9

Statistics and Neural Networks

9.1 Linear and nonlinear regression

Feed-forward networks are used to find the best functional fit for a set of input-output examples. Changes to the network weights allow fine-tuning of the network function in order to detect the optimal configuration. However, two complementary motivations determine our perception of what optimal means in this context. On the one hand we expect the network to map the known inputs as exactly as possible to the known outputs. But on the other hand the network must be capable of *generalizing*, that is, unknown inputs are to be compared to the known ones and the output produced is a kind of interpolation of learned values. However, good generalization and minimal reproduction error of the learned input-output pairs can become contradictory objectives.

9.1.1 The problem of good generalization

Figure 9.1 shows the problem from another perspective. The dots in the graphic represent the training set. We are looking for a function capable of mapping the known inputs into the known outputs. If linear approximation is used, as in the figure, the error is not excessive and new unknown values of the input x are mapped to the regression line.

Figure 9.2 shows another kind of functional approximation using linear splines which can reproduce the training set without error. However, when the training set consists of experimental points, normally there is some noise in the data. Reproducing the training set exactly is not the best strategy, because the noise will also be reproduced. A linear approximation as in Figure 9.1 could be a better alternative than the exact fit of the training data shown in Figure 9.2. This simple example illustrates the two contradictory objectives of functional approximation: minimization of the training error but also minimization of the error of yet unknown inputs. Whether or not the training set can be

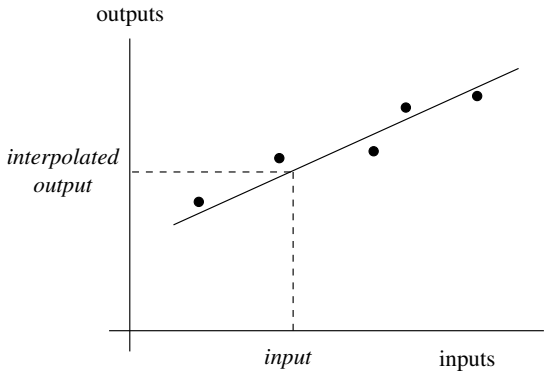

Fig. 9.1. Linear approximation of the training set

learned exactly depends on the number of degrees of freedom available to the network (number of weights) and the structure of the manifold from which the empirical data is extracted. The number of degrees of freedom determines the *plasticity* of the system, that is, its capability of approximating the training set. Increasing the plasticity helps to reduce the training error but can increase the error on the test set. Decreasing the plasticity excessively can lead to a large training and test error.

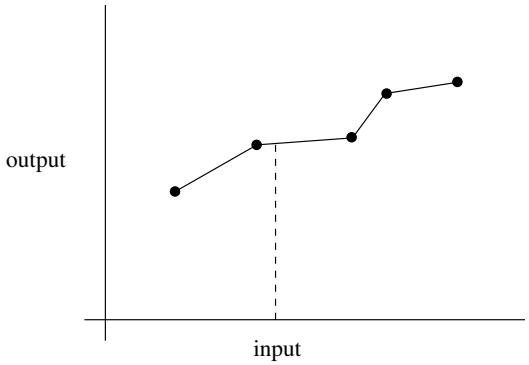

Fig. 9.2. Approximation of the training set with linear splines

There is no universal method to determine the optimal number of parameters for a network. It all depends on the structure of the problem at hand. The best results can be obtained when the network topology is selected taking into account the known interrelations between input and output (see Chap. 14). In the example above, if a theoretical analysis leads us to conjecture a linear correspondence between input and output, the linear approximation would be the best although the polylinear approximation has a smaller training error.

This kind of functional approximation to a given training set has been studied by statisticians working in the field of linear and nonlinear regression. The backpropagation algorithm is in some sense only a numerical method for statistical approximation. Analysis of the linear case can improve our understanding of this connection.

