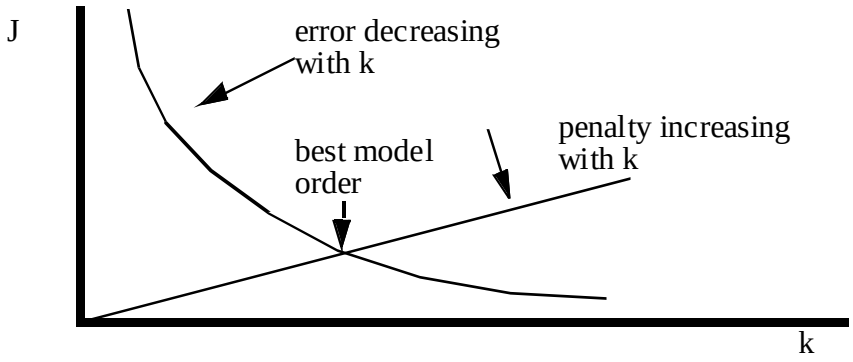


Figure 11. Best model order according to Akaike's criterion.

Notice that in Akaike's criterion the penalty is independent of the number of samples of the training data. According to Rissanen MDL criterion, a larger penalty for larger data sets can be obtained by substituting 2k by k/2ln(N) yielding

$$\min_k MDL(k) = N \ln J(k) + \frac{k}{2}\ln(N)$$

Equation 28

The appeal of these equations is that they allow us to use all the available data for training (unlike crossvalidation) and can be easily computed for practical applications since they only require the mean square error in the training set for a collection of models of different sizes (k). Akaike's method works well for one layer systems (in particular linear). However, it becomes less acccurate for multilayer networks since the size of the model it is not univocally related to the number of weights.

It is also important to relate this method with the early stopping criterion that we established in Chapter IV using crossvalidation. Remember that we stopped training based on the performance in the validation set. The early stopping criterion measures directly some type of distance between the model and the data. We can choose the best model by utilizing different model sizes (k) and pick the one that provides the minimum error in the crossvalidation set, i.e.

$$\min_k \quad J_{val}(k)$$

Equation 29

It has been shown that this use of crossvalidation is asymptotically equivalent to Akaike's criterion. In neural networks, these equations have to be interpreted in an approximate sense, in particular for multilayer architectures. In fact the role of the PEs and their weights is very different so it is not enough to naively count the number of free parameters. The principle of structural Risk Minimization and the VCdimension is the most principled way to select the best size model. We will address it shortly.

NeuroSolutions 11

5.11 Akaike's criterion for RBFs

This example demonstrates Akaike's criterion for selecting the number of PEs in the RBF network. We have added a few DLLs to the breadboard, one of which computes Akaike's criterion. The other(s) change the widths and centers of the RBFs to automatically span the input range [0,1] based upon the number of hidden PEs. Thus, you can change the number of RBFs and run the network to see what the final Akaike's criterion value will be.

NeuroSolutions Example

6.4. Regularization

Regularization theory was proposed by Tikhonov to deal with ill-posed problems . As an example, the equation **xA**=**y** is said ill-conditioned when a slight modification $\Delta\mathbf{y}$ due to noise in a dependent variable **y** produces an enormous change in the solution for **x**. One way to solve this type of problem is to minimize the residue

$$R(\mathbf{x}) = \left|\mathbf{Ax} - \left(\mathbf{y} + \Delta\mathbf{y}\right)\right|^2 \qquad \textbf{Equation 30}$$

Tikhonov proposed to stabilize the solutions to such problems by adding a regularizing function $\Gamma(\mathbf{x})$ to the solution

$$R(\mathbf{x}) = \left|\mathbf{Ax} - \left(\mathbf{y} + \Delta\mathbf{y}\right)\right|^2 + \lambda\Gamma(\mathbf{x}) \qquad \textbf{Equation 31}$$

and was able to show that when $\Delta\mathbf{y}$ approaches 0 the solution approaches the true value

$\mathbf{yA}^{-1}$. λ is a small constant called the *regularization constant*, and the regularizing function is a non-negative function that includes some a priori information to help the solution. Normally these regularizers impose smooth constraints, i.e. they impose limits on the variability of the solution. Inverse problems in general are ill-posed.

When one deals with the determination of the complexity of a learning machine with information restricted to the training set, the problem is ill-posed because we do not have access to the performance in the test set. The basic idea of regularization theory is to add an extra term to the cost function such that the optimization problem becomes more constrained, i.e

$$J_{new} = J_c + \lambda J_r \qquad \textbf{Equation 32}$$

where J_c is the cost function, J_r is the regularizer and λ is a parameter that weights the influence of the regularizer versus the cost. Tikhonov regularizers penalize the curvature of the original solution, i.e. they seek smoother solutions to the optimization problem. If we recall the training algorithms, we should choose regularizers for which derivatives with respect to the weights are efficiently computed. One such regularizer is

$$J_r = \sum_n \left(\frac{\partial^2 y_n}{\partial x_n^2} \right)^2 \qquad \textbf{Equation 33}$$

which penalizes large values of the second derivative of the input-output map. There is evidence that even first order penalty works in practice. The value of λ must be experimentally selected.

Regularization is closely related to the optimal brain damage (which uses the Hessian to compute saliencies) and to the weight decay ideas to eliminate weights. In fact, weight decay (Eq. 16 in Chapter IV) is equivalent to a regularization term that is a function of the L_2 norm of the weights (Gaussian prior), i.e.

$$J_{new} = J_c + \lambda \sum_i w^2{}_i \qquad \textbf{Equation 34}$$

The square in Eq. 34 can be substituted by the absolute value to obtain a L_1 norm of the weight yielding Eq. 19 in Chapter IV (Laplacian prior).

It is interesting to compare Eq. 34 with Eq. 27 . Both are effectively creating a new cost function that penalizes large models. However the principles utilized to derive both expressions are very different. This analogy suggests that the determination of the regularization constant λ is critical to find the best possible model order. Too large a value for λ will choose networks that are smaller than the optimum, while too small λ will yield too large networks. Moreover, we can relate these choices to the bias and variance of the model. We can expect that large λ will produce smooth models (too large a bias), while too small λ will produce models with large variance. The best value of the regularization constant can be computed from statistical arguments Wahba . Let us experimentally verify these statements.

NeuroSolutions 12

5.12 Weight-decay to prune RBFs

As discussed above, the weight decay DLL which we introduced in Chapter IV can be used to implement the regularization discussed above. We will use the same RBF breadboard and set the number of hidden PEs to 20. Then, using weight decay on the output synapse, we can dynamically "turn off" unnecessary PEs by driving their output weights to zero. By adjusting the decay parameter of the weight decay algorithm, we can produce smoother or more exact outputs from the network.

NeuroSolutions Example

Go to next section

7. Applications of Radial Basis Functions

7.1. Radial Basis functions for Classification

Going back to Eq. 1 and 2, let us interpret them for a classification problem. In classification f(**x**) becomes the indicator function {-1 (or 0),1}. So what this equation says is that one can construct arbitrary complex discriminant function in the input space by constructing linear discriminant functions in an auxiliary space (the space of the elementary functions) of large dimension which is nonlinearly related (by $\varphi(\mathbf{x})$) to the input space x. This is a counter intuitive result that was first proved by Cover (Cover Theorem) and is associated with the fact that in sufficiently high dimensional spaces data is always sparse, i.e. the data clusters are always far apart. So it is always possible to use hyperplanes to separate them. The problem is that one needs to determine many parameters.

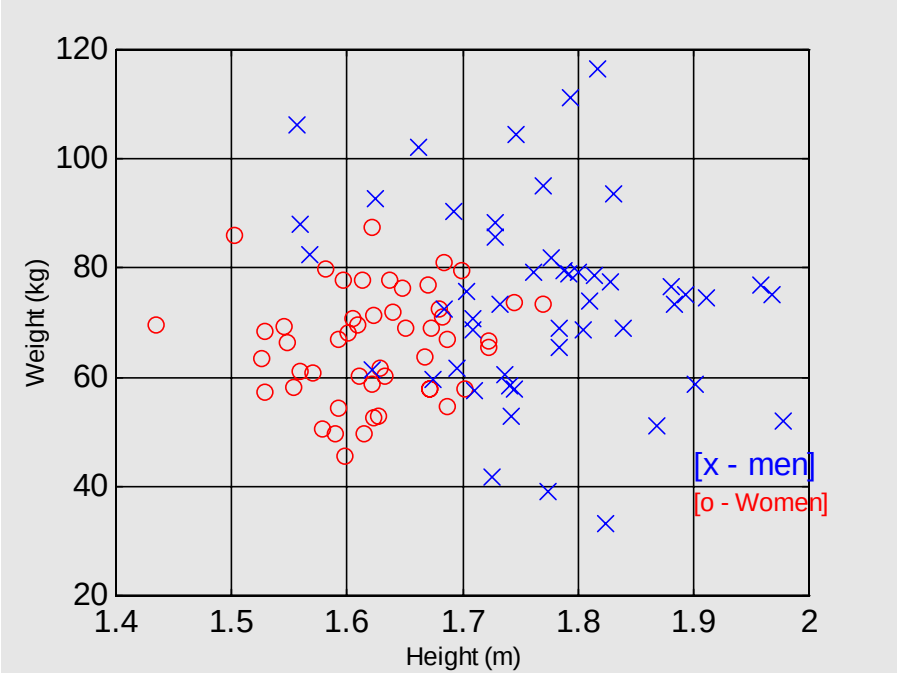
Radial basis function networks implement directly this idea by using Gaussian functions to project the input space to an intermediary space where the classification is done with an hyperplane implemented as the weighted sum of the Gaussian PE outputs. This result can be understood if we focus on the output of each Gaussian. No matter how intertwined the classes are, if the centers of the Gaussians and its radius (the variance) is properly controlled, each Gaussian can always be made to respond to a single classes. The obvious limit is to assign a Gaussian to each sample. But generally this is not necessary.

