Lesson 8

Regression vs. Progression

In the earlier chapters, methods were discussed that are based on "traditional" statistical approaches. On the other hand, lately the so called *soft computing* methods have become popular, seemingly shadowing the more traditional modeling methods. In this chapter, the new methodologies are briefly discussed exclusively focusing on *neural networks.* This paradigm consists of a wide variety of different approaches and methods, but there are some common features — the methods are data-based, iterative, and massively parallel. And, what is more, the intended applications are typically extremely complex and nonlinear (see [5], [13], [7], and [36]).

However, neural networks and statistical methods are not competing methodologies for data analysis, and the aim of this chapter is to discuss the connections between these two seemingly very different approaches. It is shown how understanding the (linear) statistical phenomena help in understanding the operation of the more complex algorithms — it is statistical properties between data that there only exist, no matter what is the analysis method, after all.

Neural networks research is a quite diverse field of methods originating from different kinds of intuitions. Three different branches of neural networks research are discussed here separately: The first branch is unsupervised neural clustering methods and regression based on them, and the second branch is perceptron networks and regression. Finally, it is shown how the statistical methods are not only related to artificial neural networks but also to natural neuron structures.

8.1 Neural clustering

For practically any statistical data processing method, there exist a neural counterpart. When clustering, for example, is done using the neural networks algorithms, there are typically some benefits: The robustness of the clustering process can be enhanced, and as a bonus, some kind of "topology" can be found between clusteres, making it easier to gain intuition about the data properties. But, on the other hand, there are drawbacks, like the longer execution time of the algorithms.

8.1.1 Self-organizing maps

The celebrated *Self-Organizing Map (SOM)* algorithm by Teuvo Kohonen [27] performs nonlinear dimensonality reduction using *competitive learning.* It accomplishes data clustering in such a way that the topology of the input space is somehow preserved, that is, nearby data in the input space are mapped into clusters (now called "nodes") that are near each other also in the low-dimensional grid[1].

The self-organizing map consists of N nodes, each of which is characterized by an n dimensional prototype vector $\bar{x}^c$, where $1 \leq c \leq N$, standing for the cluster centers. The batch SOM algorithm that iteratively organizes the map can be expressed as follows:

1. Choose a set of original node prototypes $\bar{x}^1, \ldots, \bar{x}^N$ arbitrarily.
2. Assign the k samples to the N nodes using the minimum distance rule, that is, sample $x(\kappa)$ belongs to node $c(\kappa)$ such that
$$\bar{c}(\kappa) = \operatorname*{argmin}_{c \in [1,N]} (x(\kappa) - \bar{x}^c)^T (x(\kappa) - \bar{x}^c). \tag{8.1}$$
3. For all pairs of center $\bar{c}(\kappa)$ and node c calculate the "distance measure" (for explanation of the parameters, see below):
$$h_c(\kappa) = \exp\left(- \frac{d^2(c, \bar{c}(\kappa))}{2\sigma^2(\kappa)} \right). \tag{8.2}$$
4. Compute new node prototypes, that is, for all $1 \leq c \leq N$:
$$\bar{x}^c \leftarrow \sum_{\kappa=1}^{k} h_c(\kappa) \cdot x(\kappa) / \sum_{\kappa=1}^{k} h_c(\kappa). \tag{8.3}$$
5. If any of the node prototypes changes, return to step 2, otherwise, stop.

Above, in Step 4, the value of the *neighborhood parameter* $h_c(\kappa)$ determines the behavior of the adaptation process. This parameter determines the net topology, giving large values if the node c and the best matching node $\bar{c}(\kappa)$ are "near" each other in the net, and smaller values otherwise. In (8.2) the parameter $d(c, \bar{c}(\kappa))$ gives the distance between nodes c and $\bar{c}(\kappa)$ in the grid of network neurons, and $\sigma(\kappa)$ determines the "width" of the neighborhood. This parameter can be time-varying, starting from a relatively large value, but getting smaller (the neighborhood "shrinking") as the adaptation continues, making the adaptation more local. When the algorithm has converged, the node prototypes $\bar{x}^c$ contain the cluster centers being arranged within a grid structure.

The selection (8.2) for $h_c(\kappa)$ gives a Gaussian form for the neighborhood effect. As shown in [18], this parameter can be interpreted as probability for a sample to belong to a specific node; that is, the net can be interpreted again as a Gaussian mixture model for the data, and the algorithm tries to adjust the Gaussian centers to best match the data.

