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

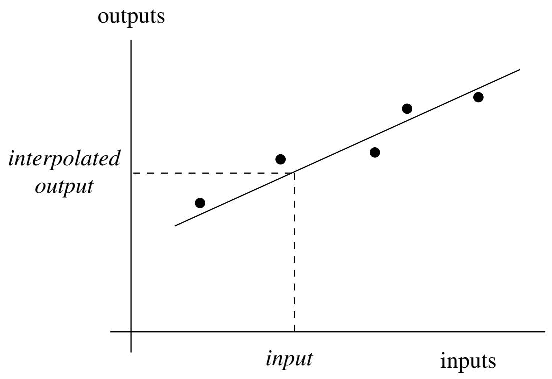


Fig. 9.1. Linear approximation of the training set

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.

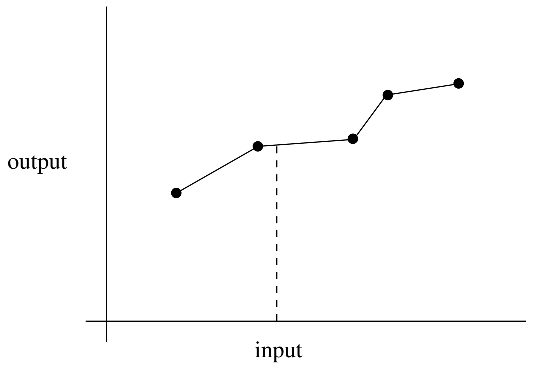


Fig. 9.2. Approximation of the training set with linear splines

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 $y = w_1x_1 + \cdots + w_nx_n$. 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.

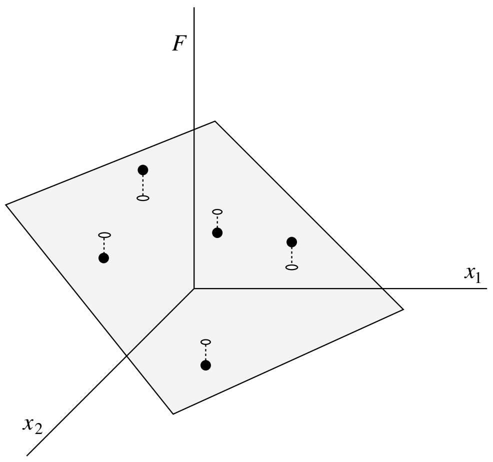


Fig. 9.3. Learning problem for a linear associator

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.

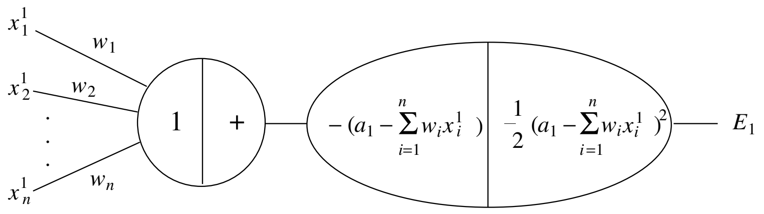

x_1^1
w_1
x_2^1
w_2
x_n^1
w_n
1
+
$-(a_1 - \sum_{i=1}^{n} w_i x_i^1)$
$\frac{1}{2}(a_1 - \sum_{i=1}^{n} w_i x_i^1)^2$
E_1

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 **a**, **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 **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 **w** to zero:

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

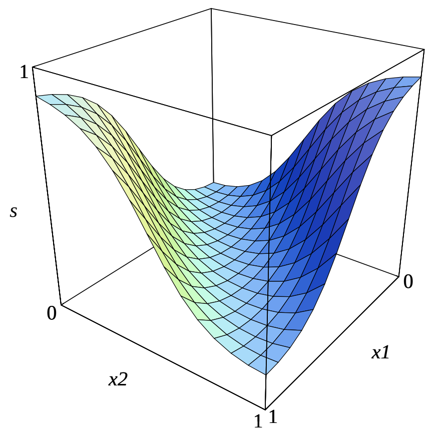
It follows that $\mathbf{X}^{\mathrm{T}}\mathbf{X}\mathbf{w} = \mathbf{X}^{\mathrm{T}}\mathbf{a}$ and if the matrix $\mathbf{X}^{\mathrm{T}}\mathbf{X}$ is invertible, the solution to the problem is given by

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

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.



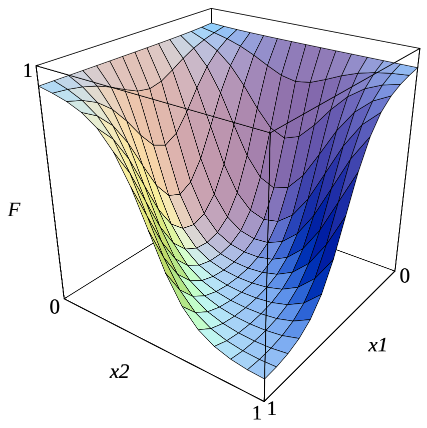


Fig. 9.5. Output of networks for the computation of XOR (left) and NAND (right)

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 $\hat{\theta}^*$.

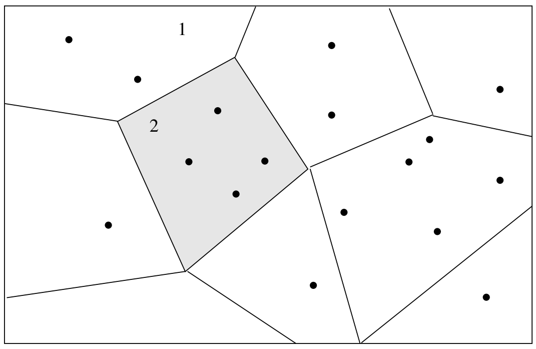


Fig. 9.6. Distribution of data in input space

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

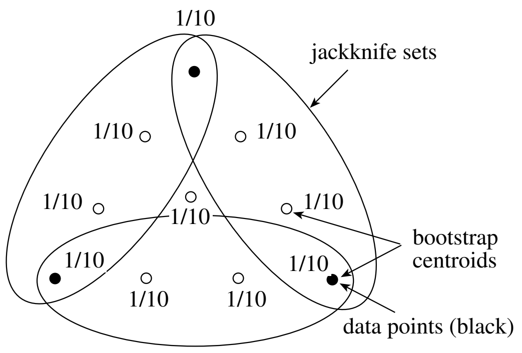


Fig. 9.7. Comparison of the bootstrap and jackknife sampling points for $n = 3$

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.

9.2 Multiple regression

Backpropagation networks are a powerful tool for function approximation. Figure 9.8 shows the graphs of the output function produced by four networks. The graph on the lower right was produced using four hidden units, the other using three. It can be seen that such a small variation in the topology of the network leads to an increase in the plasticity of the network function. The number of degrees of freedom of a backpropagation network, and therefore its plasticity, depends on the number of weights, which in turn are a function of the number of hidden units. How many of them should be used to solve a given problem? The answer is problem-dependent: in the ideal case, the network function should not have more degrees of freedom than the data itself, because otherwise there is a danger of overtraining the network.