9.1.2 Linear regression

Linear associators were introduced in Chap. 5: they are computing units which just add their weighted inputs. We can also think of them as the integration part of nonlinear units. For the n-dimensional input $(x_1, x_2, \ldots, x_n)$ the output of a linear associator with weight vector $(w_1, w_2, \ldots, w_n)$ is $y = w_1 x_1 + \cdots + w_n x_n$. The output function represents a hyperplane in $(n+1)$-dimensional space. Figure 9.3 shows the output function of a linear associator with two inputs. The learning problem for such a linear associator is to reproduce the output of the input vectors in the training set. The points corresponding to the training set are shown in black in Figure 9.3. The parameters of the hyperplane must be selected to minimize the error, that is, the distance from the training set to the hyperplane. The backpropagation algorithm can be used to find them.

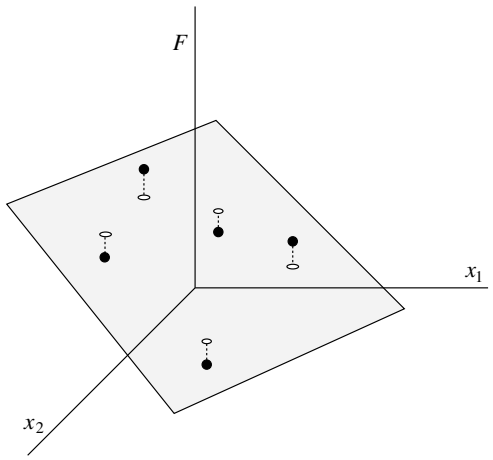

Fig. 9.3. Learning problem for a linear associator

Consider a training set $T = \{(\mathbf{x}^1, a_1), \ldots, (\mathbf{x}^m, a_m)\}$ for a linear associator, where the inputs $\mathbf{x}^1, \ldots, \mathbf{x}^m$ are n-dimensional vectors and the outputs $a_1, \ldots, a_m$ real numbers. We are looking for the weight vector $(w_1, \ldots, w_n)$ which minimizes the quadratic error

$$E = \frac{1}{2}\left[\left(a_1 - \sum_{i=1}^{n} w_i x_i^1\right)^2 + \cdots + \left(a_m - \sum_{i=1}^{n} w_i x_i^m\right)^2\right] \tag{9.1}$$

where x_i^j denotes the i-th component of the j-th input vector. The components of the gradient of the error function are

$$\frac{\partial E}{\partial w_j} = -\left(a_1 - \sum_{i=1}^{n} w_i x_i^1\right) x_j^1 - \cdots - \left(a_m - \sum_{i=1}^{n} w_i x_i^m\right) x_j^m \tag{9.2}$$

for $j = 1, 2, \ldots, n$. The minimum of the error function can be found analytically by setting $\nabla E = \mathbf{0}$ or iteratively using gradient descent. Since the error function is purely quadratic the global minimum can be found starting from randomly selected weights and making the correction $\Delta w_j = -\gamma \partial E / \partial w_j$ at each step.

Figure 9.4 shows the B-diagram for a linear associator. The training vector $\mathbf{x}^1$ has been used to compute the error E_1. The partial derivatives $\partial E/\partial w_1, \ldots, \partial E/\partial w_n$ can be computed using a backpropagation step.

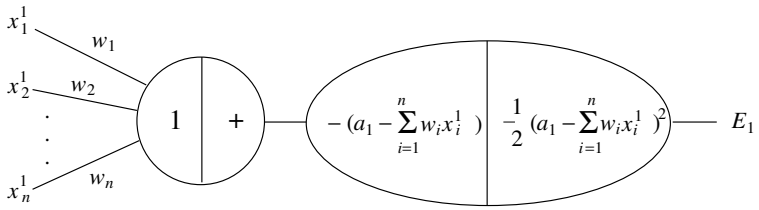

Fig. 9.4. Backpropagation network for the linear associator

The problem of finding optimal weights for a linear associator and for a given training set T is known in statistics as *multiple linear regression*. We are looking for constants $w_0, w_1, \ldots, w_n$ such that the y values can be computed from the x values:

$$y_i = w_0 + w_1 x_1^i + w_2 x_2^i + \cdots + w_n x_n^i + \varepsilon_i \,,$$

where ε_i represents the approximation error (note that we now include the constant w_0 in the approximation). The constants selected should minimize the total quadratic error $\sum_{i=1}^{n} \varepsilon_i^2$. This problem can be solved using algebraic methods. Let X denote the following $m \times (n+1)$ matrix:

$$\mathbf{X} = \begin{pmatrix} 1 & x_1^1 & \cdots & x_n^1 \\ 1 & x_1^2 & \cdots & x_n^2 \\ \vdots & \vdots & \ddots & \vdots \\ 1 & x_1^m & \cdots & x_n^m \end{pmatrix}$$

The rows of the matrix consist of the extended input vectors. Let $\mathbf{a}$, $\mathbf{w}$ and ε denote the following vectors:

$$\mathbf{a} = \begin{pmatrix} a_1 \\ a_2 \\ \vdots \\ a_m \end{pmatrix} \qquad \mathbf{w} = \begin{pmatrix} w_0 \\ w_1 \\ \vdots \\ w_n \end{pmatrix} \qquad \boldsymbol{\varepsilon} = \begin{pmatrix} \varepsilon_1 \\ \varepsilon_2 \\ \vdots \\ \varepsilon_m \end{pmatrix}$$

The vector $\mathbf{w}$ must satisfy the equation $\mathbf{a} = \mathbf{X}\mathbf{w} + \boldsymbol{\varepsilon}$, where the norm of the vector $\boldsymbol{\varepsilon}$ must be minimized. Since

$$\|\boldsymbol{\varepsilon}\|^2 = (\mathbf{a} - \mathbf{X}\mathbf{w})^{\mathrm{T}}(\mathbf{a} - \mathbf{X}\mathbf{w})$$

the minimum of the norm can be found by equating the derivative of this expression with respect to $\mathbf{w}$ to zero:

$$\frac{\partial}{\partial \mathbf{w}}(\mathbf{a} - \mathbf{X}\mathbf{w})^{\mathrm{T}}(\mathbf{a} - \mathbf{X}\mathbf{w}) = -2\mathbf{X}^{\mathrm{T}}\mathbf{a} + 2\mathbf{X}^{\mathrm{T}}\mathbf{X}\mathbf{w} = \mathbf{0}.$$

It follows that $\mathbf{X}^{\mathrm{T}}\mathbf{X}\mathbf{w} = \mathbf{X}^{\mathrm{T}}\mathbf{a}$ and if the matrix $\mathbf{X}^{\mathrm{T}}\mathbf{X}$ is invertible, the solution to the problem is given by

$$\mathbf{w} = \left(\mathbf{X}^{\mathrm{T}}\mathbf{X}\right)^{-1}\mathbf{X}^{\mathrm{T}}\mathbf{a}.$$

9.1.3 Nonlinear units

Introducing the sigmoid as the activation function changes the form of the functional approximation produced by a network. In Chap. 7 we saw that the form of the functions computed by the sigmoidal units corresponds to a smooth step function. As an example in Figure 9.5 we show the continuous output of two small networks of sigmoidal units. The first graphic corresponds to the network in Figure 6.2 which can compute an approximation to the XOR function when sigmoidal units are used. The output of the network is approximately 1 for the inputs $(1, 0)$ and $(0, 1)$ and approximately 0 for the inputs $(0, 0)$ and $(1, 1)$. The second graph corresponds to the computation of the NAND function with three sigmoidal units distributed in two layers.

Much more complicated functions can be produced with networks which are not too elaborate. Figure 9.8 shows the functions produced by a network with three and four hidden units and a single output unit. Small variations of the network parameters can produce widely differing shapes and this leads us to suspect that any continuous function could be approximated in this manner, if only enough hidden units are available. The number of foldings of the functions corresponds to the number of hidden units. In this case we have a situation similar to when polynomials are used to approximate experimental data—the degree of the polynomial determines the number of degrees of freedom of the functional approximation.