[1]Now we are not specially interested in the mapping, or the visual properties of the data, but on the clusters in the input space as generated by the algorithm. The topological ordering is reached as a side-effect

8.1.2 “Expectation Maximizing SOM”

The SOM algorithm is related to the K-means clustering, because in both cases the Gaussian distributions are assumed to have identity covariance matrix. As in the case of the EM algorithm, the above algorithm can also be extended to “EMSOM”:

1. Choose a set of original node prototypes $\bar{x}^1, \ldots, \bar{x}^N$ arbitrarily, for example, using the SOM algorithm; the cluster covariances are originally identity matrices, that is, $R^c = I$ for all $1 \leq c \leq N$.
2. Assign the k samples to the N nodes using the minimum (balanced) Mahalanobis distance rule:
$$\bar{c}(\kappa) = \underset{c \in [1,N]}{\operatorname{argmin}} \ln(\det\{R^c\}) + (x(\kappa) - \bar{x}^c)^T (R^c)^{-1} (x(\kappa) - \bar{x}^c). \quad (8.4)$$
3. Calculate the neighborhoods for all node/sample pairs according to
$$h_c(\kappa) = \exp\left(-\frac{d^2(c, \bar{c}(\kappa))}{2\sigma^2(\kappa)}\right). \quad (8.5)$$
4. Compute new node prototypes, that is, for all $1 \leq c \leq N$:
$$\bar{x}^c \leftarrow \sum_{\kappa=1}^{k} h_c(\kappa) \cdot x(\kappa) / \sum_{\kappa=1}^{k} h_c(\kappa). \quad (8.6)$$
5. Correspondingly, update covariance estimates
$$R^c \leftarrow \sum_{\kappa=1}^{k} h_c(\kappa) \cdot (x(\kappa) - \bar{x}^c)(x(\kappa) - \bar{x}^c)^T / \sum_{\kappa=1}^{k} h_c(\kappa). \quad (8.7)$$
6. If any node prototypes changes, return to step 2, otherwise, stop.

The algorithm can be stabilized by introducing some gradual forgetting in (8.7). Note that if the number of clusters N is high, this algorithm typically behaves better than the original EM algorithm: The R^c matrices do not become singular as easily.

When doing neural networks modeling, it is typical that very little is assumed about the data. In Chapter 2, it was (optimistically) assumed that the data distributions can be expressed as combinations of Gaussian subdistributions. When doing data modeling with SOM-type algorithms, the philosophy is very different: All assumptions about the underlying distribution structure are discarded, one just tries to capture the data density as exactly as possible. The Gaussian formulas that are used in the self-organization algorithms are used only as *basis functions* for spanning the observed data density, they do not stand for separately distinguishable clusters. That is why the number of nodes in SOM does not usually match the number of real clusters in data (if there exists some) but is considerably higher, each condensation of data being represented possibly by various nodes. This generity makes it possible to model very complex data distributions having no *a priori* information about the nature of the distributions.

8.1.3 Radial basis function regression

It turns out that nonlinear regression can be realized directly based on the clustered data model[2]. This kind of regression methods are studied under the name of *radial basis function networks.*

For example, one can assume that if the sample is explained exclusively by some specific subdistribution c, the correct (constant) output vector is $\bar{Y}^c$. Limiting the y values to one constant in a distribution sounds to be a harsh limitation — but, as can be seen later, this is not true for a mixture model (see Fig. 8.1). When the distributions are weighted in a reasonable way, according to their probabilities of explaining the measured sample, continuity is achieved.

To apply basis function regression, one first has to find the probabilities for a sample x being represented by a specific subdistribution. The estimated value for the output is (in a maximum likelihood sense) a weighted sum of the candidate outputs; these weighting parameters are the probabilities of the corresponding subdistributions, so that

$$\hat{y} = \sum_{c=1}^{N} \bar{p}_c(x) \cdot \bar{Y}^c, \tag{8.8}$$

where the normalized probabilities are calculated as

$$\bar{p}_c(x) = \frac{p_c(x)}{\sum_{c'=1}^{N} p_{c'}(x)} \tag{8.9}$$

with the individual densities being determined as Gaussian distributions

$$p_c(x) = \frac{1}{\sqrt{(2\pi)^n \det\{R^c\}}} \cdot \mathrm{e}^{-\frac{1}{2}\cdot(x-\bar{x}^c)^T (R^c)^{-1} (x-\bar{x}^c)}. \tag{8.10}$$