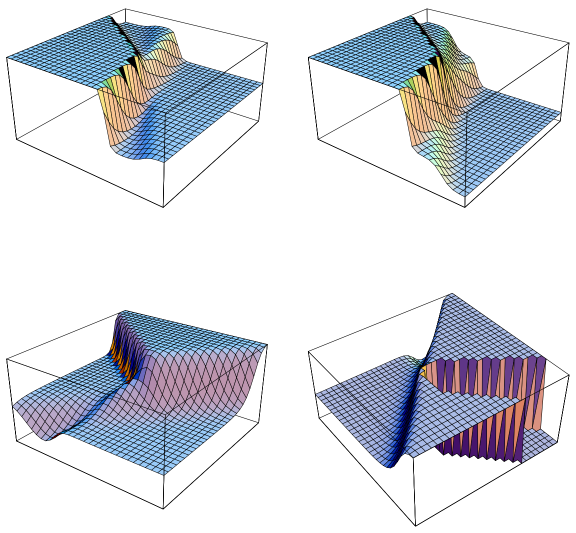


Fig. 9.8. Network functions of networks with one hidden layer

In this section we look at the role of the hidden layer, considering the interplay between layers in a network, but first of all we develop a useful visualization of the multiple regression problem.

9.2.1 Visualization of the solution regions

Consider a training set $T = \{(\mathbf{x}^1, a_1), (\mathbf{x}^2, a_2), \ldots, (\mathbf{x}^m, a_m)\}$ consisting of n-dimensional inputs and scalar outputs. We are looking for the best approximate solution to the system of equations

$$s(\mathbf{x}^i \cdot \mathbf{w}) = a_i, \quad \text{for } i = 1, 2, \ldots, m, \tag{9.5}$$

where s denotes, as usual, the sigmoid and $\mathbf{w}$ is the weight vector for a linear associator. If the outputs a_i are real and lie in the interval $(0, 1)$, equation (9.5) can be rewritten as

$$\mathbf{x}^i \cdot \mathbf{w} = s^{-1}(a_i), \quad \text{for } i = 1, 2, \ldots, m. \tag{9.6}$$

The m equations in (9.6) define m hyperplanes in weight space. If all hyperplanes meet at a common point $\mathbf{w}_0$, then this weight vector is the exact solution of the regression problem and the approximation error is zero. If there is no common intersection, there is no exact solution to the problem. When $m > n$, that is, when the number of training pairs is higher than the dimension of weight space, the hyperplanes may not meet at a single common point. In this case we must settle for an approximate solution of the regression problem.

Consider now the polytopes defined in weight space by the system of equations (9.6). First consider the case of a linear associator (eliminating the sigmoid and its inverse). The training equations are in this case

$$\mathbf{x}^i \cdot \mathbf{w} = a_i, \quad \text{for} \quad i = 1, 2, \ldots, m. \tag{9.7}$$

If there is no common intersection of the m hyperplanes, we look for the weight vector $\mathbf{w}'$ which minimizes the quadratic norms ε_i^2, where

$$\varepsilon_i = \mathbf{x}^i \cdot \mathbf{w}' - a_i, \quad \text{for} \quad i = 1, 2, \ldots, m. \tag{9.8}$$

A two-dimensional example can serve to illustrate the problem. Consider the three lines ℓ_1, ℓ_2 and ℓ_3 shown in Figure 9.9. The three do not intersect at a common point but the point with the minimum total distance to the three lines is α. This is also the site in weight space which can be found by using linear regression.

Important for the solution of the regression problem is that the square of the distance of α to each line is a quadratic function. The sum of quadratic functions is also quadratic and its minimization presents no special numerical problem. The point α in Figure 9.9 lies at a global minimum (in this case unique) of the error function.

The systems of equations (9.6) and (9.7) are very similar, but when the sigmoid is introduced the approximation error is given by

$$E = \sum_{i=1}^{m} \left(s(\mathbf{x}^i \cdot \mathbf{w}) - a_i \right)^2 , \tag{9.9}$$

which is not a quadratic function of $\mathbf{w}$. Suboptimal local minima can now appear.

Figure 9.10 shows the form of the error function for the same example of Figure 9.9 when the sigmoid is introduced. The error function now has three different local minima and the magnitude of the error can be different in any of them. Gradient descent would find one of the three minima, but not necessarily the best.

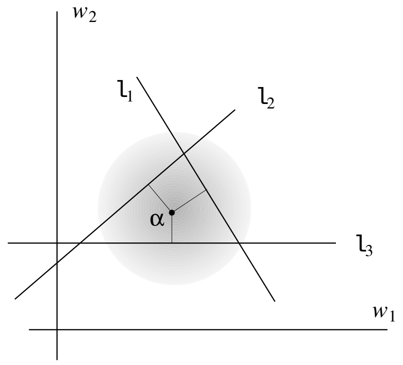


Fig. 9.9. Point of minimal distance to three lines

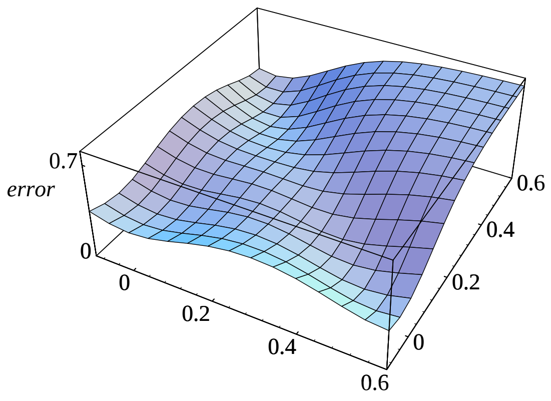


Fig. 9.10. Local minima of the error function

9.2.2 Linear equations and the pseudoinverse

Up to this point we have only considered the regression problem for individual linear associators. Consider now a network with two layers of weights, as shown in Figure 9.11. Assume that the training set consists of the n-dimensional input vectors $\mathbf{x}^1, \mathbf{x}^2, \ldots, \mathbf{x}^m$ and the k-dimensional output vectors $\mathbf{y}^1, \mathbf{y}^2, \ldots, \mathbf{y}^m$. Let $\mathbf{W}_1$ be the weight matrix between the input sites and the hidden layer (with the same conventions as in Chap. 7, but without bias terms). Let $\mathbf{W}_2$ be the weight matrix between hidden and output layer. If all units in the network are linear associators, the output for the input $\mathbf{x}$ is $\mathbf{x}\mathbf{W}_1\mathbf{W}2$. The weight matrix $\mathbf{W} = \mathbf{W}_1\mathbf{W}_2$ could be used in a network without a hidden layer and the output would be the same. The hidden layer plays a role *only* if the hidden units introduce some kind of nonlinearity in the computation.

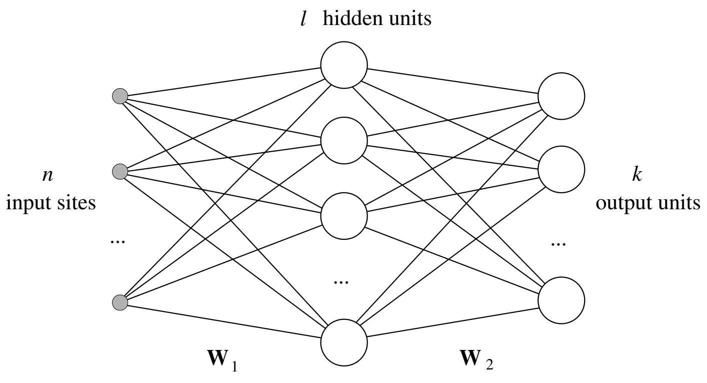


Fig. 9.11. Multilayer network

Assume that the hidden layer consists of ℓ units. Let $\mathbf{Y}$ denote the $m \times k$ matrix whose rows are the row vectors $\mathbf{y}^i$, for $i = 1, \ldots, m$. Let $\mathbf{Z}$ denote the $m \times \ell$ matrix whose rows are each one of the vectors produced by the hidden layer for the inputs $\mathbf{x}^1, \mathbf{x}^2, \ldots, \mathbf{x}^m$. The output of the network can be written as