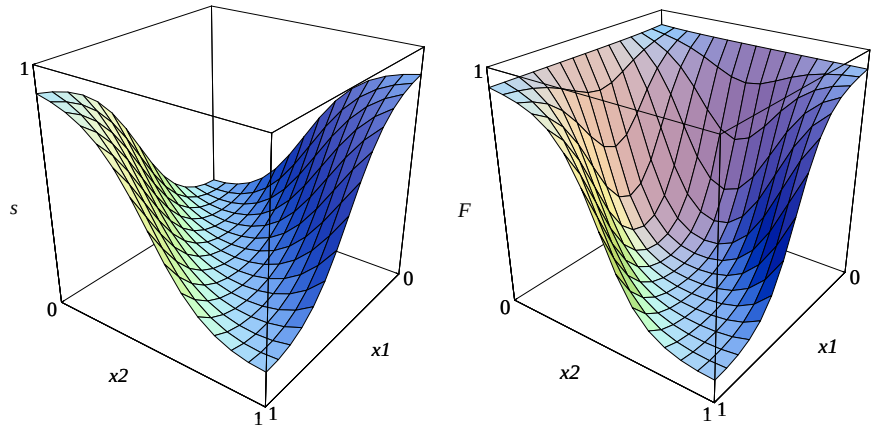

Fig. 9.5. Output of networks for the computation of XOR (left) and NAND (right)

Logistic regression

Backpropagation applied to a linear association problem finds the parameters of the optimal linear regression. If a sigmoid is computed at the output of the linear associator, we are dealing with the conventional units of feed-forward networks.

There is a type of nonlinear regression which has been applied in biology and economics for many years called *logistic regression*. Let the training set T be $\{(\mathbf{x}^1, a_1), (\mathbf{x}^2, a_2), \ldots, (\mathbf{x}^m, a_m)\}$, where the vectors $\mathbf{x}^i$ are n-dimensional. A sigmoidal unit is to be trained with this set. We are looking for the n-dimensional weight vector $\mathbf{w}$ which minimizes the quadratic error

$$E = \sum_{i=1}^{m} (a_i - s(\mathbf{w} \cdot \mathbf{x}^i))^2 \,,$$

where s denotes the sigmoid function. Backpropagation solves the problem directly by minimizing E. An approximation can be found using the tools of linear regression by inverting the sigmoid and minimizing the new error function

$$E' = \sum_{i=1}^{m} (s^{-1}(a_i) - \mathbf{w} \cdot \mathbf{x}^i)^2 .$$

Since the a_i are constants this step can be done at the beginning so that a linear associator has to approximate the outputs

$$a_i' = s^{-1}(a_i) = \ln\left(\frac{a_i}{1 - a_i}\right), \quad \text{for } i = 1, \ldots, m. \tag{9.3}$$

All the standard machinery of linear regression can be used to solve the problem. Equation (9.3) is called the *logit transformation* [34]. It simplifies the

approximation problem but at a cost. The logit transformation modifies the weight given to the individual deviations. If the target value is 0.999 and the sigmoid output is 0.990, the approximation error is 0.009. If the logit transformation is used, the approximation error for the same combination is 2.3 and can play a larger role in the computation of the optimal fit. Consequently, backpropagation is a type of nonlinear regression [323] which solves the approximation problem in the original domain and is therefore more precise.

9.1.4 Computing the prediction error

The main issue concerning the kind of functional approximation which can be computed with neural networks is to obtain an estimate of the prediction error when new values are presented to the network. The case of linear regression has been studied intensively and there are closed-form formulas for the expected error and its variance. In the case of nonlinear regression of the kind which neural networks implement, it is very difficult, if not impossible, to produce such analytic formulas. This difficulty also arises when certain kinds of statistics are extracted from empirical data. It has therefore been a much-studied problem. In this subsection we show how to apply some of these statistical methods to the computation of the expected generalization error of a network.