One can assign a Gaussian to a sub-cluster of one of the classes. Then the classification is made by linearly combining the responses of each one of the Gaussians such that it is +1 for one class and -1 (or 0) for the other. From this discussion, one sees that there are two fundamental steps in designing RBFs for classification: the first is the placement and the selection of the radius for each Gaussian. The second is the weighting of the individual Gaussian responses to obtain the desired classification. It would also be convenient to assign a radius that would change with direction, which extends the contours of the Gaussians from circles to ellipses.

NeuroSolutions 13

5.13 MLPs and RBFs for classification

This example uses RBFs to do classification. We are repeating the problem from chapter 1 where we have height and weight data on various people and are trying to classify whether they are male or female.



Remember that we cannot get perfect classification. For reference, we have included a link to the MLP which solves this problem.

NeuroSolutions Example (MLP)

Now run the RBF network and see how the classifier performs at the same level but the discrimant functions are different (her they are more curved). You should change the number of RBF PEs and see how the discriminant plot and confusion matrix are affected.

NeuroSolutions Example (RBF)

7.2. Radial Basis Functions as Regularizers

Radial basis functions can be derived mathematically from the theory of Tikhonov regularizers Poggio . Interestingly, when the solution of Eq. 32 is carried out using

calculus of variations, radial basis functions appear as the natural solution for regularization (for rotationally and translation invariant kernels). So this means that instead of using directly the data collected from the outside world, there is advantages in first fitting one radial basis function to each data point, and work with their outputs. In a sense the RBFs are interpolating the collected data, but the method is too cumbersome for large data sets (and the variances must be experimentally determined).

7.3. Radial Basis Functions as Regressors - The probabilistic Neural Network

We can also utilize radial basis functions to estimate a regression function from noisy data following the ideas of kernel regression. In kernel regression we seek to estimate the probability density function p(x, d) of the input-desired pairs (x_i, d_i) using the Parzen window method (which is a nonparametric method). Parzen window method . One can show that the regression of the target data yields RBF as kernel regression

$$y(x) = \frac{\sum_i d_i \exp\left\{-\|x - x_i\|^2 / \left(2\sigma^2\right)\right\}}{\sum \exp\left\{-\|x - x_i\|^2 / \left(2\sigma^2\right)\right\}}$$

Equation 35

where σ is the width of the Gaussian and has to be experimentally determined. Basically the method places a Gaussian in each sample multiplied by the desired response d_i and normalized by the response in the input space. This network is called in neural network circles the *probabilistic neural network* and can be easily implemented using RBF networks.

NeuroSolutions 14

5.14 Density estimation with RBFs

In this example we are going to train a normalized radial basis function network according to Eq. 35 to show how the network handles probability density function approximation. We have a few samples in the input space that belong to two classes, and will train a probabilistic neural network to solve the problem. We select the number of RBFs equal to the number of samples. During training notice

that the RBF centers converge to the input data, and that the output of the net provides the conditional average of the target data conditioned on each input. Change the variance of the RBFs to see how they affect the estimates for the targets as given by the output weights.

NeuroSolutions Example

Go to the next section

8. Support Vector Machines

Support vector machines (SVMs) are a radically different type of classifiers that have attracted lately lots of attention due to the novelty of the concepts that they brought to pattern recognition, to their strong mathematical foundation, and also due to the excellent results in practical problems. We already covered in Chapter II and Chapter III two of the motivating concepts behind SVMs, namely: the idea that transforming the data into a high dimensional spaces makes linear discriminant functions practical; and the idea of large margin classifiers discussed to train the perceptron. Here we will couple these two concepts and create the Support Vector Machine. We refer to Vapnik's books for a full treatment.

Let us go back to the concept of kernel machines. We saw in Chapter II that the advantage of a kernel machine is that its capacity (number of degrees of freedom) is decoupled from the size of the input space. By going into a sufficiently large feature space, patterns become basically linearly separable and so a simple perceptron in feature space can do the classification. In this chapter we have discussed the RBF network, which can be considered a kernel classifier. In fact, the RBF places Gaussian kernels over the data and linearly weights their outputs to create the system output. So it conforms exactly with the notion of kernel machine presented in Chapter II, Figure 12. When used as an SVM, the RBF network places a Gaussian in each data sample, such

that the feature space becomes as large as the number of samples.

But an SVM is much more than an RBF. In order to train a RBF network as a SVM we will utilize the idea of large margin classifiers discussed in Chapter III. There we presented the Adatron algorithm which only works with perceptrons. Training an RBF for large margin will at the same time decouple the capacity of the classifier from the input space and also provides good generalization. We can not get better than this in our road to powerful classifiers. We will extend the Adatron algorithm here in two ways: we will apply it to kernel based classifiers such as RBFs and we will extend the training for non-linearly separable patterns.

8.1 Extension of the Adatron to Kernel Machines

Recall that the Adatron algorithm was able to adapt the perceptron with maximal margin. The idea was to work with data dependent representations, which lead to a very simple on-line algorithm to adapt the multipliers.

We will write the discriminant function of the RBF in terms of the data dependent representation, i.e.

$$g(x) = \sum_{k=1}^{L} w_k G_k(x, \sigma^2) + b = < \sum_{i=0}^{N} \alpha_i G(x, \sigma^2) \cdot G(x_i, \sigma^2) > + b = \sum_{i=0}^{N} \alpha_i G(x - x_i, 2\sigma^2) + b$$

Equation 36

where $G(x, \sigma^2)$ represents a Gaussian function, L is the number of PEs in the RBF, wl are the weights, N is the number of samples, αi are a set of multipliers one for each sample, and we consider the input space augmented by one dimension with a constant value of 1 to provide the bias. Notice that for the special case of the Gaussian the inner-product of Gaussians is still a Gaussian. The kernel function (the Gaussian) is first projecting the inputs (x, xi) onto a high dimensional space, and then computing an inner product there. The amazing thing is that the Gaussian kernel avoids the explicit computation of the pattern projections into the high dimensional space, as shown in Eq. 36 (the inner product of Gaussians is still a Gaussian). Any other symmetric function that

obeys the Mercer condition has the same properties. This topology is depicted in Figure 12, where we can easily see that it is a RBF, but where each Gaussian is centered at each sample, and the weights are the multipliers αi.

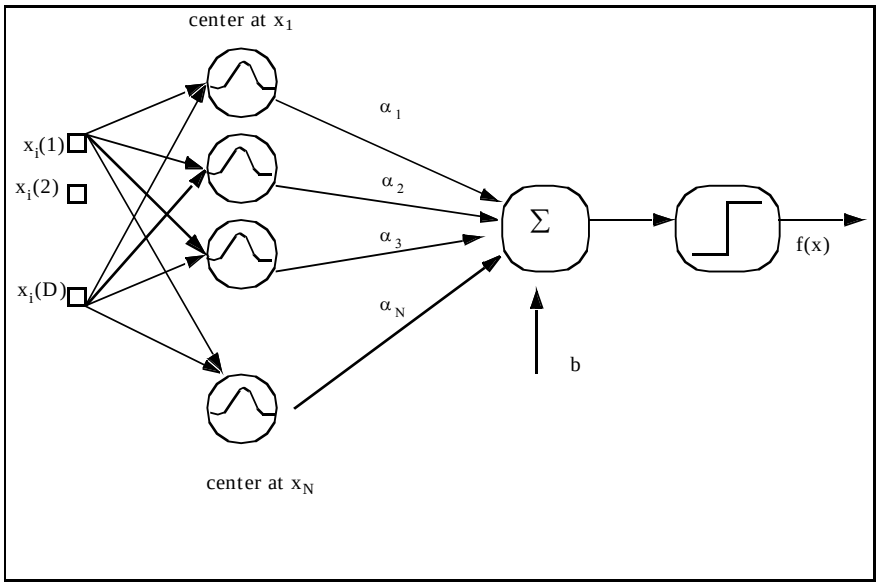


Figure 12. Topology of the SVM machine with RBF kernels

The Adatron algorithm can be easily extended to the RBF network by substituting the inner product of patterns in the input space by the kernel function, leading to the following quadratic optimization problem

$$J(\alpha) = \sum_{i=1}^{N} \alpha_i - \frac{1}{2} \sum_{i=1}^{N} \sum_{j=1}^{N} \alpha_i \alpha_j d_i d_j G(x_i - x_j, 2\sigma^2)$$

$$subject \quad to \qquad \sum_{i=1}^{N} d_i \alpha_i = 0 \qquad \alpha_i \geq 0, \forall i \in \{1,...N\}$$

Equation 37

Following the same procedure as in Chapter III we can define

$$g(x_i) = d_i \left(\sum_{j=1}^{N} d_j \alpha_j G(x_i - x_j, 2\sigma^2) + b \right) \qquad and \qquad M = \min_i g(x_i)$$

and choose a common starting multiplier (e.g. αi=0.1), learning rate η, and a small threshold (e.g., t = 0.01).