$$\mathbf{Y} = \mathbf{Z}\mathbf{W}_2\,.$$

It is interesting to point out that this condition is always fulfilled if the m rows of the matrix $\mathbf{Z}$ are linearly independent. In that case there exists a matrix called the *pseudoinverse* $\mathbf{Z}^+$ such that $\mathbf{Z}\mathbf{Z}^+ = \mathbf{I}$, where $\mathbf{I}$ denotes the $m \times m$ identity matrix (we will discuss the properties of the pseudoinverse in Chap. **??**). Setting $\mathbf{W}_2 = \mathbf{Z}^+\mathbf{Y}$ we get $\mathbf{Y} = \mathbf{Z}\mathbf{W}_2$ because $\mathbf{Z}\mathbf{Z}^+\mathbf{Y} = \mathbf{Y}$ [349]. If the input vectors can be mapped to linearly independent vectors in the hidden layer the learning problem has a solution. This requires that $m \leq \ell$. This loose upper bound on the number of hidden units is not better than when each hidden unit acts as a feature detector for each input vector. Genuine learning problems can usually be solved with smaller networks.

9.2.3 The hidden layer

We can give the nonlinearity in the hidden layer a geometric interpretation. The computation between input sites and hidden layer corresponds to a linear transformation followed by a nonlinear "compression", that is, the evaluation of a squashing function at the hidden units. Let us first consider units with a step function as nonlinearity, that is, perceptrons. Assume that a unit in the hidden layer has the associated weight vector $\mathbf{w}_1$. All vectors in input space close enough to vector $\mathbf{w}_1$ are mapped to the same vector in feature space, for example the vector (0,1) in a network with two hidden units. A cone around vector $\mathbf{w}_1$ acts as its basin of attraction. Figure 9.12 also shows the basin of attraction of vector $\mathbf{w}_2$. Vectors close to $\mathbf{w}_2$ are mapped to the vector $(1, 0)$ in feature space.

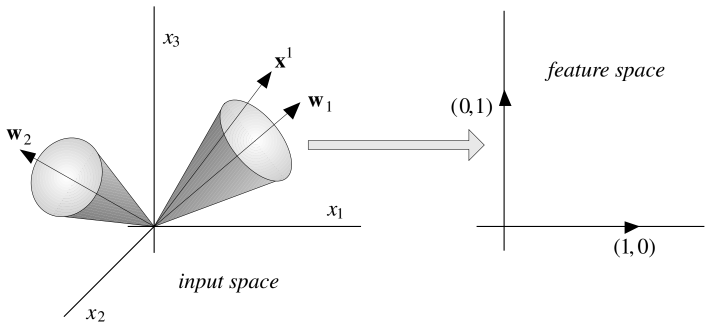


Fig. 9.12. Mapping input space into feature space

If all computed vectors in feature space are linearly independent, it is possible to find a matrix $\mathbf{W}_2$ that produces any desired output. If not a step function but a sigmoid is used as nonlinearity, the form of the basins of attraction changes and the vectors in feature space can be a combination of the basis vectors.

We can summarize the functioning of a network with a hidden layer in the following way: the hidden layer defines basins of attraction in input space so that the input vectors are mapped to vectors in feature space. Then, it is necessary to solve a linear regression problem between hidden and output layer in order to minimize the quadratic error over the training set.

9.2.4 Computation of the pseudoinverse

If m input vectors are mapped to m ℓ-dimensional linearly independent vectors $\mathbf{z}^1, \mathbf{z}^2, \ldots, \mathbf{z}^m$ in feature space, the backpropagation algorithm can be used to find the $\ell \times m$ matrix $\mathbf{Z}^+$ for which $\mathbf{Z}\mathbf{Z}^+ = \mathbf{I}$ holds. In the special case $m = \ell$ we are looking for the inverse of the square matrix $\mathbf{Z}$. This can be done using gradient descent. We will come back to this problem in Chap. **??**. Here we only show how the linear regression problem can be solved.

The network in Figure 9.13 can be used to compute the inverse of $\mathbf{Z}$. The training input consists of the m vectors $\mathbf{z}^1, \mathbf{z}^2, \ldots, \mathbf{z}^m$ and the training output of the m rows of the $m \times m$ identity matrix (the elements of the identity matrix are represented by Kronecker's delta, and the j-th component of the network output for the i-th training vector by o^i_j). When the input vectors are linearly independent the error function has a unique global minimum, which can be found using the backpropagation algorithm. The weight matrix $\mathbf{W}$ converges to $\mathbf{Z}^{-1}$.

If the m input vectors are not linearly independent, backpropagation nevertheless finds a minimum of the error function. This corresponds to the problem in which we do not look for the common intersection of hyperplanes but for the point with the minimal cumulative distance to all of them (Figure 9.9). If the minimum is unique the network finds the pseudoinverse $\mathbf{Z}^+$ of the matrix

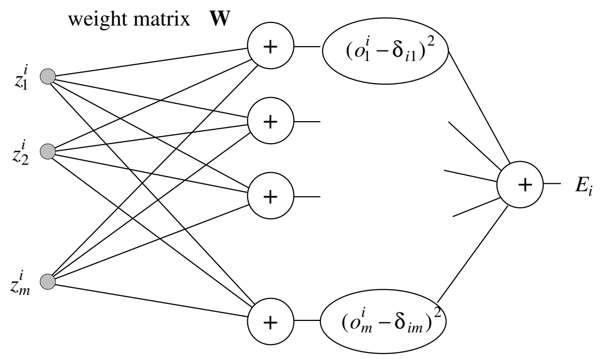


Fig. 9.13. Network for the computation of the inverse

Z. If the minimum is not unique (this can happen when $m < \ell$) it is necessary to minimize the norm of the weight matrix in order to find the pseudoinverse [14]. This can be done by adding a *decay term* to the backpropagation weight updates. The modified updates are given by

$$\Delta w_{ij} = -\gamma \frac{\partial E}{\partial w_{ij}} - \kappa w_{ij},$$

where κ and γ denote constants. The decay term tends to lower the magnitude of each weight—it corresponds to the negative partial derivative of w_{ij}^2 with respect to w_{ij}.

9.3 Classification networks

Multilayered neural networks have become a popular tool for a growing spectrum of applications. They are being applied in robotics, in speech or pattern recognition tasks, in coding problems, etc. It has been said that certain problems are theory-driven whereas others are data-driven. In the first class of problems theory predominates, in the latter there is much data but less theoretical understanding. Neural networks can discover statistical regularities and keep adjusting parameters even in a changing environment. It is interesting to look more closely at some applications where the statistical properties of neural networks become especially valuable.

There are many applications in which a certain input has to be classified as belonging to one of k different classes. The input is a certain measurement which we want to label in a predetermined way. This kind of problem can be solved by a *classification network* with k output units. Given a certain input vector $\mathbf{x}$ we expect the network to set the output line associated with the correct classification of $\mathbf{x}$ to 1 and the others to 0. In this section we discuss how to train such networks and then we look more closely at the range of output values being produced.

9.3.1 An application: NETtalk

Speech synthesis systems have been commercially available for quite a number of years now. They transform a string of characters into a string of phonemes by applying some linguistic transformation rules. The number of such rules is rather large and their interaction is not trivial [196].