One might be inclined to think that the expected generalization error of a network is just the square root of the mean squared training error. If the training set consists of N data points and E is the total quadratic error of the network over the training set, the generalization error $\tilde{E}$ could be set to

$$\tilde{E} = \sqrt{E/N}.$$

This computation, however, would tend to underestimate the true generalization error because the parameters of the network have been adjusted to deal with exactly this data set and could be biased in favor of its elements. If many additional input-output pairs that do not belong to the training set are available, the generalization error can be computed directly. New input vectors are fed into the network and the mean quadratic deviation is averaged over many trials. Normally, this is not the case and we want to use all of the available data to train the network *and* to predict the generalization error.

The *bootstrap* method, proposed by Efron in 1979, deals with exactly this type of statistical problem [127]. The key observation is that existent data can be used to adjust a predictor (such as a regression line), yet it also tells us something about the distribution of the future expected inputs. In the real world we would perform linear regression and compute the generalization error using new data not included in the training set. In the *bootstrap world* we try to imitate this situation by sampling randomly from the existing data to create different training sets.

Here is how the bootstrap method works: assume that a data set $X = \{x_1, x_2, \ldots, x_n\}$ is given and that we compute a certain statistic $\hat{\theta}$ with this

data. This number is an estimate of the true value θ of the statistic over the whole population. We would like to know how reliable is $\hat{\theta}$ by computing its standard deviation. The data is produced by an unknown probability distribution F. The bootstrap assumption is that we can approximate this distribution by randomly sampling from the set X with replacement. We generate a new data set X^* in this way and compute the new value of the statistics which we call $\hat{\theta}^*$. This procedure can be repeated many times with many randomly generated data sets. The standard deviation of $\hat{\theta}$ is approximated by the standard deviation of $\hat{\theta}^*$.

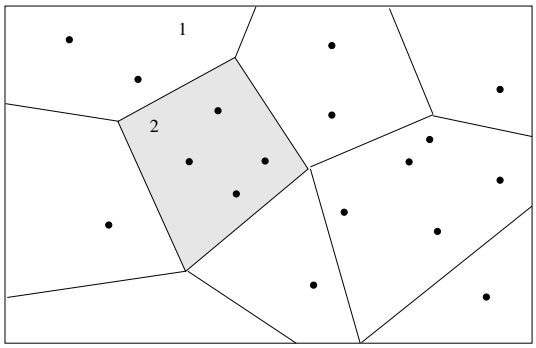

Fig. 9.6. Distribution of data in input space

Figure 9.6 graphically shows the idea behind the bootstrap method. The experimental data comes from a certain input space. If we want to compute some function over the whole input space (for example if we want to find the centroid of the complete input domain), we cannot because we only have some data points, but we can produce an estimate assuming that the distribution of the data is a good enough approximation to the actual probability distribution. The figure shows several regions where the data density is different. We approximate this varying data density by sampling *with replacement* from the known data. Region 2 in the figure will then be represented twice as often as region 1 in the generated samples. Thus our computations use not the unknown probability distribution F, but an approximation $\hat{F}$. This is the "plug-in principle": the empirical distribution $\hat{F}$ is an estimate of the true distribution F. If the approximation is good enough we can derive more information from the data, such as the standard deviation of function values computed over the empirical data set.

Algorithm 9.1.1 *Bootstrap algorithm*

i) Select N independent bootstrap samples $\mathbf{x}^{*1}, \mathbf{x}^{*2}, \ldots, \mathbf{x}^{*N}$ each consisting of n data values selected with replacement from X.

ii) Evaluate the desired statistic S corresponding to each bootstrap sample,

$$\hat{\theta}^*(b) = S(\mathbf{x}^{*b}) \qquad b = 1, 2, \ldots, N.$$

iii) Estimate the standard error $\hat{s}_N$ by the sample standard deviation of the N replications

$$\hat{s}_N = \left(\sum_{b=1}^{N} [\hat{\theta}^*(b) - \tilde{\theta}]^2 / (N-1) \right)^{1/2}$$

where $\tilde{\theta} = \sum_{b=1}^{N} \hat{\theta}^*(b)/N$.