If using K-means or SOM for clustering, the covariance is $R^c = \sigma^2 \cdot I$ for all $1 \le c \le N$. If there are k samples $x(\kappa)$ and $y(\kappa)$ available for constructing the model, one has to optimize the values $\bar{Y}^c$ to fit the regression curve with the observations X and Y. The above normalized probabilities can be collected (in the familiar way) into the $k \times N$ matrix $\bar{P}(X)$, and the optimal prototype outputs for the clusters can be calculated in the MLR style as

$$\bar{Y} = \left(\bar{P}^T(X)\bar{P}(X)\right)^{-1} \bar{P}^T(X) \cdot Y, \tag{8.11}$$

so that the final nonlinear regression model becomes

$$\begin{aligned} \hat{Y}_{\text{est}} &= \bar{P}(X_{\text{est}}) \cdot \bar{Y} \\ &= \bar{P}(X_{\text{est}}) \cdot \left(\bar{P}^T(X)\bar{P}(X)\right)^{-1} \bar{P}^T(X)Y. \end{aligned} \tag{8.12}$$

[2]Note that we are assuming that the data density is now represented by (ovelapping) Gaussian distributions, no matter whether there really exist separate clusters or not

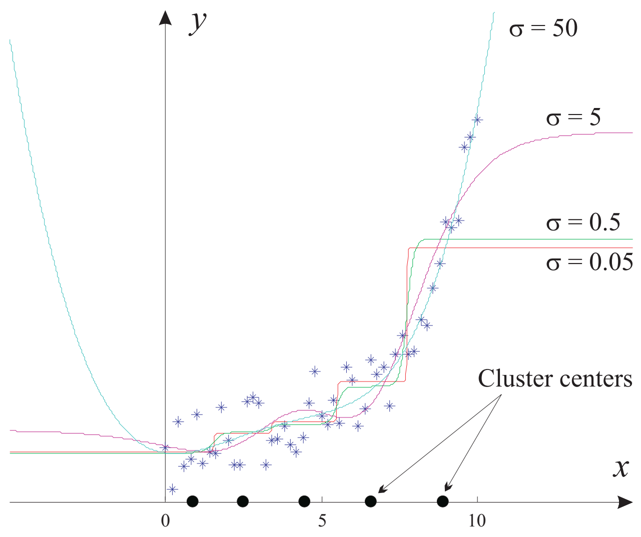


Figure 8.1: The effects of different basis distribution sizes in one dimension. For very narrow distributions, the regression curve becomes stepwise, representing only N distinct values, one for each cluster; when the distribution becomes wider, the curve becomes smoother, but abrupt changes cannot any more be modeled

$$\begin{tikzcd} X \arrow[r, "g^1 \circ F^1"] & Z^1 \arrow[r, "g^2 \circ F^2"] & \circ\circ\circ \arrow[r, "g^{N-1} \circ F^{N-1}"] & Z^{N-1} \arrow[r, "g^N \circ F^N"] & Y \end{tikzcd}$$

Figure 8.2: Introducing more and more latent variables ...!

8.2 Feedforward networks

This far, we have been mainly interested in linear models, assuming that nonlinearities can be circumvented by appropriate preprocessing of data, clustering or variable selection. However, this is not always enough, specially if the system structure is unknown, and more general methods may sometimes be needed. Nonlinear regression

$$y = g(x) \tag{8.13}$$

can be accomplished in a variety of ways. The key question is: How to parameterize the function g?

As compared to linear models, the nonlinear regression problem is much more complex. First, there is the model structure selection problem, and even if the type of the nonlinearity has been determined, the algorithms for finding the best parameters are complicated. Only iterative methods exist. The *multilayer feedforward perceptron networks (MLP's)* are taken here as a prototype of nonlinear regression models. The overall "layered" MLP regression structure is depicted in Fig. 8.2.

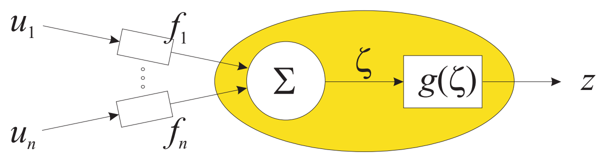


Figure 8.3: A single perceptron: $z = g(\zeta) = g(\sum_{j=1}^{n} f_j u_j)$

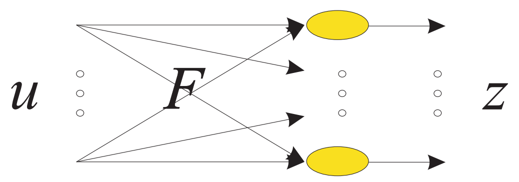


Figure 8.4: A layer of perceptrons: $z = g(F^T u)$, now $g : \mathcal{R}^{n_1} \to \mathcal{R}^{n_2}$

8.2.1 Perceptron networks

Within the neural networks paradigm, it is customary to construct the overall nonlinearity from simple nonlinear units called *perceptrons* (see Fig. 8.3). These perceptrons are independent computing elements trying to mimic the information processing of the biological nerves. Various outputs can be realized when more perceptrons are used in a layer; more sophisticated functions can be implemented by connecting several layers after each other (Figs. 8.4 and 8.5).