In the 1980s Sejnowski and Rosenberg developed a backpropagation network that was able to synthesize speech of good quality without applying explicit linguistic transformations [396]. The authors used a backpropagation network composed of seven groups of 29 input sites, 80 hidden and 26 output units. The text to be pronounced by the system is scanned using a sliding window of seven characters. Each one of the characters is coded as one of 29 possible letters (one input line is set to 1 the other 28 to 0). Consequently there are $7 \times 29 = 203$ input sites. The network must produce the correct phoneme for the pronunciation of the character in the middle of the window, taking into account the three characters of context to the left and to the right. The network was connected to an electronic speech synthesizer capable of synthesizing 26 phonemes (later variants of NETtalk have used more phonemes). The network contains around 18,000 weights which must be found by the learning algorithm. We expect the network to be able to extract the statistical regularities from the training set by itself.

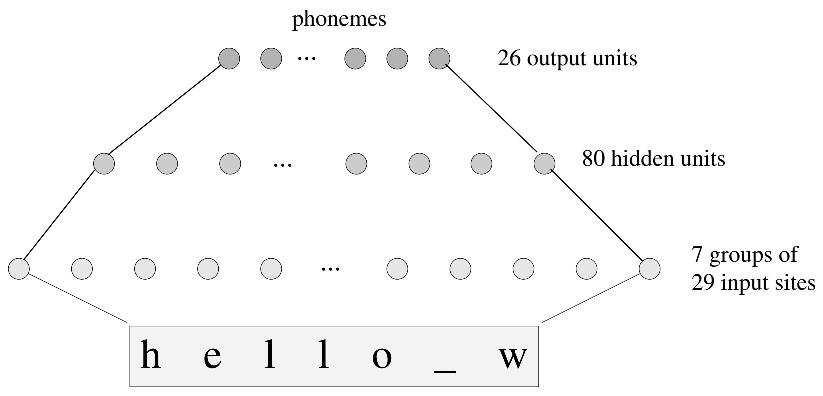


Fig. 9.14. The NETtalk architecture

The training set consists of a corpus of several hundred words, together with their phonetic transcription. The network is trained to produce a 1 at the output unit corresponding to the right phoneme. After training, an unknown text is scanned and the output units are monitored. At each time step only the unit with the maximum output level is selected. Surprisingly, the speech generated is of comparable quality to that produced by much more intricate rule-based systems.

Sejnowski and Rosenberg also looked at the proficiency of the network at different learning stages. At the beginning of learning the network made some of the same errors as children do when they learn to speak. Damaging some of the weights produced some specific deficiencies. Analyzing the code produced by the hidden units, the authors determined that some of them had implicitly learned some of the known linguistic rules.

NETtalk does not produce exactly ones or zeros, and the pronounced phoneme is determined by computing the maximum of all output values. It is interesting to ask what the produced output values stand for. One possibility is that it indeed represents the probability that each phoneme could be the correct one. However, the network is trained with binary values only, so that the question to be answered is: how do classifier networks learn probabilities?

9.3.2 The Bayes property of classifier networks

It is now well known that neural networks trained to classify an n-dimensional input x in one out of M classes can actually learn to compute the *a posteriori* probabilities that the input x belongs to each class. Several proofs of this fact, differing only in the details, have been published [65, 365], but they can be simplified. In this section we offer a shorter proof of the probability property of classifier neural networks proposed by Rojas [377].

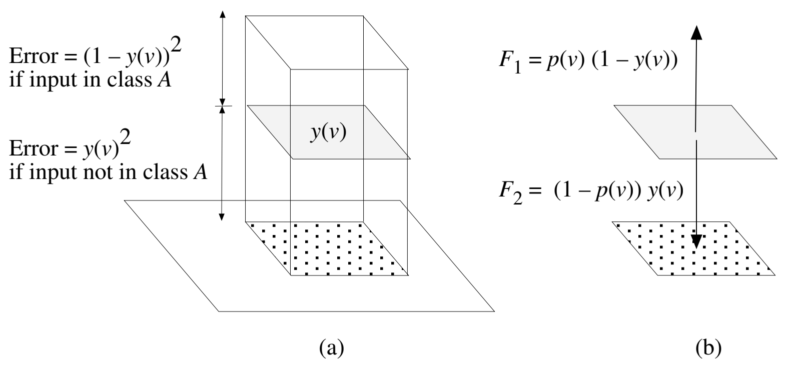


Fig. 9.15. The output $y(v)$ in a differential volume

Part (a) of Figure 9.15 shows the main idea of the proof. Points in an input space are classified as belonging to a class A or its complement. This is the first simplification: we do not have to deal with more than one class. In classifier networks, there is one output line for each class C_i, $i = 1, \ldots, M$. Each output C_i is trained to produce a 1 when the input belongs to class i, and otherwise a 0. Since the expected total error is the sum of the expected individual errors of each output, we can minimize the expected individual errors independently. This means that we need to consider only one output line and whether it should produce a 1 or a 0.

Proposition 13. *A classifier neural network perfectly trained and with enough plasticity can learn the a posteriori probability of an empirical data set.*

Proof. Assume that input space is divided into a lattice of differential volumes of size dv, each one centered at the n-dimensional point v. If at the output representing class A the network computes the value $y(v) \in [0,1]$ for any point x in the differential volume $V(v)$ centered at v, and denoting by $p(v)$ the probability $p(A|x \in V(v))$, then the total expected quadratic error is

$$E_A = \sum_V \{p(v)(1-y(v))^2 + (1-p(v))y(v)^2\}dv,$$

where the sum runs over all differential volumes in the lattice. Assume that the values $y(v)$ can be computed independently for each differential volume. This means that we can independently minimize each of the terms of the sum. This is done by differentiating each term with respect to the output $y(v)$ and equating the result to zero:

$$-2p(v)(1-y(v)) + 2(1-p(v))y(v) = 0.$$

From this expression we deduce $p(v) = y(v)$, that is, the output $y(v)$ which minimizes the error in the differential region centered at v is the a posteriori probability $p(v)$. In this case the expected error is

$$p(v)(1-p(v))^2 + (1-p(v))p(v)^2 = p(v)(1-p(v))$$

and E_A becomes the expected variance of the output line for class A. □

Note that extending the above analysis to other kinds of error functions is straightforward. For example, if the error at the output is measured by $\log(1-y(v))$ when the desired output is 1, and $\log(y(v))$ when it is 0, then the terms in the sum of expected differential errors have the form

$$p(v)\log(1-y(v)) + (1-p(v))\log(y(v)).$$

Differentiating and equating to zero we again find $y(v) = p(v)$.

This short proof also strongly underlines the two conditions needed for neural networks to produce a posteriori probabilities, namely *perfect training* and *enough plasticity* of the network, so as to be able to approximate the patch of probabilities given by the lattice of differential volumes and the values $y(v)$ which we optimize independently of each other.

It is still possible to offer a simpler visual proof "without words" of the Bayesian property of classifier networks, as is done in part (b) of Figure 9.15. When training to produce 1 for the class A and 0 for A^c, we subject the function produced by the network to an “upward force” proportional to the derivative of the error function, i.e., $(1 - y(v))$, and the probability $p(v)$, and a downward force proportional to $y(v)$ and the probability $(1 - p(v))$. Both forces are in equilibrium when $p(v) = y(v)$.

This result can be visualized with the help of Figure 9.16. Several non-disjoint clusters represent different classes defined on an input space. The correspondence of each input vector to a class is given only probabilistically. Such an input space could consist for example of n-dimensional vectors, in which each component is the numerical value assigned to each of n possible symptoms. The classes defined over this input space are the different illnesses. A vector of symptoms corresponds to an affliction with some probability. This is illustrated in Figure 9.16 with the help of Gaussian-shaped probability distributions. The clusters overlap, because sometimes the same symptoms can correspond to different ailments. Such an overlap could only be suppressed by acquiring more information.