In the case of functional approximation the bootstrap method can be applied in two different ways, but the simpler approach is the following. Assume that a neural network has been trained to approximate the function φ associated with the training set $T = \{(\mathbf{x}_1, \mathbf{t}_1), \ldots, (\mathbf{x}_m, \mathbf{t}_m)\}$ of m input-output pairs. We can compute a better estimate of the expected mean error by generating N different bootstrap training sets. Each bootstrap training set is generated by selecting m input-output pairs from the original training set randomly and with replacement. The neural network is trained always using the same algorithm and stop criterion. For each network trained we compute:

- The mean squared error Q_i^* for the i-th bootstrap training set,
- The mean squared error for the original data, which we call Q_i^0.

The standard deviation of the Q_i^* values is an approximation to the true standard deviation of our function fit.

In general, Q_i^* will be lower than Q_i^0, because the training algorithm adjusts the parameters optimally for the training set at hand. The *optimism* in the computation of the expected error is defined as

$$O = \frac{1}{B} \sum_{i=1}^{B} (Q_i^0 - Q_i^*).$$

The idea of this definition is that the original data set is a fair representative of the whole input space and the unknown sample distribution F, whereas the bootstrap data set is a fair representative of a generic training set extracted from input space. The optimism O gives a measure of the degree of underestimation present in the mean squared error originally computed for a training set.

There is a complication in this method which does not normally arise when the statistic of interest is a generic function. Normally, neural networks training is nondeterministic because the error function contains several global minima which can be reached when gradient descent learning is used. Retraining of networks with different data sets could lead to several completely

different solutions in terms of the weights involved. This in turn can lead to disparate estimates of the mean quadratic deviation for each bootstrap data set. However, if we want to analyze what will happen in general when the given network is trained with data coming from the given input space, this is precisely the right thing to do because we never know at which local minima training stopped. If we want to analyze just one local minimum we must ensure that training always converges to *similar* local minima of the error function (only similar because the shape of the error function depends on the training set used and different training sets have different local minima). One way to do this was proposed by Moody and Utans, who trained a neural network using the original data set and then used the weights found as initial weights for the training of the bootstrap data sets [319]. We expect gradient descent to converge to nearby solutions for each of the bootstrap data sets. Especially important is that with the bootstrap method we can compute confidence intervals for the neural approximation [127].

9.1.5 The jackknife and cross-validation

A relatively old statistical technique which can be considered a predecessor of the bootstrap method is the *jackknife*. As in the bootstrap, new data samples are generated from the original data, but in a much simpler manner. If n data points are given, one is left out, the statistic of interest is computed with the remaining $n-1$ points and the end result is the average over the n different data sets. Figure 9.7 shows a simple example comparing the bootstrap with the jackknife for the case of three data points, where the desired statistic is the centroid position of the data set. In the case of the bootstrap there are 10 possible bootstrap sets which lead to 10 different computed centroids (shown in the figure as circles with their respective probabilities). For the jackknife there are 3 different data sets (shown as ellipses) and centroids. The average of the bootstrap and jackknife "populations" coincide in this simple example. The d-jackknife is a refinement of the standard method: instead of leaving one point out of the data set, d different points are left out and the statistic of interest is computed with the remaining data points. Mean values and standard deviations are then computed as in the bootstrap.

In the case of neural networks *cross-validation* has been in use for many years. For a given training set T some of the input-output pairs are reserved and are not used to train the neural network (typically 5% or 10% of the data). The trained network is tested with these reserved input-output pairs and the observed average error is taken as an approximation of the true mean squared error over the input space. This estimated error is a good approximation if both training and test set fully reflect the probability distribution of the data in input space. To improve the results *k-fold cross-validation* can be used. The data set is divided into k random subsets of the same size. The network is trained k times, each time leaving one of the k subsets out of the training set and testing the mean error with the subset which was left out. The average of