Then, while M>t, choose a pattern xi, and calculate an update $\Delta\alpha_i = \eta(1 - g(x_i))$ and perform the update

$$\begin{cases} \alpha(n+1) = \alpha(n) + \Delta\alpha(n), \quad b(n+1) = b(n) + d(n)\Delta\alpha(n) & \alpha(n) + \Delta\alpha(n) > 0 \\ \alpha(n+1) = \alpha(n) \quad , \quad b(n+1) = b(n) & \alpha(n) + \Delta\alpha(n) \le 0 \end{cases}$$

After adaptation only some of the αi are different from zero (called the support vectors). They correspond to the samples that are closest to the boundary. This algorithm is the kernel Adatron with bias that can adapt an RBF with optimal margin. This algorithm can be considered the **"**on-line**"** version of the quadratic optimization approach utilized for SVMs, and it is guaranteed to find identical solutions as Vapnik**'**s original algorithm for SVMs, Freiss . Notice that it is easy to implement the kernel Adatron algorithm since g(xi) can be computed locally to each multiplier, provided the desired response is available in the input file. In fact the expression for g(xi) resembles the multiplication of an error with an activation, so it can be included in the framework of neural network learning.

So the Adatrom algorithm basically pruned the RBF network of Figure 12 so that its output for testing is given by

$$f(x) = \mathrm{sgn}\Big(\sum_{\substack{i \in \mathrm{sup}\,port \\ vectors}} d_i \alpha_i G(x - x_i, 2\sigma^2) - b\Big)$$

8.2 Extension of the Adatron with Soft Margin

What happens if the patterns are not exactly linearly separable? The idea is to introduce a soft margin using a slack variable $\xi_i \ge 0$, and a function $F(\xi) = \sum_{i=1}^{N} \xi_i$, which will penalize the cost function. We will still minimize the function F, but now subject to the constraints $d_i(w.x_i + b) \ge 1 - \xi_i \qquad i = 1,...N \qquad and \qquad w.w \le c_n$. The new cost function becomes

$$J(\alpha, C) = \sum_{i=1}^{N} \alpha_i - \frac{1}{2C} \sum_{i=1}^{N} \sum_{j=1}^{N} \alpha_i \alpha_j (d_i d_j G(x_i - x_j, 2\sigma^2)) - \frac{c_n C}{2}$$

$$subject \quad to \quad \sum_{i=1}^{N} d_i \alpha_i = 0 \qquad 1 \geq \alpha_i \geq 0, \forall i \in \{1, ...N\} \qquad C \geq 0$$

Normally, instead of computing the optimal C we choose a value *a priori*. C can be regarded as a regularizer. This means that the matrix of kernel inner products is augmented in the diagonal by the factor 1/C, i.e.

$$if \quad i = j \quad \Omega(x_i, x_j) = G(x_i, x_j) + 1/C \quad else \quad \Omega(x_i, x_j) = K(x_i, x_j)$$

The only difference in the algorithm for this case is the calculation of g(xi) which becomes

$g(x_i) = d_i (\sum_{j=1}^{N} d_j \alpha_j \Omega(x_i, x_j) + b)$. As we can see these calculations can be easily implemented as an iterative algorithm, but notice that large data sets produce tremendously large RBF networks (one Gaussian per data sample). Since the input layer has no free parameters, effectively the mapping can be computed once and saved on a big matrix.

SVMs have been applied to numerous problems with excellent results. They consistently are at par with the best reported results, which have taken many years of fine tuning. One of the weaknesses of the method is that it does not control the number of support vectors that solve the problem. In principle SVMs should be sensitive to outliers, even in the case of the slack variables.

8.3. A Summary of the SVM theory

We would like to present in a more principled manner the beautiful theory that gave rise to the SVMs, and show the equivalence to the above algorithms. However, this theory is beyond the scope of an introductory texbook as this one. So we will only highlight the most important concepts that gave rise to this design methodology. We will see how the ad-hoc observations made in Chapter II and III have been formulated mathematically by Vapnik and co-workers.

Learning theory can be framed as a functional approximation problem in spaces with a probability measure. The goal is to approximate a function *d=f(**x**)* where *f(.)* is a fixed but unknown conditional distribution function *F(d|**x**).* The approximant is a learning machine that implements a set of functions $\hat{f}(x,w)$ where the parameters ***w*** are to be determined through learning. The inputs ***x*** are random vectors with a fixed but also unknown probability distribution function *F(x).* The selection of the parameters ***w*** is done through a finite number M of input output observations (x_i,d_i) which are independent and identically distributed (i.i.d.).

You should be able to see how similar this is to the block diagram of Figure 1. Now we are saying that what links the desired response to the input is a conditional distribution function, which is unknown but fixed. The machine should discover this function by the repeated presentation of a finite set of exemplars which are assumed i.i.d. In learning theory, the best parameters ***w**** are chosen to minimize the *risk functional*

$$R(w) = \int L(d, f(x,w))dF(x,d) \qquad \textbf{Equation 38}$$

where *L(d,f(x,w))*, the *loss function*, measures the discrepancy between the desired response *y* and the learning machine output. However, we can not compute this integral since we do not know the joint distribution *F(x,d)*, but we have the finite number of observation pairs (x_i,d_i). So we will substitute Eq. 38 by

$$R_{emp}(w) = \frac{1}{N}\sum_{i=1}^{N} L(d, f(x_i, w)) \qquad \textbf{Equation 39}$$

which is called the empiric risk, and we will minimize this quantity instead. This method of solving the risk problem is called empirical risk minimization (ERM) principle. Notice that ERM is a principle based on induction. We may think that this substitution of the risk functional by the empiric risk would constraint the possible cases that could be solved. But it turns out that Glivenko and Cantelli proved that the empiric distribution function converges to the actual distribution and Kolmogorov even proved that the empirical

distribution function has asymptotic exponential rate of convergence. This is the basis for statistical inference.

The two problems treated so far in Chapter I and III, the regression problem and the classification problem are special cases of this formulation. In fact it is enough to define the loss function as

$$L(d, f(x,w)) = \left[d - \hat{f}(x,w)\right]^2 \qquad \textbf{Equation 40}$$

to obtain the formulation of the regression, provided the output *y* is a real value and if one assumes that the class of functions $\hat{f}(x,w)$ includes the regression function we are seeking.

If the output *y* takes the integer values d={0,1} and if the function $\hat{f}(x,w)$ is the set of indicator functions, i.e. functions that take only two values -zero and one-, if the loss function is defined as

$$L\left(d, \hat{f}(x,w)\right) = \begin{matrix} 0 & iff & d = f(x,w) \\ 1 & iff & d \neq f(x,w) \end{matrix} \qquad \textbf{Equation 41}$$

then the risk functional computes the probability of an error in the classification.

One could even show that this same formalism can provide as a special case density estimation over the class of functions p(x,w) if

$$L\left(p(x,w)\right) = -\log p(x,w) \qquad \textbf{Equation 42}$$

Learning theory provides the most general way to think about training adaptive systems. The theory addresses mathematically the problem of generalization that is vital to neurocomputing. Vapnik establishes four fundamental questions for learning machines:

- What are the necessary and sufficient conditions for consistency of a learning process.
- How fast is the rate of convergence to the solution.
- How to control the generalization ability of the learning machine.

- How to construct algorithms that implement these pre-requisites.

  We will restrict ourselves to the special case of pattern recognition (where the function is an indicator function). To study SVMs we need to address basically the last two bullets, but first provide the definition of VC (Vapnik-Chervonenkis) dimension. One of the fundamental problems in pattern recognition has always been the estimation of the Bayes error. There is no known procedure to directly minimize the Bayes error, because it involves the integration over the tails of the pdfs, which are unknown (and the multidimensional integral is not trivial either). Our procedure of designing classifiers by minimizing the training error (which in this theory corresponds to the empiric risk) is not appropriate as we have discussed in Chapter IV and in this Chapter. All the methods we discussed to control the generalization error are in fact indirect (and sometimes not principled). So researchers have tried to find methods that minimize an upperbound of the Bayes error. It is in this framework that Vapnik's contributions should be placed. Vapnik argues that the necessary and sufficient conditions of consistency (generalization) of the ERM principle depend on the capacity of the set of functions implemented by the learning machine. He has shown that the VC dimension is an upperbound for the Bayes error.

  The VC dimension *h* of a set of functions is defined as the maximum number of vectors that can be separated into two classes in all 2*h* possible ways using functions of the set. For the case of linear functions in n dimensional space, the VC dimension is *h*=n+1. So the VC dimension is a more principled way to measure the capacity of a learning machine which we discussed in Chapter II. For general topologies the VC dimension is not easy to determine, but the trend is that larger topologies will correspond to larger VC dimension.

  The VC dimension of a learning machine appears as a fundamental parameter to determine its generalization ability. In fact Vapnik proved that the generalization ability (the risk R) of a learning machine $Q(x,\alpha)$ of size k parametrized by α is bounded by

$$R(\alpha_k) \leq R_{emp}(\alpha_k) + \Phi(\frac{N}{h})$$ **Equation 43**

where $R_{emp}(\alpha)$ is the empirical risk (the error measured in the training set) and the second term is a confidence interval. So the generalization ability depends upon the training error, the number of observations and the VC dimension of the learning machine. There are basically two ways to handle the design.