As shown in Fig. 8.6, MLP's are general-purpose, flexible, nonlinear models that, given enough hidden neurons and enough data, can approximate virtually any function to any desired degree of accuracy. In other words, MLP's are universal approximators. MLP's can be used when there exists little *a priori* knowledge about the form of the relationship between the input and output variables. MLP's are especially useful because the complexity of the model can be varied easily.

However, if no assumptions are made about the structure of nonlinearity, there are too many degrees of freedom to fix the model using some training data (remember the "Flatland"!). That is why, in practice, the assumption about function *smoothness* is made. This assumption is not well suited for modeling functions with abrupt changes.

8.2.2 Back-propagation of errors

The original multilayer perceptron training method was *back-propagation of errors.* The basic backpropagation algorithm is a gradient method, where the weights are adapted in the negative error gradient direction, thus being relatively inefficient. Later, various enhancements have been proposed, but in this context only the original version is presented. The training algorithm can be divided in two parts, forward regression and backward adaptation:

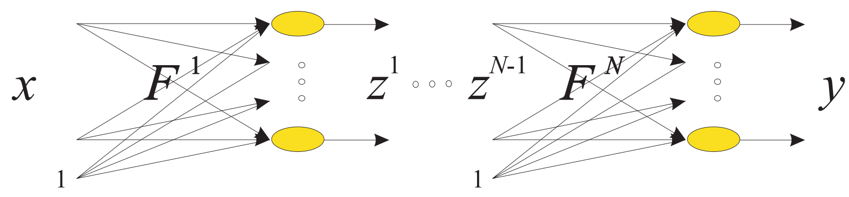


Figure 8.5: The complete perceptron network (note the additional "bias input" in each layer

1. **Neural regression:** For each neuron layer i, where $1 \leq i \leq N$, apply the following:

   (a) Augment the input (note that $\hat{z}^0(\kappa) = x(\kappa)$):
   $$u^i(\kappa) = \left(\frac{\hat{z}_{i-1}(\kappa)}{1} \right). \tag{8.14}$$

   (b) Calculate the weighted sum of the inputs:
   $$\zeta^i(\kappa) = (F^i)^T \cdot u^i(\kappa). \tag{8.15}$$

   (c) Apply the ouput function for all $1 \leq j \leq N_i$:
   $$\hat{z}^i_j(\kappa) = g^i\left(\zeta^i_j(\kappa)\right). \tag{8.16}$$

2. **Error back-propagation:** If the weights are to be adapted, assuming that the "correct" output $y(\kappa)$ is available, do the following:

   (a) Calculate the error $e^i(\kappa) = z^i(\kappa) - \hat{z}^i(\kappa)$. Only for the last layer this can be carried out explicitly, because $z^N(\kappa) = y(\kappa)$ is known. All other errors can only be approximated; a heuristic approach is to forget about the nonlinearities, etc., and back-propagate the error from the outer level, assuming that the appropriate "error distribution" between the inner level neurons is determined by the weighting matrix F^i:
   $$e^{i-1}(\kappa) = (F^i)^T \cdot e^i(\kappa). \tag{8.17}$$

   (b) Calculate the error gradients for all $1 \leq i \leq N$ and $1 \leq j \leq N_i$ (that is, i is the layer index, and j is the neuron index within a layer). The idea is to calculate the effect of one output at a time on all of the inputs:
   $$\begin{aligned}
   \frac{d(e^i_j)^2}{dF^i_j}(\kappa) &= \frac{d(z^i_j - \hat{z}^i_j)^2}{dF^i_j}(\kappa) \\
   &= -2e^i_j(\kappa) \cdot \frac{d\hat{z}^i_j}{dF^i_j}(\kappa) \\
   &= -2e^i_j(\kappa) \cdot \frac{dg^i(\zeta^i_j)}{dF^i_j}(\kappa) \\
   &= -2e^i_j(\kappa) \cdot \frac{dg^i}{d\zeta}(\zeta^i_j(\kappa)) \cdot \frac{d\zeta^i_j}{dF^i_j}(\kappa) \\
   &= -2e^i_j(\kappa) \cdot \frac{dg^i}{d\zeta}(\zeta^i_j(\kappa)) \cdot \frac{d((F^i_j)^T u^i)}{dF^i_j}(\kappa) \\
   &= -2e^i_j(\kappa) \cdot \frac{dg^i}{d\zeta}(\zeta^i_j(\kappa)) \cdot u^i(\kappa).
   \end{aligned} \tag{8.18}$$

   (c) Update the parameters applying the gradient descent algorithm (γ being the step size). The whole column is updated simultaneously:
   $$F^i_j \quad \leftarrow \quad F^i_j - \gamma \cdot \frac{d(e^i_j)^2}{dF^i_j}(\kappa). \tag{8.19}$$

   This process of forward and backward propagation is repeated for different training samples until the network converges.