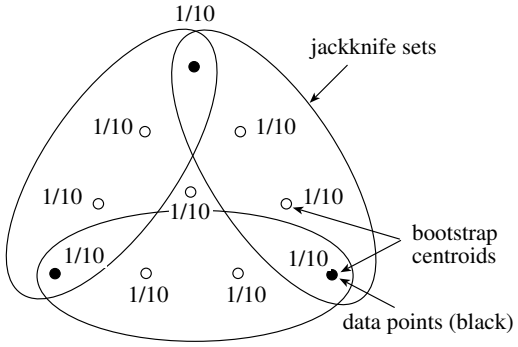

Fig. 9.7. Comparison of the bootstrap and jackknife sampling points for $n = 3$

the k computed mean quadratic errors is our estimate of the expected mean quadratic error over the whole of input space. As in the case of the bootstrap, the initial values of the weights for each of the k training sets can be taken from previous results using the complete data set, a technique called *nonlinear cross-validation* by Moody and Utans [319], or each network can be trained with random initial weights. The latter technique will lead to an estimation of the mean quadratic deviation over different possible solutions of the given task.

The bootstrap, jackknife, and cross-validation are all methods in which raw computer power allows us to compute confidence intervals for statistics of interest. When applied to neural networks, these methods are even more computationally intensive because training the network repetitively consumes an inordinate amount of time. Even so, if adequate parallel hardware is available the bootstrap or cross-validation provides us with an assessment of the reliability of the network results.

9.1.6 Committees of networks

The methods for the determination of the mean quadratic error discussed in the previous section rely on training several networks with the same basic structure. If so much computing power is available, the approximation capabilities of an ensemble of networks is much better than just using one of the trained networks. The combination of the outputs of a group of neural networks has received several different names in the literature, but the most suggestive denomination is undoubtedly *committees* [339].

Assume that a training set of m input-output pairs $(\mathbf{x}^1, t_1), \ldots, (\mathbf{x}^m, t_m)$ is given and that N networks are trained using this data. For simplicity we consider n-dimensional input vectors and a single output unit. Denote by f_i the network function computed by the i-th network, for $i = 1, \ldots, N$. The network function f produced by the committee of networks is defined as

$$f = \frac{1}{N} \sum_{i=1}^{N} f_i.$$

The rationale for this averaging over the network functions is that if each one of the approximations is biased with respect to some part of input space, an average over the ensemble of networks can reduce the prediction error significantly. For each network function f_i we can compute an m-dimensional vector $\mathbf{e}^i$ whose components are the approximation error of the function f_i for each input-output pair. The quadratic approximation error Q of the ensemble function f is

$$Q = \sum_{i=1}^{m} \left(t_i - \frac{1}{N} \sum_{j=1}^{N} f_j(\mathbf{x}^i) \right)^2 .$$

This can be written in matrix form by defining a matrix $\mathbf{E}$ whose N rows are the m components of each error vector $\mathbf{e}^i$:

$$\mathbf{E} = \begin{pmatrix} e_1^1 & e_2^1 & \cdots & e_m^1 \\ \vdots & \vdots & \ddots & \vdots \\ e_1^N & e_2^N & \cdots & e_m^N \end{pmatrix}$$

The quadratic error of the ensemble is then

$$Q = \left| \frac{1}{N}(1,1,\ldots,1)\mathbf{E} \right|^2 = \frac{1}{N^2}(1,1,\ldots,1)\mathbf{E}\mathbf{E}^{\mathrm{T}}(1,1,\ldots,1)^{\mathrm{T}} \tag{9.4}$$

The matrix $\mathbf{E}\mathbf{E}^{\mathrm{T}}$ is the correlation matrix of the error residuals. If each function approximation produces uncorrelated error vectors, the matrix $\mathbf{E}\mathbf{E}^{\mathrm{T}}$ is diagonal and the i-th diagonal element Q_i is the sum of quadratic deviations for each functional approximation, i.e., $Q_i = \|\mathbf{e}^i\|^2$. In this case

$$Q = \frac{1}{N} \left(\frac{1}{N}(Q_1 + \cdots + Q_N) \right),$$

and this means that the total quadratic error of the ensemble is smaller by a factor $1/N$ than the average of the quadratic errors of the computed functional approximations. Of course this impressive result holds only if the assumption of uncorrelated error residuals is true. This happens mostly when N is not too large. In some cases even $N = 2$ or $N = 3$ can lead to significant improvement of the approximation capabilities of the combined network [339].