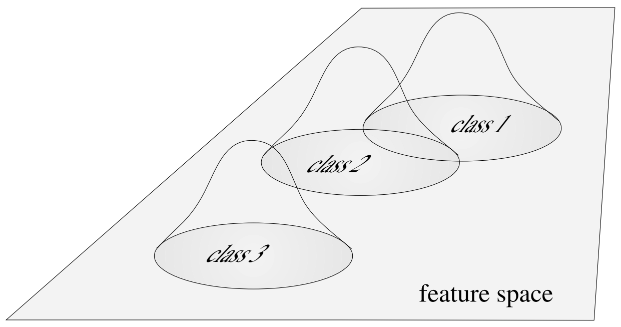


Fig. 9.16. Probability distribution of several classes in feature space

This is a nice example of the kind of application that such classification networks can have, namely in medical diagnosis. The existing data banks can be used as training sets for a network and to compute the margin of error associated with the classifications. The network can compute a first diagnosis, which is then given to a physician who decides whether or not to take this information into account.

9.3.3 Connectionist speech recognition

In automatic speech recognition we deal with the inverse problem of NETtalk: given a sequence of acoustic signals, transcribe them into text. Speech recognition is much more difficult than speech synthesis because cognitive factors play a decisive role. The recognition process is extremely sensitive to context in such a way that, if some phonemes are canceled in recorded speech, test subjects do not notice any difference. We are capable of separating speech signals from background noise (the so-called cocktail party effect) without any special effort, whereas this separation is a major computational problem for existing speech recognition systems. This leads to the suspicion that in this case deterministic rules would do much worse than a statistical system working with probabilities and likelihoods.

Building computers capable of automatically recognizing speech has been an old dream of both the field of electronics and computer science. Initial experiments were conducted as early as the 1950s, and in the 1960s some systems were already capable of recognizing vowels uttered by different speakers [12]. But until now all expectations have not been fully met. We all know of several small-scale commercial applications of speech technology for consumer electronics or for office automation. Most of these systems work with a limited vocabulary or are speaker-dependent in some way. Yet current research has as its goal the development of *large-vocabulary speaker-independent continuous speech recognition*. This long chain of adjectives already underlines the difficulties which still hamper the large-scale commercial application of automatic speech recognition: We would like the user to speak without artificial pauses, we would like that the system could understand anybody, and this without necessarily knowing the context of a conversation or monologue.

Artificial neural networks have been proposed as one of the building blocks for speech recognizers. Their function is to provide a statistical model capable of associating a vector of speech features with the probability that the vector could represent any of a given number of phonemes. Neural networks have here the function of statistical machines. Nevertheless we will see that our knowledge of the speech recognition process is still very limited so that fully connectionist models are normally not used. Researchers have become rather pragmatic and combine the best features of neural modeling with traditional algorithms or with other statistical approaches, like Hidden Markov Models, which we will briefly review. Current state-of-the-art systems combine different approaches and are therefore called *hybrid speech recognition systems*.

Feature extraction

The first problem for any automatic speech recognizer is finding an appropriate representation of the speech signal. Assume that the speech is sampled at constant intervals and denote the amplitude of the speech signal by $x(0), x(2), \ldots, x(n-1)$. For good recognition the time between consecutive measurements should be kept small. The microphone signal is thus a more or less adequate representation of speech but contains a lot of redundancy. It would be preferable to reduce the number of data points in such a way as to preserve most of the information: this is the task of all *feature extraction methods*. Choosing an appropriate method implies considering the speech production process and what kind of information is encoded in the acoustic signal.

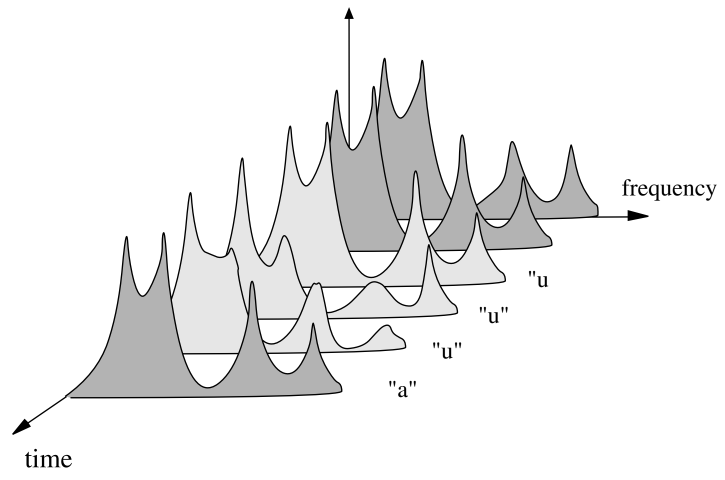


Fig. 9.17. Temporal variation of the spectrum of the speech signal

Speech is produced in the vocal tract, which can be modeled as a tube of varying diameter extending from the vocal chords to the lips. The vocal chords produce a periodic pressure wave which travels along the vocal tract until the energy it contains is released through the mouth and nose. The vocal tract behaves as a *resonator* in which some frequencies are amplified whereas others are eliminated from the final speech signal. Different configurations of the vocal organs produce different resonating frequencies, so that it is safe to assume that detecting the mixture of frequencies present in the speech signal can provide us with information about the particular configuration of the vocal tract, and from this configuration we can try to deduce what phoneme has been produced.

Figure 9.17 shows a temporal sequence of stylized spectra. The first spectrum, for example, corresponds to the vowel "a". There are four fairly distinct resonance maxima. They are called the *formants* of the speech signal. Each phoneme has a distinctive formant signature and if we could identify the sequence of formant mixtures we could, in principle, decode the speech signal.

Many methods have been proposed to deal with the task of spectral analysis of speech. Some of them have a psychophysical foundation, that is, they are based on physiological research on human hearing [353]. Others have arisen in other fields of engineering but have proved to be adequate for this task. Certainly one of the simplest, but also more powerful, approaches is computing a short-term Fourier spectrum of the speech signal.

Fourier analysis

Given a data set $\mathbf{x} = (x(0), x(2), \ldots, x(n-1))$ it is the task of Fourier analysis to reveal its periodic structure. We can think of the data set as function X

evaluated at the points $0, 2, \ldots, n-1$. The function X can be written as a linear combination of the basis functions

$$
\begin{aligned}
f_0(t) &= \frac{1}{\sqrt{n}}\left(\cos\left(2\pi t\frac{0}{n}\right) - i\sin\left(2\pi t\frac{0}{n}\right)\right) = \frac{1}{\sqrt{n}}(\omega_n^*)^{0\cdot t}\\
f_1(t) &= \frac{1}{\sqrt{n}}\left(\cos\left(2\pi t\frac{1}{n}\right) - i\sin\left(2\pi t\frac{1}{n}\right)\right) = \frac{1}{\sqrt{n}}(\omega_n^*)^{1\cdot t}\\
\vdots\ &\ \ \vdots\\
f_{n-1}(t) &= \frac{1}{\sqrt{n}}\left(\cos\left(2\pi t\frac{n-1}{n}\right) - i\sin\left(2\pi t\frac{n-1}{n}\right)\right) = \frac{1}{\sqrt{n}}(\omega_n^*)^{(n-1)\cdot t}
\end{aligned}
$$