The first is to design a learning machine with a given topology, which will have a given VC dimension (that needs to be estimated). This is the conventional neural network design. Once this is done, Eq.43 tells us all. We train the ANN and this gives us an estimate of the empirical risk, but also a confidence interval. Eq. 43 describes the bias variance dilemma very precisely. In order to decrease the training set error, we may have to go to large ANNs which will provide a large confidence interval, i.e. the test error may be much larger than the training error. We say that the machine memorized the training data. So the problem becomes one of trading-off training set error and small VC dimension, which is handled heuristically by the size of the learning machine. This compromise was thought intrinsic in inductive inference, going back to the famous Occam razor principle (the simplest explanation is the best).

The second approach is called the structural risk minimization (SRM) principle and gives rise to the support vector machines (SVMs). The principle specifies to keep the empirical risk fixed (at zero if possible) and minimize the confidence interval. Since the confidence interval depends inversely on the VC dimension, this principle is equivalent to searching for the machine that has the smallest VC dimension. Notice that there is no compromise in the SRM principle. It states that the best strategy is to use a machine with the smallest VC dimension. Another point to make is that VC dimension and number of free parameters are two very different things, unlike the indications from Akaike and Rissanen work. We now know that we can apply very large machines to small data sets and still be able to generalize due to capacity control. So this SRM approach has profound implications in the design and use of classifiers. Let us now see how we can implement

SRM in practice.

Here the concept of hyperplanes and margin becomes critical. Although the VC dimension of the set of hyperplanes in n dimensions is n+1, it can be less for a subset. In fact Vapnik proved that the optimal hyperplane (the smallest norm) provides the smallest confidence interval. *So the problem in SRM is one of designing a large margin classifier*. Let us briefly describe here Vapnik's formulation to allow us a comparison with our previous approaches.

Assume we have a set of data samples

$$S = \{(x_1, d_1),....., (x_N, d_N)\}, \qquad d_i \in \{-1,1\}$$

What we want is to find the hyperplane *y = w.x+b* with the smallest norm of coefficients $\|w\|^2$ (largest margin). To find this hyperplane we can solve the following quadratic programming problem: minimize the functional

$$\Phi(x) = \frac{1}{2}(x.x)$$

under the constraint of inequality

$$d_i[(x_i.w) + b] \geq 1 \qquad i = 1,2,...N$$

where the operation is an inner product. The solution to this optimization is given by the saddle points of the Lagrangian

$$L(w,b,\alpha) = \frac{1}{2}(w.w) - \sum_{i=1}^{N} \alpha_i \{[(x.w) + b]d_i - 1\} \qquad \textbf{Equation 44}$$

By using the dual formulation, we can rewrite Eq. 44 as

$$J(\alpha) = \sum_{i=1}^{N} \alpha_i - \frac{1}{2}\sum_{i=1}^{N}\sum_{j=1}^{N} \alpha_i \alpha_j d_i d_j (x_i.x_j) \qquad subject \quad to \qquad \alpha_i \geq 0, \forall i \in \{1,...N\}$$

under the constraint $\sum_{i=1}^{N} \alpha_i y_i = 0$. The solution is a set of α^*. We can show that only some of the samples will correspond to Lagrangian multipliers different from zero, and will be called the support vectors. They are the ones that control the positioning of the optimal hyperplane. So the large margin classifier will be specified by

$$f(x) = sign\left(\sum_{\substack{\text{sup } port \\ vectors}} d_i \alpha_i^* (x_i . x) - b^* \right)$$

Equation 45

One of the characteristics of the SVM is that the user has no control on the number of support vectors, i.e. the size of the final machine. During training all the RBFs are used, but once the SVM is trained the RBF should be trimmed, discarding the RBFs that are not support vectors. The number of support vectors is dependent upon the data, which makes sense but practically it is not a useful feature. The expressions we arrived are exactly the same as the one for the Adatron algorithm we discussed in Chapter III. Except that Vapnik suggests a quadratic programming solution, while the Adatron is an "on-line" solution, easily implementable in neural network software. As any on-line algorithm, the Adatron requires the control of learning rate and suffers from the problem of misadjustment and stoping criterion. We can expect that training SVMs with large data sets demands a lot from computer resources (memory or computation).

Now we have a better understanding of why optimal margins are good for classification. SVMs can also be used for regression and density estimation.

Go to the next section

9. Project: Applications of Neural Networks as Function Approximators

In Chapter IV we have seen how neural networks can be applied to classification. Here we would like to show how the same topologies can be applied as function approximators

(nonlinear regressors) in a wealth of practical applications. We selected one application in the financial industry, and another in real state. The goal is to discover nonlinear mappings between input variables and important outcomes. In the financial arena the outcome is to predict the value of the S&P 500 using several financial indicators, while in the real state application, the problem is to estimate the price of a house based on several indicators. We will see that neural networks provide a very powerful analysis tool.

Prediction of S&P 500

This example will develop a very simple model for predicting the S&P 500 one week in advance. You can use this demo as a starting point for developing your own more complex financial models. The inputs to the model consist of the 1 year Treasury Bill Yield, the earnings per share and dividend per share for the S&P 500, and the current week's S&P 500. The desired output is the next week's S&P 500. The data has been stored in the file "Financial Data". There are 507 weeks worth of data which cover approximately a ten year period.

The data has been pre-sampled such that the first 254 exemplars contain the data for weeks 1, 3, 5,…, 505, 507 and the last 253 exemplars contain the data for weeks 2, 4, 6,..., 504, 506. The first 254 exemplars will be used for training and the last 253 exemplars will be used for evaluating the trained networks performance.

NeuroSolutions 15

5.15 Prediction of SP 500

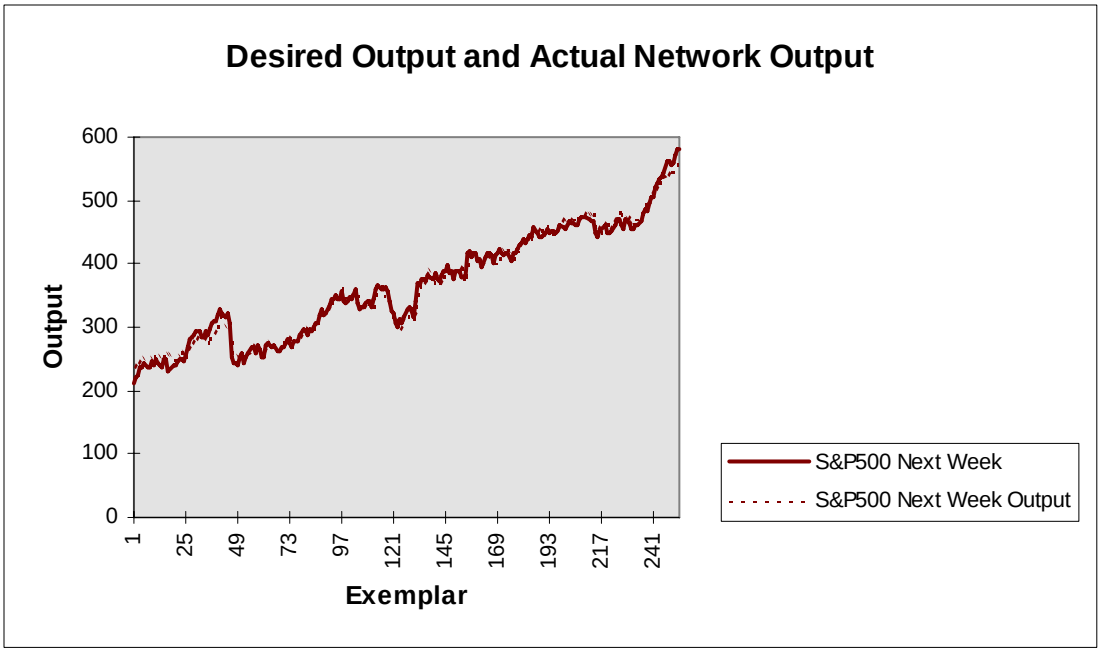
We will use a simple one hidden layer MLP to model this data. The network has 4 inputs and the desired response is the next week value of the S&P 500. The topology has to be carefully developed as we exemplified in Chapter IV. We recommend that weight decay be utilized to avoid overfitting. Alternatively, Akaike's criterion should be used to find the best model order as we did in example 11. Let us train the network until the error stabilizes.

The next step is to verify the performance of the network in the unseen data. In the figure below we show the performance of the network we trained. The solid line is

the actual value while the network output is dashed. As we can see the network fits rather well the actual value of the S&P 500 with the 4 inputs selected. We can compute the correlation coefficient between the actual and predicted curves to have normalized (but linear) measure of performance.

We suggest that you try RBFs for the same problem and compare performance. This can be the embrio of a financial model, but remember that predicting the value of the stock is just one of many factors needed to come up with an investment strategy.

NeuroSolutions Example



Estimating the price of a house

The final example that we will like to address in this chapter is how to help decide which inputs are more significant in our application. This is an important issue because in many practical problems we have many different indicators or sensors which may require very large input layers (hence many network weights), and very few exemplars to train the networks. One possibility is to prune the number of inputs without affecting

performance.

We have to understand that this is a compromise. The more variables we have about a problem the better is the theoretical performance, assuming that we have infinite noise free data. One can think that each variable is a dimension to represent our problem, so the higher the number of dimensions the better the representation. But notice that for each extra input the representation problem is posed in a larger dimensionality space, so training the regressor (or the classifier) appropriately requires many more data samples. This is where the compromise comes in. Since we always have finite, noisy data, the fundamental issue is to find the best “projection” to represent our data well.

One approach is to use all the available data to train a neural network and then ask which are the most important inputs for our model. It is obvious that this requires the calculation of the relative importance of each input for the overall result, i.e. the sensitivity of the outcome with respect to each input.