If the quadratic errors are not uncorrelated, that is if $\mathbf{E}\mathbf{E}^{\mathrm{T}}$ is not symmetric, a weighted combination of the N functions f_i can be used. Denote the i-th weight by w_i. The ensemble functional approximation f is now

$$f = \sum_{i=1}^{N} w_i f_i.$$

The weights w_i must be computed in such a way as to minimize the expected quadratic deviation of the function f for the given training set. With the same definitions as before and with the constraint $w_1 + \cdots + w_N = 1$ it is easy to see that equation (9.4) transforms to

$$Q = \frac{1}{N^2}(w_1, w_2, \ldots, w_N)\mathbf{E}\mathbf{E}^{\mathrm{T}}(w_1, w_2, \ldots, w_N)^{\mathrm{T}}.$$

The minimum of this expression can be found by differentiating with respect to the weight vector $(w_1, w_2, \ldots, w_N)$ and setting the result to zero. But because of the constraint $w_1 + \cdots + w_N = 1$ a Lagrange multiplier λ has to be included so that the function to be minimized is

$$\begin{aligned} Q' &= \frac{1}{N^2}\mathbf{w}\mathbf{E}\mathbf{E}^{\mathrm{T}}\mathbf{w}^{\mathrm{T}} + \lambda(1, 1, \ldots, 1)\mathbf{w}^{\mathrm{T}} \\ &= \frac{1}{N^2}\mathbf{w}\mathbf{E}\mathbf{E}^{\mathrm{T}}\mathbf{w}^{\mathrm{T}} + \lambda\mathbf{1}\mathbf{w}^{\mathrm{T}} \end{aligned}$$

where $\mathbf{1}$ is a row vector with all its N components equal to 1. The partial derivative of Q' with respect to $\mathbf{w}$ is set to zero and this leads to

$$\frac{1}{N^2}\mathbf{w}\mathbf{E}\mathbf{E}^{\mathrm{T}} + \lambda\mathbf{1} = 0\,.$$

If the matrix $\mathbf{E}\mathbf{E}^{\mathrm{T}}$ is invertible this leads to

$$\mathbf{w} = -\lambda N^2\mathbf{1}(\mathbf{E}\mathbf{E}^{\mathrm{T}})^{-1}.$$

From the constraint $\mathbf{w}\mathbf{1}^{\mathrm{T}} = 1$ we deduce

$$\mathbf{w}\mathbf{1}^{\mathrm{T}} = -\lambda N^2\mathbf{1}(\mathbf{E}\mathbf{E}^{\mathrm{T}})^{-1}\mathbf{1}^{\mathrm{T}} = 1\,,$$

and therefore

$$\lambda = -\frac{1}{N^2\mathbf{1}(\mathbf{E}\mathbf{E}^{\mathrm{T}})^{-1}\mathbf{1}^{\mathrm{T}}}.$$

The final optimal set of weights is

$$\mathbf{w} = \frac{\mathbf{1}(\mathbf{E}\mathbf{E}^{\mathrm{T}})^{-1}}{\mathbf{1}(\mathbf{E}\mathbf{E}^{\mathrm{T}})^{-1}\mathbf{1}^{\mathrm{T}}},$$

assuming that the denominator does not vanish. Notice that the constraint $\mathbf{w}\mathbf{1}^{\mathrm{T}}$ is introduced only to simplify the analysis of the quadratic error.

This method can become prohibitive if the matrix EE^{T} is ill-conditioned or if its computation requires too many operations. In that case an adaptive method can be used. Note that the vector of weights can be learned using a Lagrange network of the type discussed in Chap. 7.