It needs to be noted that training a nonlinear neural network, or finding the values of the parameters in the matrices F^i, is a much more complicated thing than it is in the linear case. First, the results are very much dependent of the initialization of the parameters; and during training, there are the problems caused by local minima and neuron saturation effects, not to mention overfitting problems, etc.

Typically, the nonlinear perceptron activation function g^i is selected as hyperbolic tangent (or "tansig"):

$$g^i_{\text{tansig}}(\zeta) = \frac{2}{1+\mathrm{e}^{-a\zeta}} - 1, \tag{8.20}$$

so that the derivative becomes

$$\frac{dg^i_{\text{tansig}}}{d\zeta}(\zeta) = \frac{2a\mathrm{e}^{-a\zeta}}{\left(1+\mathrm{e}^{-a\zeta}\right)^2}, \tag{8.21}$$

where a is some constant (see Figs. 8.8 and 8.9). In some cases, linear neurons may be used[3]. It is reasonable to let at least the outermost layer have linear activation function, otherwise (in the case of g_{tansig}) the network output would be limited between $-1 < \hat{y}_j < 1$:

$$g^i_{\text{linear}}(\zeta) = a\zeta, \tag{8.22}$$

so that

$$\frac{dg^i_{\text{linear}}}{d\zeta}(\zeta) = a. \tag{8.23}$$

The selection of the number of hidden layer neurons is a delicate matter. Note that if there are N_i neurons on the previous level and N_j neurons on the next level, $N_i \cdot N_j$ free parameters are introduced (plus the additional bias term). Too many degrees of freedom may make the model useless (see Fig. 8.7); however, the results are very dependent of the training method, too.

8.2.3 Relations to subspace methods

Nonlinear versions of PCA and PLS can also be constructed using the feedforward perceptron network (see Fig. 8.10). The key point is that there is a layer i of relatively low dimension N_i, no matter how complicated layers there are before and after this layer. If the dependency between input and output can be compressed, these lower-dimensional hidden layer activations can be interpreted as latent variables.

[3]Note that successive linear layers can be "collapsed"; various linear layers do not expand the expressional power as compared to a single linear layer. If all g^i are linear, the whole network, no matter how complex, can be presented as a single matrix multiplication — and because the cost criterion is identical (minimization of squared error average) the result of training must be the same as in the case of linear regression!

It must be recognized that this kind of results are seldom unique, and the results are often difficult to interpret: It is no more only linear subspaces that are spanned. For example, see Fig. 8.11: It turns out that the functions being modeled are symmetric, so that $y_i = f_i(x) = f_i(\|x\|)$ for $i = 1, 2$, and this can efficiently be utilized in compression — see Fig. 8.12. Only the positive part needs to be modeled, the negative region being taken care of automatically because of the latent variable construction. The network for reaching appropriate behavior was as simple as $N_1 = 2$, $N_2 = 1$ (the latent layer), $N_3 = 2$, and $N_4 = 2$ (the output layer). It is no wonder that the algorithms often fail in this kind of complex tasks (see Exercise 2); typically the training becomes more and more complex as the number of the layers grows.

Again, if all neurons are linear, variables z span the same linear subspace as in normal PCA/PLS; this gives us a new, iterative way to determine the latent basis. In the linear case, again, only one layer of neurons is needed to achieve the required mappings, so that $z = (F^1)^T x$ and $y = (F^2)^T z$. However, it has to be recognized that the same ordering as in standard PCA/PLS cannot generally be reached: None of the hidden layer neurons start uniquely representing the first principal component, etc., and the latent variables are linear combinations of the actual PCA/PLS latent variables that would be derived using the methods presented in earlier chapters.

8.3 "Anthropomorphic models"

Of course, all artificial neural networks do have their underlying ideas in neurophysiology, the power of brains having boosted the interest. But typically it is only the network structure that is copied, the functional characteristics being simplified to extreme. However, long before the research on artificial neural networks started, some fundamental phenomena of the neural functions were noticed by Donald O. Hebb [14]:

> Neurons seem to adapt so that the synaptic connections become stronger if the neuronal activation and its input signals correlate.