In this example we will develop a model for real estate appraisal in the Boston area. We will use 13 indicators as inputs to this model. These indicators are per capita crime rate by town (CRIM), proportion of residential land zoned for lots over 25,000 sq.ft. (ZN), proportion of non-retail business acres per town (INDUS), bounds Charles River (CHAS), nitric oxides concentration (NOX), average number of rooms per dwelling (RM), proportion of owner-occupied units built prior to 1940 (AGE), weighted distances to five Boston employment centers (DIS), index of accessibility to radial highways (RAD), full-value property-tax rate per $10,000 (TAX), pupil-teacher ratio by town (PTRATIO), 1000(Bk - 0.63)^2 where Bk is the proportion of blacks by town (B), % lower status of the population (LSTAT).

The desired output for this model is the Median value of owner-occupied homes (in $1000’s). Hence this is a mapping problem which we will solve with a MLP (nonlinear regression). There are 400 total samples. Three hundred of them will be used as “Training” and the other 100 as “Testing”. The data is located in the file named “Housing

Data".

The way we can do input sensitivity analysis is to train the network as we normally do and then fix the weights. The next step is to randomly perturb, one at a time, each channel of the input vector around its mean value, while keeping the other inputs at their mean values, and measure the change in the output. The change in the input is normally done by adding a random value of a known variance to each sample and compute the output. The sensitivity for input k is expressed as

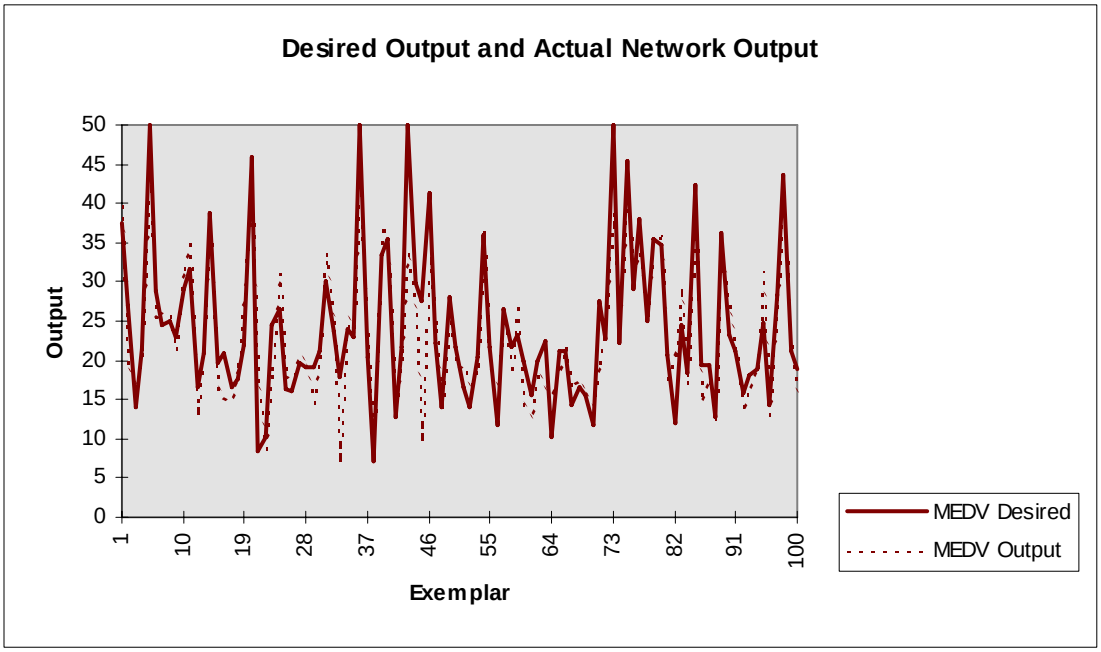
$$S_k = \frac{\sum_{p=1}^{P}\sum_{i=1}^{o}(y_{ip} - \bar{y}_{ip})^2}{\sigma_k^2}$$

where $\bar{y}_{ip}$ is the i^{th} output obtained with the fixed weights for the p^{th} pattern, o is the number of network outputs, P is the number of patterns, and σ_k^2 is the variance of the input perturbation. So this is really easy to compute in the trained network, and effectively measures how much a change in a given input affects the output across the training data set. Inputs that have large sensitivities are the ones that have more importance in the mapping and therefore are the ones we should keep. The inputs with small sensitivities can be discarded. This helps the training (because it decreases the size of the network) and decreases the cost of data collection, and when done right has negligible impact on performance.

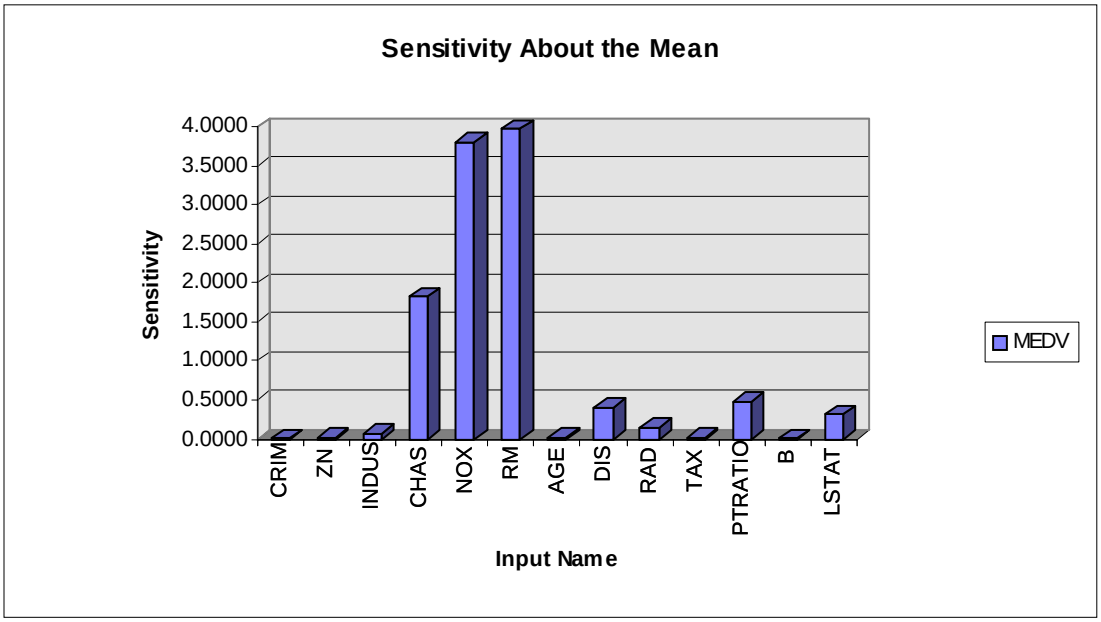
NeuroSolutions 16

5.16 Estimating prices in the Boston housing data

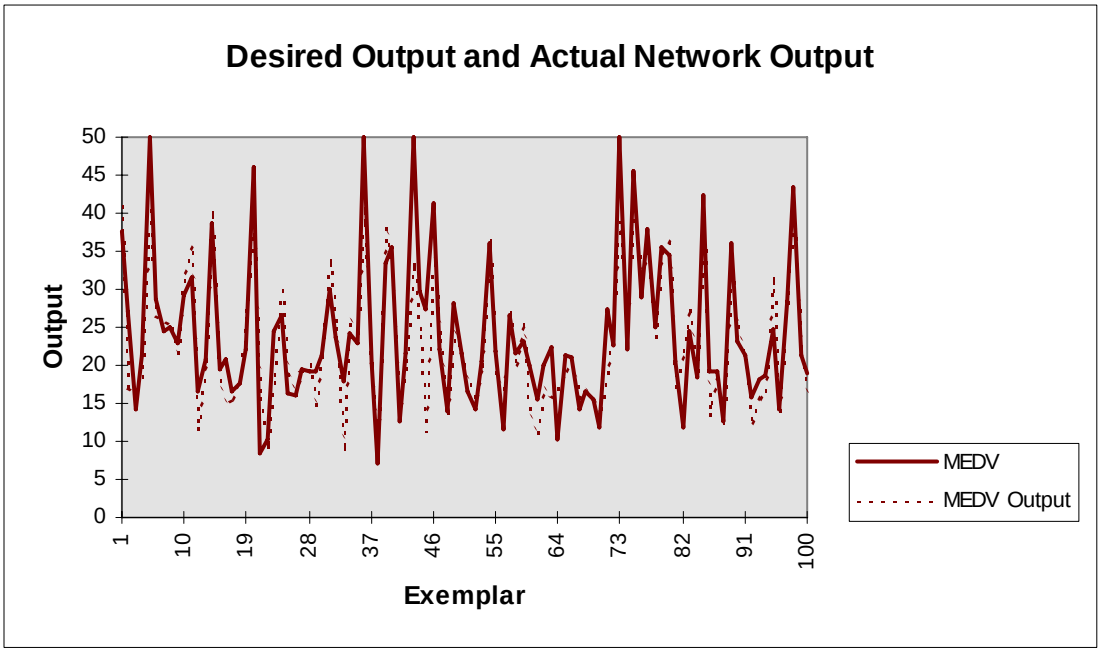
Let us train a one hidden layer MLP initially with 14 inputs and one output. The choice of the number of hidden PEs should be done as before, i.e. starting small and plotting the output MSE for several different runs as a function of the number of PEs. Train the network and run it on the test set to visually observe the performance in the test set. The network produces a very good fit in most cases indicating a successful model.