where ω_n denotes the n-th complex root of unity $\omega_n = \exp(2\pi i/n)$ and ω_n^* its complex conjugate. Writing the data set as a linear combination of these functions amounts to finding which of the given frequencies are present in the data. Denote by $\mathbf{F}_n^*$ the $n \times n$ matrix whose columns are the basis functions evaluated at $t = 0, 1, \ldots, n-1$, that is, the element at row i and column j of $\mathbf{F}_n^*$ is $(\omega_n^*)^{ij}/\sqrt{n}$, for $i, j = 0, \ldots, n-1$. We are looking for a vector $\mathbf{a}$ of amplitudes such that

$$\mathbf{F}_n^*\mathbf{a} = \mathbf{x}.$$

The n-dimensional vector $\mathbf{a}$ is the *spectrum* of the speech signal. The matrix $\mathbf{F}_n$ defined as

$$
\mathbf{F}_n = \frac{1}{\sqrt{n}}\begin{pmatrix}
\omega_n^0 & \omega_n^0 & \cdots & \omega_n^0\\
\omega_n^0 & \omega_n^1 & \cdots & \omega_n^{n-1}\\
\omega_n^0 & \omega_n^2 & \cdots & \omega_n^{2n-2}\\
\vdots & & \ddots & \vdots\\
\omega_n^0 & \omega_n^{n-1} & \cdots & \omega_n^{(n-1)(n-1)}
\end{pmatrix}
$$

is the transpose conjugate of the matrix $\mathbf{F}_n^*$. Since the basis functions $f_0, \ldots, f_{n-1}$ are mutually orthogonal, this means that $\mathbf{F}_n^*$ is unitary and in this case

$$\mathbf{F}_n\mathbf{F}_n^*\mathbf{a} = \mathbf{F}_n\mathbf{x} \qquad \Rightarrow \qquad \mathbf{a} = \mathbf{F}_n\mathbf{x}.$$

The expression $\mathbf{F}_n\mathbf{x}$ is the discrete Fourier transform of the vector $\mathbf{x}$. The inverse Fourier transform is given of course by

$$\mathbf{F}_n^*\mathbf{F}_n\mathbf{x} = \mathbf{F}_n^*\mathbf{a} \qquad \Rightarrow \qquad \mathbf{x} = \mathbf{F}_n^*\mathbf{a}.$$

The speech signal is analyzed as follows: a window of length n (n data samples) is used to select the data. Such a window can cover, for example, 10 milliseconds of speech. The Fourier transform is computed and the magnitudes of the spectral amplitudes (the absolute values of the elements of the vector $\mathbf{a}$) are stored. The window is displaced to cover the next set of n data points and the new Fourier transform is computed. In this way we get a sequence of short-term spectra of the speech signal as a function of time, as shown in Figure 9.17. Since each articulation has a characteristic spectrum, our speech recognition algorithms should recover from this kind of information the correct sequence of phonemes.

Fast transformations

Since we are interested in analyzing the speech signal in real time it is important to reduce the number of numerical operations needed. A Fourier transform computed as a matrix-vector multiplication requires around $O(n^3)$ multiplications. A better alternative is the Fast Fourier transform, which is just a rearrangement of the matrix-vector multiplication. The left graphic in Figure 9.18 shows the real part of the elements of the Fourier matrix $\mathbf{F}_n$ (the shading is proportional to the numerical value). The recursive structure of the matrix is not immediately evident, but if the even columns are permuted to the left side of the matrix and the odd columns to the right, the new matrix structure is the one shown on the right graphic in Figure 9.18. Now the recursive structure is visible. The matrix $\mathbf{F}_n$ consists of four submatrices of dimension $n/2 \times n/2$, which are related to the matrix $\mathbf{F}_{n/2}$ through a simple formula. In order for the reduction process to work, n must be a power of two.

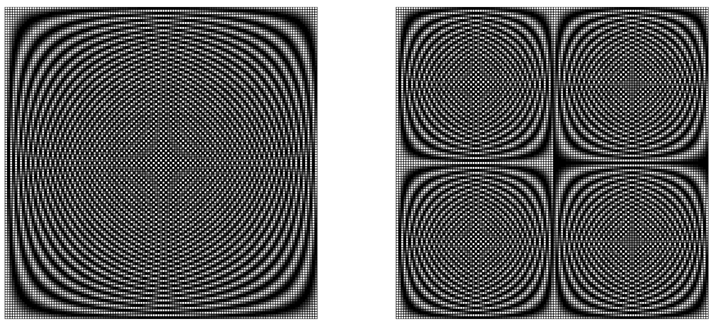


Fig. 9.18. The Fourier matrix and the permuted Fourier matrix

This rearrangement of the Fourier matrix is the basis of the Fast Fourier Transform (FFT) (See Exercise 3).

Many speech recognition systems use some kind of variation of the Fourier coefficients. The problem with short-term spectra is that the base frequency of the speaker should be separated from the medium-term information about the shape of the vocal tract. Two popular alternatives are cepstral coefficients and linear predictive coding (LPC) [353, 106].

Training of the classifier

Neural networks are used as classifier networks to compute the probability that any of a given set of phonemes could correspond to a given spectrum and the context of the spectrum. The speech signal is divided into frames of, for example, 10 ms length. For each frame the short-term spectrum or cepstrum is computed and quantized using 18 coefficients. We can train a network to associate spectra with the probability that each phoneme is present in a speech segment. A classifier network like the one shown in Figure 9.20 is used. The coefficients of the six previous and also of the six following frames are used together with the coefficients of the frame we are evaluating. The dimension of the input vector is thus 234. If we consider 61 possible phonemes we end up with the network of Figure 9.20, which is in fact very similar to NETtalk.

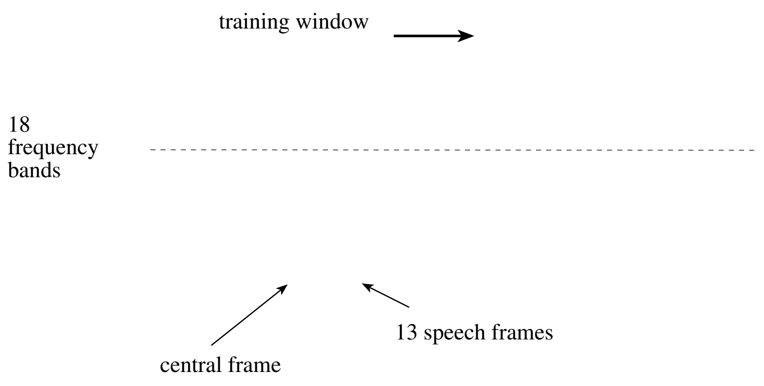


Fig. 9.19. Training window for the neural network

The network is trained with labeled speech data. There are several data bases which can be used for this purpose, but also semiautomatic methods for speech labeling can be used [65]. Once the network has been trained it can be used to compute the emission probabilities of phonemes.

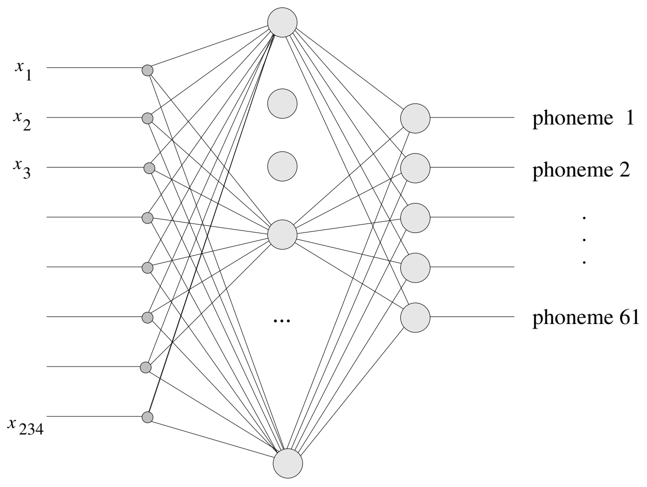


Fig. 9.20. Classification network for 61 phonemes

Hidden Markov Models

In speech recognition researchers postulate that the vocal tract shapes can be quantized into a discrete set of states roughly associated with the phonemes that compose speech. But when speech is recorded the exact transitions in the vocal tract cannot be observed and only the produced sound can be measured at some predefined time intervals. These are the *emissions*, and the *states* of the system are the quantized configurations of the vocal tract. From the measurements we want to infer the sequence of states of the vocal tract, i.e., the sequence of utterances which gave rise to the recorded sounds. In order to make this problem manageable, the set of states and the set of possible sound parameters are quantized.