This general idea of *Hebbian learning* has later become one of the basic paradigms in unsupervised learning where there is no external training.

8.3.1 Hebbian algorithms

In unsupervised learning the only thing that the adaptation algorithm can do is to try to find some statistical structure within the input signals. According to the Hebbian principle, it is correlation that should be maximized — this goal sounds distantly familiar, and as it will turn out, the results will also look familiar.

In the simplistic technical implementation, as compared to the earlier discussions, the Hebbian neuron is a *linear perceptron without bias,* that is, the activation (output) of the neuron can be expressed as

$$z(\kappa) = f^T \cdot x(\kappa). \tag{8.24}$$

The parameters in f are interpreted as synaptic weights connecting the neuron to other neurons. According to the Hebbian rule the change in the weights can be expressed as

$$\Delta f = \gamma \cdot z(\kappa)x(\kappa), \tag{8.25}$$

where $z(\kappa)x(\kappa)$ denotes the correlation between the neuronal activation and input, so that

$$f(\kappa+1) = f(\kappa) + \gamma \cdot z(\kappa)x(\kappa). \tag{8.26}$$

Here γ is a small adaptation factor. The change in the vector f is determined essentially by the match between input $x(\kappa)$ and the contents of the Hebbian neuron f.

One thing plaguing the extremely simplified linear Hebbian neuron is that above learning law (8.26) is not stable but boosts the parameters in f without limits. To enhance the basic Hebbian model, let us prevent f from exploding. A simple solution to this (as proposed by Erkki Oja) is to normalize the length of f to unity after each adaptation step. Assume that, to begin with, $\|f(\kappa)\| = 1$, and this normality is returned after each iteration as follows:

$$\begin{aligned} f(\kappa+1) &= \frac{f(\kappa) + \gamma \cdot z(\kappa)x(\kappa)}{\|f(\kappa) + \gamma \cdot z(\kappa)x(\kappa)\|} \\ &= \frac{f(\kappa) + \gamma \cdot z(\kappa)x(\kappa)}{\sqrt{\left(f(\kappa) + \gamma \cdot z(\kappa)x(\kappa)\right)^T \left(f(\kappa) + \gamma \cdot z(\kappa)x(\kappa)\right)}} \\ &= \frac{f(\kappa) + \gamma \cdot z(\kappa)x(\kappa)}{\sqrt{f^T(\kappa)f(\kappa) - 2\gamma \cdot z(\kappa)x^T(\kappa)f(\kappa) + \mathcal{O}\{\gamma^2\}}} \\ &= \frac{f(\kappa) + \gamma \cdot z(\kappa)x(\kappa)}{\sqrt{1 + 2\gamma \cdot z^2(\kappa) + \mathcal{O}\{\gamma^2\}}}. \end{aligned} \tag{8.27}$$

Assuming that γ is small, terms including powers of γ higher than two can be ignored. Here one can further approximate the square root by noticing that for small values of α there holds

$$\frac{1}{\sqrt{1+\alpha}} \approx 1 - \frac{1}{2} \cdot \alpha, \tag{8.28}$$

giving (again forgetting terms containing γ^2)

$$\begin{aligned} f(\kappa+1) &= \left(f(\kappa) + \gamma \cdot z(\kappa)x(\kappa)\right) \cdot \left(1 - \gamma \cdot z^2(\kappa)\right) \\ &\approx f(\kappa) + \gamma \cdot z(\kappa)x(\kappa) - \gamma \cdot z^2(\kappa)f(\kappa). \end{aligned} \tag{8.29}$$

The "stabilized" Hebbian algorithm also becomes

$$f(\kappa+1) = f(\kappa) + \gamma \cdot \left(z(\kappa)x(\kappa) - z^2(\kappa) \cdot f(\kappa)\right). \tag{8.30}$$

In addition to the nominal correlation-motivated factor $z(\kappa)x(\kappa)$, another non-linear term has emerged, preventing the algorithm from growing excessively.