Then the next step is to run the sensitivity analysis which will estimate the importance of each input to the overall performance. NeuroSolutions has this built in feature. Let us work with the trained network and turn learning off (fix the weights). We have to specify the amount of random noise we want to add to each individual input (which is done automatically one input at a time), while keeping the other inputs at their mean values. NeuroSolutions computes the sensitivity at the output. Let us place a MatrixViewer at the L2 criterion in the sensitivity access point, and write down the values. We should use different values of dither to obtain a reasonable linear approximation to the operating point of the regressor. We can then plot the different values of the sensitivity for each input variable as shown in the Figure below



From the figure we see that there are 5 inputs that display a very low sensitivity so they can be omitted without affecting appreciably the quality of the mapping. Hence, a reduced network with the inputs INDUS, CHAS, NOX, RM, DIS, RAD, PTRATIO, LSTAT shall be trained again. As you can see in the figure below, the matching is basically the same, but now we have a smaller network that will generalize better, and we can reduce the cost of collecting data for this problem.



NeuroSolutions Example

Go to next section

10. Conclusion

In this chapter we provided a view of neural networks as function approximators. This is the more general view of this family of systems and impacts our understanding about their capabilities, establishes new links to alternate methods and provides, we hope, a better understanding of the problems faced in training and using adaptive systems.

One of the interesting things about neurocomputing is that it lies at the intersection of many diverse and complementary theories, so it is a very rich field. The price paid is that the reader was bombarded with many different concepts and since our goal is to keep the text at the introductory level the presentation only addressed the key concepts. Our hope is that the reader was motivated enough to pursue some of these topics.

MLPs are very important for function approximation because they are universal approximators and their approximation properties have remarkably nice properties (the approximation error decays independently of the size of the input space). This may explain why MLPs have been shown to outperform other statistical approaches in classification.

In this chapter we also introduced another class of neural networks, called the radial basis function networks (RBFs). RBFs can be used in the same way as MLPs since they are also universal approximators, i.e. they can be classifiers, regressors or density estimators.

We also presented the basic concepts of the structural risk minimization principle and support vector machines (SVM). This is a difficult theory so we merely highlighted the important implications, which are many. At the top is the paradigm shift from the conventional compromise between generalization and network size, to the strict recipe of

using the smallest capacity machine for best generalization. The innovative ideas contained in the SRM principle will impact tremendously the evolution of the whole field of learning from examples and inductive inference. So you should be alert to follow the developments.

SVMs are practical learning machines that minimize an upper bound to the Bayes error, so they are very useful in pattern recognition. SVMs are very easy to apply to practical problems, provided the user has large computers since they do not have free parameters (just the slack variable for the nonseparable case). The kernel Adatron algorithm allows a simple sample by sample implementation of the quadratic programming required to find the support vectors, and conquers one of the difficulties of the method (having access to quadratic programming software).

For the reader with a background in engineering this chapter provided a view of MLPs and RBFs as implementing function approximation with a new set of bases (the sigmoids which are global or the Gaussians, which are local). For the reader with a statistical background, the chapter provided a new view of generalization. Neural networks belong to the exciting class of nonparametric nonlinear models, which learn directly from the data, and so can be used in experimental science.

NeuroSolutions Examples

5.11 Akaike's criterion for RBFs

5.12 Weight-decay to prune RBFs

5.13 MLPs and RBFs for classification

5.14 Density estimation with RBFs

5.15 Prediction of SP 500

5.16 Estimating prices in the Boston housing data

Concept Maps for Chapter V

Chapter V


Function Approximation with Adaptive Systems 1, 1.1
Probabilistic Interpretation 4
Nonlinear Models Chapter X, XI
The problem 1.1
Projects 9
The Theory 2
Geometric Interpretation 2.1
Support Vector Machines 8
How to Choose Basis 3
How to find projections 5
Examples 3.1
How Many Bases 6
Linear Case 3.2
Resource Allocation 5.2
Backprop 5.1
Bias Variance Dilemma 6.1, 6.2
Nonlinear-Global Bases 3.3
Nonlinear - Local Bases 3.4
Penalty 6.3
MLP 3.3
Radial Basis Functions (RBFs)
Regularization 6.4
Structu Learni (link
Classifiers 7.1
Regularizers 7.2
Regressors 7.3

Go to Next Chapter

Go to the Tables of Content

calculation of the orthonormal weights

Let us assume that the bases $\varphi(x)$ are orthonormal. Let $f(x)$ be any square integrable function . The goal is to find the coefficients w_i such that

$$f(x) = \sum_{i=1}^{N} w_i \varphi_i(x)$$

Taking the inner product of $f(x)$ with $\varphi_i(x)$

$$< f(x), \varphi_1 >= \sum_{i=1}^{N} w_i < \varphi_i, \varphi_1 >= w_0 < \varphi_1, \varphi_1 >$$

since the vectors are orthogonal. Moreover since they are of unit length (orthonormal), we get

$$w_1 =< f(x), \varphi_1 >$$

which corroborates the interpretation that the weight can be thought as the projection in each elementary function. So in general we get the pair of relations

$$\begin{cases} f(x) = \sum_i w_i \varphi_i(x) \\ w_i = \int_D f(x) \varphi_i(x) dx \end{cases}$$

(Note: if the signals are complex, then the coefficients are given by

$$w_i = \int_D f(x) \varphi_i^*(x) dx$$

where * means complex conjugate). On the other hand if we are working with discrete spaces, this pair of equations becomes

$$\begin{cases} f(k) = \sum_{i=1}^{N} w_i \varphi_i(k) \\ w_i = \sum_{k=1}^{N} f(k) \varphi_i(k) \end{cases}$$

(where once again in the second equation the basis has to be the complex conjugate if they are complex). This pair is what we need to know to apply the projection theorem for orthogonal basis and provide the relationships for the Fourier transforms and Fourier series respectively.

Return to Text

sinc decomposition

The formulas derived above can be used to find out exactly what is the decomposition obtained when the basis are *sinc* functions. We would like to write

$$f(x) = \sum_i w_i \varphi_i(x)$$

The bases are

$$\varphi_i(x) = \sin c(x - x_i)$$

Applying now Eq 7 we have that the weights become

$$w_i = \int_D f(x) \sin c(x - x_i) dx = f(x_i)$$

which means that the weights become exactly the value of the function at the point (i.e. the sample value). So this explains figure 4.

Return to text

Fourier formulas

Applying again the pair of formulas of Eq 7 and Eq. 8 we will present the Fourier transform pair. Remember that the Fourier uses as basis the complex exponentials, i.e.

$$\varphi_i(t) = e^{j\frac{2\pi}{T}it}$$

where T is related to the interval where the function *f(t)* is defined and $j = \sqrt{-1}$. The complex exponentials are a set of orthogonal basis. This means that we are going to expand the function *f(t)* as

$$f(t) = \sum_{i=-\infty}^{\infty} w_i e^{j\frac{2\pi}{T}it}$$

In the interval D= [0,T] we can compute the weights as (Eq.7)

$$w_i = \frac{1}{T}\int_0^T f(t)e^{-j\frac{2\pi}{T}it}dt$$

This means that we have formulas to compute the weights so we do not need to use adaptation.

Note that the complex exponential can be expressed as (Euler relation)

$$e^{jwt} = \cos(wt) + j\sin(wt)$$

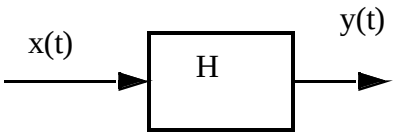
so in fact we are decomposing the signals in sums of sinusoids (but pairs of them).

Return to Text

eigendecomposition

In engineering we use systems to modify time signals according to the user specifications. So a system is a piece of equipment that modifies input signals *x(t)* to produce another

signal *y(t)* (see Figure)



Mathematically we can describe a system by a function *H* operating on the real (or complex) numbers

$$H: \quad x \to Hx$$

and we will call *H* an operator. The output *y* of the system *H* when *x* is applied at the input is

$$y = Hx$$

The response of the system at time t is written *y(t) = [Hx](t).*

A linear system is a system described by the operator *H* which obeys the following properties

$$H\alpha x = \alpha Hx$$
$$H[x + y] = Hx + Hy$$

where α is a scalar. We are normally interested in linear systems that are shift invariant, i.e. where the response does not depend upon the particular instant of application. Let us define another operator *T* which delays *x* by *t* seconds, i.e.

$$x(t - \tau) = [Tx](t)$$

In shift invariant systems, *H* and *T* commute, i.e. *THx = HTx*.

So lets ask the following question. Which are the signals *x(t)* that when applied to a linear time invariant system *H* produce a response that has the same form as the input, apart from a multiplicative factor (gain factor)?

Mathematically this can be written

$Hx = \lambda x$

This is the same problem that gives rise to the eigenvector problem of matrices, and this is the reason why the input *x* that obeys this condition is called an eigenfuntion of *H*.

Linear shift invariant systems have the special property that they commute with the derivative operator *D* (which is a composition of T operators), i.e.

$$If\ y = Hx\ and\ x' = Dx,\ then\ y' = Hx'$$

or in words, if we know the response of the system to an input, and we want to know the response to the derivative of the input, then it is enough to take the derivative of the output.

This is what we need to answer the original question. This property shows that the question is equivalent to finding a signal *x(t)* that is proportional to its derivative, i.e.

$$\frac{dx(t)}{dt} = sx(t)$$

which we know accepts the solution $x(t) = \alpha e^{st}$,i.e. a complex exponential.