A first-order Markov model is any system capable of assuming one of n different states at time t. The system does not change its state at each time step deterministically but according to a stochastic dynamic. The probability of transition from the i-th to the j-th state at each step is given by $0 \leq p_{ij} \leq 1$ and does not depend on the previous history of transitions. We also assume that at each step the model emits one of m possible output values. We call the probability of emitting the k-th output value $\mathbf{x}$ while in the i-th state $b_{ik} = P(\mathbf{x}_k|s_i)$. Starting from a definite state at time $t = 0$, the system is allowed to run for T time units and the generated outputs are recorded. Each new run of the system produces in general a different sequence of output values.

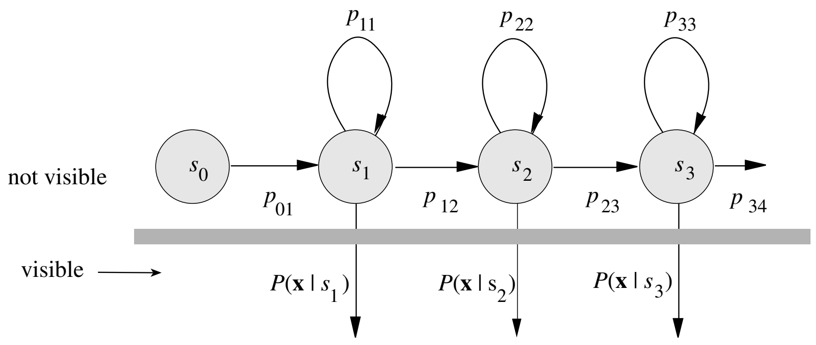


Fig. 9.21. A Hidden Markov Model

A Hidden Markov Model has the structure shown in Figure 9.21. The state transitions remain invisible for the observer (we cannot see the configuration of the vocal tract). The only data provided are the emissions (i.e., the spectrum of the signal) at some points in time. Figure 9.21 represents a model with four states, linked in such a way that we have sequential transitions. This model could represent the vocalization of a word. Each of the states s_i is a phoneme. Note that there is a probability p_{ii} that a phoneme state is repeated. This represents the case in which a speaker pronounces a word more slowly. In general, the word models constructed are more complicated than this. Especially in the case of very common words, we need more structure in the Markov model, as shown in Figure 9.22 which is a HMM for the word “and” [461]. The labeling of the nodes corresponds to the standard phonetic denomination of the relevant phonemes for this example.

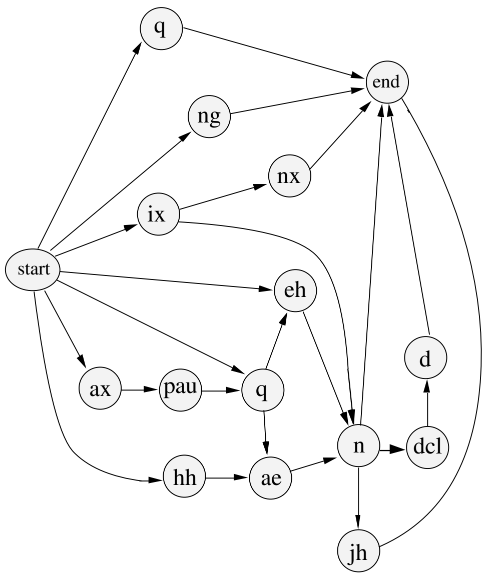


Fig. 9.22. Markov chain for the word “and”

The general problem we have when confronted with the recorded sequence of output values of a HMM is to compute the most probable sequence of state transitions which could have produced them. But first of all, we have to train the model, that is, compute the transition and emission probabilities. We discussed in Sect. 7.4.2 how this can be done applying the backpropagation algorithm.

Computation of the most probable path

Once a set of emission probabilities has been computed for several time frames $1, 2 \ldots, m$ it is necessary to compute the most probable path of transitions of the vocal tract and emissions. This can be done using dynamic programming methods of the same type as those generically known as *time warping.*

The general method is the following: the trained classifier network is applied to the speech data and for every time frame we obtain from the network the a posteriori probability of 61 phonemes. Figure 9.23 shows, for example, that for $t = 1$, that is for the first frame, the probability of having detected phoneme 1 is 0.1, for phoneme 2 it is 0.7, etc. For the second frame ($t = 2$) we get another set of 61 a posteriori probabilities and so on. We are looking for the path connecting the true sequence of produced sounds (the shaded portions of the table). The probability of any path is given by the product of the a posteriori probabilities of the phoneme sequence and the probability of transitions between phonemes. Denote by $p_a^{(t)}$ the a posteriori probability of phoneme a at time t and by $a_{i,j}$ the transition probability from phoneme i to phoneme j. Given any sequence of phonemes $k_1, k_2, \ldots, k_m$ the probability P of this special sequence is given by

$$P = p_{k_1}^{(1)} a_{k_1,k_2} p_{k_2}^{(2)} a_{k_2,k_3} \cdots p_{k_m}^{(m)} .$$

The transition probabilities are taken from the trained HMM for a word (in this case we are doing isolated word recognition). We pick the sequence of transitions with the greatest probability P and record it. The same procedure is repeated for all words in the vocabulary and the word with the greatest associated probability is selected as result of the recognition process.

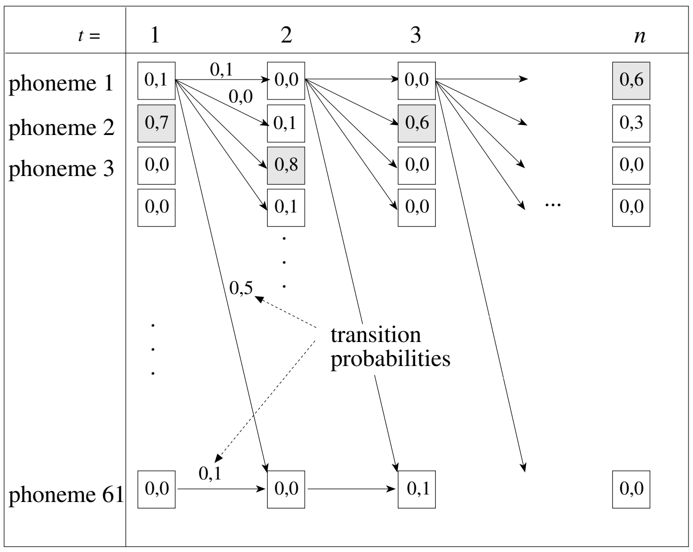


Fig. 9.23. Determination of the most probable path

The method used for the computation of the optimal path is *dynamic programming* [12]. Nevertheless it should be mentioned that one problem with this approach is that the long products of probabilities sometimes produce very small values which are difficult to discriminate. See [65] for an in-depth discussion of the pitfalls associated with speech recognition based on neural models.