Assuming that the algorithm converges to some fixed f (indeed, it does; for example, see [13]), this parameter change trend $\Delta f(\kappa) = f(\kappa+1) - f(\kappa)$ should vanish. One can study the properties of this fixed state by taking expectation values on both sides:

$$\begin{array}{rcl} \mathrm{E}\{\Delta f(\kappa)\} & = & \gamma \cdot \big(\mathrm{E}\{x(\kappa)z(\kappa)\} - \mathrm{E}\{z^2(\kappa)\} \cdot f\big) \\ & = & \gamma \cdot \big(\mathrm{E}\{x(\kappa)x^T(\kappa)\} \cdot f - \mathrm{E}\{z^2(\kappa)\} \cdot f\big) . \end{array} \quad (8.31)$$

It turns out that in the fixed state there must hold

$$\mathrm{E}\{x(\kappa)x^T(\kappa)\} \cdot f = \mathrm{E}\{z^2(\kappa)\} \cdot f, \quad (8.32)$$

where $\mathrm{E}\{x(\kappa)x^T(\kappa)\}$ is the data covariance matrix. This means that the above formula has the same structure as the PCA problem has, the eigenvector of the data covariance matrix being f and the eigenvalue being $\lambda_1 = \mathrm{E}\{z^2(\kappa)\} \neq 0$. It also turns out that the Hebbian algorithm converges to the principal component (the most significant one; see [13]), so that one can redefine $\theta_1 = f$. Further,

$$\begin{array}{rcl} \lambda_1 & = & \mathrm{E}\{z^2(\kappa)\} \\ & = & \mathrm{E}\{\theta_1^T x(\kappa)x^T(\kappa)\theta_1\} \\ & = & \theta_1^T \cdot \big(\mathrm{E}\{x(\kappa)x^T(\kappa)\} \cdot \theta_1\big) \\ & = & \theta_1^T \cdot \lambda_1 \cdot \theta_1 \\ & = & \lambda_1 \cdot \|\theta_1\|. \end{array} \quad (8.33)$$

When λ_1 is eliminated on both sides, it turns out that the eigenvector is automatically normalized by this Hebbian algorithm:

$$\|\theta_1\| = 1. \quad (8.34)$$

So, it is no wonder that correlation maximization results in the principal components of the data. What *is* interesting, is that this seems to be happening also in the brain!

8.3.2 Generalized Hebbian algorithms

Assume that the contribution of θ_1 is eliminated from the data, so that

$$x'(\kappa) \leftarrow z(\kappa) \cdot \theta_1. \quad (8.35)$$

Note that after this operation the modified input vector is orthogonal to θ_1:

$$\theta_1^T \cdot x'(\kappa) = \theta_1^T \cdot (x(\kappa) - z(\kappa) \cdot \theta_1) = z(\kappa) - z(\kappa) \cdot 1 = 0. \quad (8.36)$$

Applying the Hebbian algorithm using this modified $x'(\kappa)$ as input extracts the most significant principal component that is left after the elimination of the first principal component — that is, now f converges towards θ_2. This procedure can be continued, and the resulting *Generalized Hebbian Algorithm (GHA)* can be used to iteratively extract as many principal components that is needed; if the procedure is formalized as a N layer network, the algorithm to be applied for all $1 \leq i \leq N$ (note that $x^1(\kappa) = x(\kappa)$) becomes

$$\begin{array}{rcl} x^i(\kappa) & = & x^{i-1}(\kappa) - z^{i-1}(\kappa) \cdot f^{i-1}(\kappa) \qquad \text{for layers } i > 1 \\ z^i(\kappa) & = & (x^i)^T(\kappa) \cdot f^i(\kappa) \\ f^i(\kappa+1) & = & f^i(\kappa) + \gamma \cdot z^i(\kappa) \cdot \big(x^i(\kappa) - z^i(\kappa) \cdot f^i(\kappa)\big) . \end{array} \quad (8.37)$$

After the iteration has converged, the principal components are

$$\left(\ \theta_1 \mid \cdots \mid \theta_N\ \right) = \left(\ f^1 \mid \cdots \mid f^N\ \right). \tag{8.38}$$

Note that the algorithm (8.37) does probably no more have any connection to operations taking place in the brain; it is just an extension of the basic Hebbian idea for easily extracting the principal component structure from the data. The structures of the Hebbian algorithms are so simple that they can be easily implemented; there is no explicit reference to their neural background, and this is an example of how the new paradigms can give important contribution to other branches of research.

Note that when using the Hebbian algorithms, the covariance matrix never needs to be explicitly constructed — that is why, the Hebbian approach may be useful in specially high-dimensional data analysis tasks.

8.3.3 Further extensions

The generalized Hebbian algorithm can still be extended. For example, take the *anti-Hebbian learning,* where, in addition to maximizing the correlation with the inputs, the correlations with *other* outputs is minimized. The goal is to make the neurons as independent from each other as possible; as we have seen, this kind of independence often reveals underlying structure in the data. The explicit decorrelation between outputs results in sparse coding as shown in [8]. However, the correlation maximization/minimization structure is recursive, and the training algorithms are rather inefficient.