What this means is that a linear shift invariant operator *H* when applied to a complex exponential e^{sx} will only change its magnitude and phase, i.e.

$$y = He^{sx} = \alpha e^{sx} \qquad \textbf{Equation 46}$$

where α is a complex number . So, if an arbitrary function *u(x)* is decomposed into exponentials,

$$u(x) = \sum_i w_i e^{s_i x} \qquad \textbf{Equation 47}$$

then the response of *H* to *u(x)* can always be evaluated as a sum of weighted responses to exponentials, i.e.

$$Hu(x) = H\left[\sum_i w_i e^{s_i x}\right] = \sum_i w_i H\left[e^{s_i x}\right] = \sum_i w_i \alpha_i e^{s_i x} \qquad \textbf{Equation 48}$$

where the α_i do not depend of *u(x).* The importance of this equation has to be noted, since it tells us that no matter how complicated the input might be, we always can compute its output by adding the responses to individual exponential components. It also tells us that all we need to know to describe the linear system are the complex numbers α_i.

Fourier analysis is a special case of this decomposition where the complex exponentials have zero real part, i.e. *s = jw* yielding

$$e^{jw_i x} = \cos(w_i x) + j\sin(w_i x)$$

Now we understand the relevance of complex exponentials to study linear systems.

Return to text

Weierstrass Theorem

Weierstrass proved the following important theorem: Let S[a,b] be the space of continuous real valued functions defined in the real segment [a,b]. If $f \in S[a,b]$ then there exists a polynomial $P(x) = \sum_{i=0}^{N} \alpha_i x^i$ with real coefficients α for which

$$|f(x) - P(x)| < \varepsilon$$

for ε>0 and $x \in [a,b]$

In words this says that any function can be approximately arbitrary well (i.e. with an error as small as we want) by a sufficiently large order polynomial.

The Weierstrass theorem is the starting point for most proofs of the universal mapping properties of the MLP.

Return to text

multi-hidden-layer MLPs

The multilayer perceptron architecture is not in the general form of the projection theorem discussed above. As seen in Chapter III the MLP implements an embedding of functions so its approximation properties can not be directly studied with the projection theorem except for the case of the one hidden layer MLP with linear output PE as mentioned above. For this case we can recognize the output of each hidden PE has producing the elementary functions. When the output PE is a logistic function (or tanh) these values are nonlinearly combined, so the projection space is no longer an hyperplane in the input space.

Remember also the properties discussed in Chapter III that the MLP with two hidden layers is an universal approximator, and even with a single hidden layer can approximate any continuous function on a compact set. There are now many theorems that provide proofs depending upon the nonlinearity. So we can conclude that the essence of the power of the approximation is in the topology, not in the specifics of the nonlinearity. As remarked before, these are existence theorems, so the designer still needs to select the topology to actually create the universal approximation properties for the class of functions of interest.

Return to text

outline of proof

We will only outline here the proof for the universal mapping characteristics of the MLP. We start by extending the Weierstrass theorem to metric spaces (the Stone- Weierstrass theorem).

Polynomials can be extended to metric spaces by defining the concept of an algebra. A

family of functions F that map the metric space V to the real line is an algebra if their elements have the properties

$$f_1, f_2 \in F \Rightarrow \alpha f_1 + \beta f_2 \in F$$
$$\textit{and} \qquad f_1 . f_2 \in F$$

where α and β are real numbers.

The Stone Weierstrass theorem can be enunciated in the following way. Let V be a metric space and F an algebra that maps V into the reals. If there is a function $f \in F$ for which $f(v_1) \neq f(v_2) \qquad for\, v_1 \neq v_2$ and $f(v) \neq 0$ in V, then F is dense in the mapping of V into the reals. The idea of dense is the same as arbitrary close approximation as stated in the Weierstrass theorem.

This theorem has been used to show the universal mapping capabilities of the MLP. In fact, the function f(v) can be expanded in a special type of "Fourier series" with squashing cosine functions, i.e.

$$f(v) = \sum_{i=1}^{N} \alpha_i \cos_t(\beta_i v + b_i)$$

where

$$\cos_t(v) = \begin{cases} 1 & v \geq \pi/2 \\ 0.5[1 + \cos(v + 3\pi/2) & -\pi/2 < v < \pi/2 \\ 0 & v \leq -\pi/2 \end{cases}$$

The nonlinearity of the MLP belongs to this family of squashing functions. Notice that *f(v)* is exactly the output of the one hidden layer MLP with a cosine nonlinearity.

Return to text

local minima for Gaussian adaptation

We should be able after Chapter III and IV to understand the difficulty of using

backpropagation to adapt the centers and variances of the Gaussians. With this method, the centers (and variances) are moved in the input space by virtue of the gradient. However, with the RBF, both the local activity and the local error are attenuated by the shape of the Gaussian kernel, while in the MLP only the error was attenuated by the derivative of the sigmoid. The net effect is that training becomes very slow and the chances of getting stuck in local minima are large. Another reported problem is that during adaptation the variances of the Gaussians can become very broad and the RBF looses its local nature.

Return to Text

approximation properties of RBF

The formulation of function approximation using the projection theorem (Eq. 1) can be directly applied to study the approximation properties of the RBF network. Sandberg showed that the RBF network is in fact a general function approximator. This is an existence theorem, so it is up to the designer to choose the number, localize, set the variance and the weighting of Gaussians to achieve an error as small as required.

Using again the Stone Weierstrass theorem, they showed that the RBFs were dense in the mapping from a metric space V to the real line. This is not difficult because the RBFs create an algebra and they do not vanish in V.

The RBF and the MLP achieve the property of universal approximation with different characteristics since the basis functions are very different. In the RBF the basis are local, so each can be changed without disturbing the approximation of the net in other areas of the space. But one needs exponentially more RBF to cover high dimensional spaces (the curse of dimensionality). In the MLP this is not the case as we mentioned in the result by Barron. As we saw in Chapter III, changing one MLP weight has the potential to produce drastic changes in the overall input-output map. This has advantages in some aspects such as more efficient use of PEs, but also disadvantages since the training becomes

slower and the adaptation can be caught in local minima.

RBFs train very efficiently once the centers are determined, since the error is linear in the weights. This fact also guarantees the convergence for the global minimum (if the centers are optimally set). This makes RBFs very useful for system identification.

From the theoretical point of view of function approximation, RBFs possess the property of *best approximation* as defined by Chebyshev, unlike the MLP (i.e. there is always a RBF that provides the minimum error for a given function to be approximated).

Return to text

MDL and Bayesian theory

There are some technical details in implementing this idea of measuring errors by code lengths, but they have been worked out in Information theory. See Rissanen .

Another interesting link is to look at model selection from the point of view of Bayes theory. The probability of a model given the data can be computed using Bayes rule, i.e.

$$P(M_i|D) = \frac{P(D|M_i)P(M_i)}{P(D)}$$

We can forget about P(D) since it is common to all the models. So the most probable model will maximize the numerator. We know that the maximization is not affected if we take the log (since the log is a monotonic function), i.e.

$$\max_i \; [\log(P(D|M_i) + \log(P(M_i))]$$

This expression is very similar to the result obtained using Rissanen's idea. In fact we can interpret Rissanen description length as the sum of the error and the complexity of the model. Now the minimum amount of information required to transmit a message x is given by $-\ln p(x)$. If p(x) is the correct distribution for the message x (our model),

then it will correspond to the smallest message length for a given error. The error in turn can be interpreted as the conditional probability of the data given the model. So we can say that the description length can be expressed as

$$MDL = -\log(p(D \mid M)) - \log(p(M))$$

Since the maximization is equivalent to the minimization of the negative we get the minimal code lengths for the data and the model. See Zemel for a complete treatment.

Return to Text

derivation of the conditional average

This results can be demonstrated (see Bishop for a full treatment) if we write the MSE for the case of large number of patterns as an integral (t is the desired response)

$$J = \frac{1}{2}\sum_k \iint [y_k(x,w) - t_k]^2 \, p(d_k, x) dt_k dx$$

Note that the index k sums over the targets, and the sum over the data exemplars was transformed in the integral, which has to be written as a function of the joint probability of the desired response and the input. This joint probability can be factored in the product of the input pdf p(x) and the conditional of the target data given the input p(tk|x).

The square can be written

$$\left(y_k(x,w) - t_k\right)^2 = \left(y_k(x,w) - << t_k \mid x >> + << t_k \mid x >> - t_k\right)^2$$

where <<tk|x>> is the conditional average given by Eq. 14. We can write further

$$\left(y_k(x,w) - t_k\right)^2 = \left(y_k(x,w) - < t_k \mid x >\right)^2 + 2(y_k(x,w) - < t_k \mid x >)(< t_k \mid x > - t_k) + (< t_k \mid x > - t_k)^2$$

Now if we substitute back into the MSE equation we obtain

$$J = \frac{1}{2}\sum_k \int \left(y_k(x,w) - << t_k \mid x >>\right)^2 p(x)dx + \frac{1}{2}\sum_k \int (<< t_k^2 \mid x >> - << t_k \mid x >>^2)\, p(x)dx$$

The second term of this expression is independent of the network, so will not change during training. The minimum of the first term is obtained when the weights produce

$$y_k(x, w^*) = << t_k \mid x >>)$$

since the integrand is always positive. This is the result presented in the text.

Return to Text

Parzen window method

The Parzen window method is a nonparametric density estimation method widely used in statistics. It is related to learning because it provides a way to estimate the pdf of the data, and unlike the maximum likelihood method it can be applied to a wide range of functions.