9.3.4 Autoregressive models for time series analysis

It has been always an important issue to develop good forecasting techniques for time series in economics and statistics. A stochastic variable X produces a sequence of observations $x_1, x_2, \ldots, x_t$ at t different points in time that can be used to forecast the value of the variable at time $t+1$. If there is a functional relation between the successive values of the stochastic variable, we can try to formulate a linear or nonlinear model of the time series. Usually, linear models have been favored because of the accumulated experience and existing literature.

The general approach used in the neural networks field is to use historical values of the time series to train the network and test it with new values. Assume that the network has 8 input sites and one output. We can use the values $x_1, x_2, \ldots, x_8$ to forecast the value x_9. Sliding the training window one step at a time we can extract $n-8$ training examples from a time series with n data points. The network learns to forecast x_t using $x_{t-8}, \ldots, x_{t-1}$ as input. After several training steps we can measure how well the network has learned to forecast the future [291, 446].

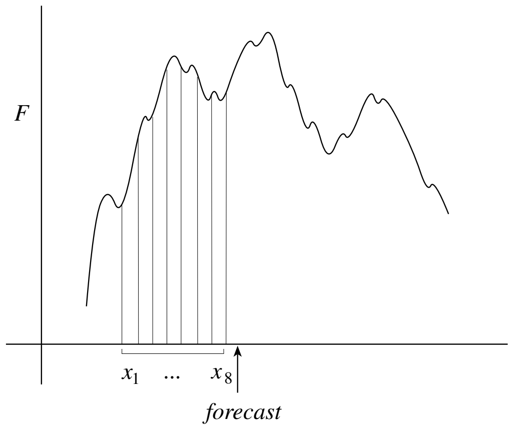


Fig. 9.24. Time series and a training window

This technique corresponds to the *autoregression models* popular among statisticians [86]. The desired approximation is of the type

$$x_t = \alpha_0 + \alpha_1 x_{t-1} + \alpha_2 x_{t-2} + \cdots + \alpha_p x_{t-p} + Z_t$$

where $\alpha_1, \ldots, \alpha_p$ are constants and Z_t a stochastic variable. This is a linear model of the kind considered before. Note that in this case we are predicting only one value into the future.

If we want to predict more than one step into the future, we can use the schema

$$x_{t+q} = \alpha_0 + (\beta_0 x_t + \cdots + \beta_{q-1} x_{t+q-1}) + (\alpha_1 x_{t-1} + \cdots + \alpha_p x_{t-p}),$$

where we use the result of previous predictions (x_t to x_{t+q-1}) in the forecast generated for step $t+q$. The system has now a built-in feedback which complicates the numerical solution. The well-known ARMA models (autoregressive moving average) have this structure.

Since nonlinear models are more general than the linear ones, we could expect that neural networks should lead to better forecasts. However, financial or other complicated time series are seldom easy to handle. Normally several statistical tests and different preprocessing techniques have to be applied before deciding on the best statistical forecasting method. In the case of economic time series, the number of degrees of freedom of the system is so large that search space has to be constrained in a decisive manner. In many cases, it is also almost impossible to base a forecast on the time series alone. If we want to forecast stock market prices, we have to consider other factors such as interest and inflation rates, foreign exchange situation, etc. It is not surprising that naive experiments where only a few parameters are considered cannot lead to successful forecasts [467].

The best results have been obtained with econometric models that relate several variables and functional dependencies. Some authors have coupled neural networks with expert systems in order to remove some of the uncertainties associated with simple-minded autoregressive models [53].

9.4 Historical and bibliographical remarks

The connection between neural networks and statistical models has spawned an active research community. Many new studies in this direction have been made since the pioneering investigations of the PDP group [421], but much work remains to be done. It should always be emphasized that feed-forward networks are a method of function approximation that must be applied carefully and with the necessary expertise from the problem domain. There are many negative examples of the kind of forecasting errors that poorly applied neural methods can produce [99]

The bootstrap was introduced by Efron [125], but similar ideas had already been proposed several years earlier in the Monte Carlo literature of hypothesis testing and by researchers trying to compute better confidence intervals. The method received its name because it models an unknown probability distribution by pulling on its own bootstraps, i.e., by resampling the given data set. A good introduction to the bootstrap method and some of its applications is [126]. Tukey [434] studied the properties of the jackknife method and gave it its name. Cross-validation has been used in many different contexts since the 1970s and some authors have investigated its usefulness in model selection.

There are now so many applications of backpropagation networks that even just mentioning the more significant would take too much space. NETtalk was one of the first examples of how a connectionist system could reach the proficiency of a rule-based system with much less effort and fewer assumptions. Other systems similar to NETtalk have been built to deal with cognitive problems like the association of visual and semantic cues. By damaging part of the network disorders such as dyslexia can be modeled. Plaut and Shallice have called this the “neurophysiology” of neural networks [345]. Another classical application is *Neurogammon*, a program that can play backgammon at the master’s level. This program was the first *learning* system that could win a computer tournament. This was significant because the self-organizing neural network was able to defeat rule-based systems with a large amount of invested design work. The new version of the program, called TD-Gammon, is even better [426]. It is trained using the method of *temporal differences*, which is an especially powerful approach for nondeterministic games.

It has been an old dream of a fraction of the neural network community to apply neural networks for the forecasting of financial futures. Some banks have started projects to compare the new with the traditional methods. The published results are somewhat contradictory, because in many cases the experiments are performed off-line, that is without actual trading. Under such circumstances a high-risk approach can sometimes produce impressive results which would otherwise be forbidden under realistic conditions [245]. More disturbing is the fact that if a good forecasting technique finds its way into the real market, the whole exercise can become self-defeating. If *everybody*, or at least a significant part of the market actors, can predict the future and try to cash on this knowledge, the market will move to a new equilibrium where nobody can profit from the others. This makes the neural system of Odom and Sharda [330] the more interesting, since it can predict future bankruptcies, maybe even of one’s own company. More interesting results were obtained by the networks submitted to the time series competition hosted by the Santa Fe Institute during 1992 [441]. Some neural systems were able to provide good forecasts for a wide range of time series taken from synthetic and real-world problems.

Exercises

1. Show that the mean $\bar{x}$ of n real numbers $x_1, x_2, \ldots, x_n$ is also the expected value of the mean of N bootstrap samples.
2. Train a feed-forward network to approximate a polynomial using bootstrap samples of the training set. Make a graph of the different network functions. Can you compute the confidence intervals of the functional approximation?
3. The non-symmetric discrete Fourier transform is defined using the matrix

$$\mathbf{F}_n = \begin{pmatrix} \omega_n^0 & \omega_n^0 & \cdots & \omega_n^0 \\ \omega_n^0 & \omega_n^1 & \cdots & \omega_n^{n-1} \\ \omega_n^0 & \omega_n^2 & \cdots & \omega_n^{2n-2} \\ \vdots & & \ddots & \vdots \\ \omega_n^0 & \omega_n^{n-1} & \cdots & \omega_n^{(n-1)(n-1)} \end{pmatrix}$$

where ω_n denotes the n-th complex root of unity (n a power of two). Show that $\mathbf{F}_n$ can be written as

$$\mathbf{F}_n = \begin{pmatrix} \mathbf{F}_{n/2} & \mathbf{D}\mathbf{F}_{n/2} \\ \mathbf{F}_{n/2} & -\mathbf{D}\mathbf{F}_{n/2} \end{pmatrix}$$

where $\mathbf{D}$ is a diagonal matrix. Derive from this result the FFT algorithm.

4. Train a network that can make a one-step prediction of a synthetic time series. Generate the data using a sum of several sinusoidal functions with different frequencies, phases, and amplitudes.