In the Parzen window method we start by choosing a symmetrical and unimodal kernel function

$$K(x, x_i, \beta) = \frac{1}{\beta^n} K(\frac{x - x_i}{\beta})$$

and construct the pdf estimator

$$p(x) = \frac{1}{M}\sum_{i=1}^{M} K(x, x_i, \beta)$$

Normally used kernels are the Gaussian, the rectangular, and the spectral windows (Tukey, Hanning, Hamming). The Parzen estimator is consistent and its asymptotic rate of convergence is optimal for smooth densities. But it requires large number of samples to provide good results.

Return to text

RBF as kernel regression

Here the windows are multidimensional Gaussian functions (as utilized in RBF networks) that quantify the joint data distribution in the input-desired signal space, given by

$$p(x,t) = \frac{1}{N}\sum_{i=1}^{N} \frac{1}{\left(2\pi\sigma^2\right)^{\frac{t+c}{2}}} \exp(-\frac{\left\|x - x_i\right\|}{2\sigma^2} - \frac{\left\|t - t_i\right\|}{2\sigma^2})$$

As we discussed in section 3.4.1, regression can be thought as estimating the condition average of the target data t_i conditional to the input x_i, i.e. $<t_i|x_i>$. When the MSE is used the output of the network approaches this value. The conditional average can be written as a function of the pdf which yields

$$y(x) = << t \mid x >> = \frac{\int tp(x,t)dt}{\int p(x,t)dt}$$

This yields Eq. 33 in the text. In general we can utilize fewer Gaussians as done in the RBF network, yielding

$$y(x) = \frac{\sum_i P(i)\theta_i \exp(-\frac{\left\|x - \mu_i\right\|^2}{2\sigma^2})}{\sum_i P(i) \exp(-\frac{\left\|x - \mu_i\right\|^2}{2\sigma^2})}$$

where θ are the centers of the Gaussians in the desired space.

Return to text

L1 versus L2

There are minor differences between the two norms that have been used in function approximation. Eq. 2 utilizes the absolute value of the error which is commonly called uniform approximation. So strictly speaking the L1 norm should be used for function

approximation. However, the L2 norm which minimizes not the absolute value of the error but the error power is much easier to apply (in linear systems) and also produces an important interpretation of the results (nonlinear regression) as we saw in the probabilistic interpretation of the mappings. In practical situations either norm can be used.

Return to Text

function approximation

Became a theory last century with the formal definition of a limit by Cauchy and Weierstrass . It culminated a long road of discoveries by mathematical giants such as Euler, Legendre and Gauss, motivated by astronomical observations. The goal was to approximate difficult mathematical functions by ensembles of simpler functions. Approximation requires the definition of an error which implies a metric space to define the distance between the true function and the approximation. Moreover, the availability of a set of simpler functions is postulated.

functional analysis

Is the branch of analysis where the functions do not depend on single numbers but on collections (eventually an entire range of a numerical function) of components. An example is a function of a vector.

Weierstrass

Augustin Cauchy (1789-1857) and Karl Weierstrass (1815-1897) were the fathers of calculus. They captured the idea of the limit in a precise mathematical way, and open up new horizons in approximation theory.

series

are iterated sums of terms produced by a general rule, with eventually an infinite number of terms. An example is

$$1+\frac{1}{10}+\frac{1}{100}+\frac{1}{1000}+.....$$

sampling theorem

Also called the Nyquist theorem states that one can uniquely reconstruct a time signal from its sampled representation if we sample at least at twice the highest frequency present in the signal.

sinc

Is a time signal given by $\frac{\sin at}{at}$. It is the noncausal response of the ideal reconstruction filter.

Fourier series

Joseph Fourier in the late XVIII century showed that any periodic signal no matter how complex could be decomposed in sums (eventually infinite) of simple sinewaves of different frequencies. These decompositions are called the Fourier series. If y(t) is real

$$y(t) = Y_0 + \sum_{i=1}^{\infty} |Y_i| \cos(\frac{2\pi it}{T} + \theta_i)$$

delta function

The delta function can be thought of as the limit of a rectangular pulse of high $1/\varepsilon$ and width ε when ε goes to zero. Mathematically, it is a function $\delta(t)$ that obeys the relation

$$f(t) = \int_{-\infty}^{\infty} f(t-\tau)\delta(\tau)d\tau$$

linear systems theory

It is a highly mathematical branch of electrical engineering that studies linear functions, their properties and their implementations.

eigenfunctions

Are the natural modes of a system, i.e. a signal is an eigenfunction of a system when the system output is a signal of the same shape, eventually of a different amplitude and phase. Eigenfunctions are related to the concept of eigenvector in linear algebra.

shift-invariant

A shift-invariant system is a system that produces the same output no matter if the signal appears at $t=t_0$ or any other time .

complex number

A complex number is a number z that can be written as z = Re(z)+j Im (z) where $j = \sqrt{-1}$.

statistical learning theory

Is a new branch of statistics that analyzes mathematically (in functional spaces) the learning process.

manifold

A manifold is a space in which the local geometry of each point looks like a little piece of Eucledian space.

polynomials

Are rational functions that can be put in the form $y = \sum_{i=1}^{N} a_i x^i$ where both x and a are real numbers

scientific method

Is the methodology utilized in science. See Karl Poper, **"**The logic of scientific discovery**"**, Harper Torch, 1968.

Volterra expansions

of a discrete system (y the output, u the input) have the form

$$y(n) = \alpha + \sum_{i=0}^{\infty} \beta_i u(n-i) + \sum_{i=0}^{\infty} \sum_{j=0}^{\infty} \gamma_{ij} u(n-i)u(n-j) + \ldots$$

square integrable

Is a function where the integral of the square of the function over the domain (here the real numbers) is finite.

Jorma Rissanen

See the book Stochastic Complexity in Statistical Inquiry, World Scientific, 1989.

Akaike

See **"**A new look at the statistical model identification**"**, IEEE Trans. Auto. Control, AC-19, 716-723, 1974

Tikonov

A. Tikhnov and V. Arsenin, **"**Solution of ill-posed problems**"**, Winston, Washington, 1977.

ill-posed

Is a term coined by Hadamard to mean that solutions are extremely dependent upon minor changes in the form of simple equations. Ill-posed problems arise very often when one tries to reverse a cause-effect relation.

indicator function

is a function that takes only two values 1 and 0 (or -1).

splines

The kth normalized B spline of degree r-1 is given by Cox-deBoor formula

$$B_{k,r}(t) = \frac{t - t_k}{t_{k+r-1} - t_k} B_{k,r-1}(t) + \frac{t_{k+r} - t}{t_{k+r} - t_{k+1}} B_{k+1,r-1}(t)$$

with $$B(t) = \begin{cases} 1 & t_k \le t \le t_{k+1} \\ 0 & otherwise \end{cases}$$

fiducial

is the name given to the point where the function is being approximated.

code

code is a systematic translation of the data into bits.

VC dimension

Vapnik-Chervonenkis introduced the concept of dimension of the learning machine which guarantees good generalization ability.

Cover Theorem

states that it is always possible to use a linear classifier to separate arbitrary data sets if the data is nonlinear mapped to a sufficient large feature space.

learning theory

see Vladmir Vapnik, The nature of statistical learning theory", Springer, 1995.

A. Barron

Approximation and estimation bounds for ANNs, IEEE Trans. Information Theory 39, #3, 930-945, 1993.

Park and Sandberg,

"Universal approximation using radial basis function networks", Neural Computation, vol 3, 303-314, 1989.

Bishop

Pattern Recognition with Neural Networks, Oxford, 1995.

Vladimir Vapnik

The Nature of Statistical Learning Theory, Springer Verlag, 1995 and Statistical Learning Theory, Wiley, 1998.

Parzen E.

On estimation of a probability density function and mode", Annals of Mathematical Statistics, 33, 1065-1076, 1962.

Simon Haykin

Neural Networks: a comprehensive foundation, McMillan, 1994.

Eq.1

$$\hat{f}(x,w) = \sum_{i=1} w_i \varphi_i(x)$$

Eq.4

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

Eq.11

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

Eq.2

$$\left| f(x) - \hat{f}(x,w) \right| < \varepsilon$$

Eq.14

$$<< t_k \mid x >>= \int d_k \, p(t_k \mid x) dt_k$$

Eq.30

$$J_{new} = J_c + \lambda J_r$$

Eq.25

$$\min_k \quad AIC(k) = N \quad InJ(k) + 2k$$

Eq.7

$$w_i =< f(x), \varphi_i(x) >$$

Eq.8

$$< f(x), \varphi(x) >= \int_D f\ (x)\varphi\ (x)dx$$

Eq.16

$$w_{ij}(n+1) = w_{ij}(n)(1-\lambda) + \eta\delta_i x_j$$

Eq.19

$$w_{ij}(n+1) = w_{ij}(n) + \eta\delta_i x_j + \lambda \operatorname{sgn}\left(w_{ij}\right)$$

Wahba

Grace Whaba, “Splines models for observational data”, SIAM, 1990.

Poggio and Girosi

"Networks for approximation and learning", Proc. IEEE vol 78, 1990.

R. Zemel

Minimum Description Length Framework for Unsupervised Learning, Ph.D. Thesis, U. of Toronto, 1993.

Thilo Freiss

Support Vector Neural Networks: The kernel Adatron with bias and soft margin, University of Sheffield technical report.

Kay

Modern Spectral Estimation, Prentice Hall, 1988.

Index