Another extension towards multiple mixture models is the *Generalized Generalized Hebbian Algorithm (GGHA)* [24]. The idea is to explicitly assume sparsity in the data; that is, there are various trains of candidate principal components, and only one of these candidate sequences is selected at a time (using the "best match" principle). When the selected components are eliminated from data, as in GHA, the rest is explained by the remaining components. This algorithm has been applied to a number of high-dimensional feature extraction problems.

8.4 Cybernetic neurons*

Cybernetics is a branch of complex systems research, where it is assumed that the observed complex functionalities can be explained in terms of interactions and feedbacks among the underlying local "agents". Specially, in *neocybernetics* the approaches are made very concrete: In the spirit of multivariate models, it is assumed that understanding of high dimensionality and dynamic structures can help in explaining the emergent functionalities. Furthermore, there are some very stringent assumptions: First, it is assumed that there is dynamic balance on all levels in a complex hierarchical system; second, model structures are kept as simple as possible — one could speak of *linearity pursuit.* Despite the constraints, it seems that non-trivial systems can be modeled in this way (see [?]).

These guidelines can be exploited on different levels of modeling the neuronal system, and analysis of a Hebbian neuron grid is carried out here. First, one can study the synaptic level: The stabilization of the synaptic weight can be accomplished not only applying nonlinearity, as is done when following Oja's rule, but also applying linear feedback. So, assume that instead of (14) one defines

$$\Delta f = \gamma \cdot z(\kappa)x(\kappa) - \alpha f \tag{8.39}$$

for some scalar α. It turns out that, assuming stationarity of the input, the synapse finds a stable value that is proportional to the correlation between the input and the neuronal activity. In the matrix form, the steady state of all synaptic weights can be expressed (β being some scalar) using an (unnormalized) correlation matrix

$$W = \beta \mathrm{E}\{zx^T\}. \tag{8.40}$$

To reach some added value, the neocybernetic intuitions can be applied also on the next level, or to the analysis of the whole neuron grid. Assume that the behavior of the grid of individual Hebbian neurons is orchestrated again by linear feedback, so that some of the synapses are between neurons — this means that one has a dynamic structure of the form

$$\frac{dz}{dt} = -Az + Bx. \tag{8.41}$$

Here, the matrices A and B contain the synaptic weights of W as divided according to their roles: synapses between inputs and neurons are collected in B, whereas the inter-neuronal connections are represented by the matrix A. In front of A there is the "$-$" sign to explicitly emphasize the negative feedback nature of these "anti-Hebbian" connections. The adaptation of the neuronal activities is presented here in the continuous-time form, and it is assumed that dynamics of this internal loop is much faster than the dynamics of the input x, so that one can solve for the steady state

$$\bar{z} = F^T x = A^{-1}B\, x, \tag{8.42}$$

where now

$$A = \beta \mathrm{E}\{\bar{z}\bar{z}^T\}, \qquad \text{and} \qquad B = \beta \mathrm{E}\{\bar{z}x^T\}. \tag{8.43}$$

Note that because A represents the covariance matrix of the state vector, all eigenvalues being non-negative, dynamics determined by the matrix $-A$ always remains stable. The covariance matrix estimates can be adapted, for example, using a continuous-time algorithm for some time constant $\tau \gg 1/\beta$ as

$$\frac{d\,\hat{\mathrm{E}}\{\bar{z}x^T\}}{dt} = -\frac{1}{\tau}\hat{\mathrm{E}}\{\bar{z}x^T\} + \frac{1}{\tau}\bar{z}x^T. \tag{8.44}$$

Because the neocybernetic studies concentrate on balances on all levels, it is of interest to see what are the stationary properties of $\bar{x}$. One has

$$\begin{array}{rcl} \mathrm{E}\{\bar{z}\bar{z}^T\} & = & A^{-1}B\, \mathrm{E}\{xx^T\}\, B^T A^{-1} \\ & = & \mathrm{E}\{\bar{z}\bar{z}^T\}^{-1}\mathrm{E}\{\bar{z}x^T\}\, \mathrm{E}\{xx^T\}\, \mathrm{E}\{\bar{z}x^T\}^T \mathrm{E}\{\bar{z}\bar{z}^T\}^{-1}, \end{array} \tag{8.45}$$

or, when simplified

$$\mathrm{E}\{\bar{z}\bar{z}^T\}^3 = \mathrm{E}\{\bar{z}x^T\}\mathrm{E}\{xx^T\}\mathrm{E}\{\bar{z}x^T\}^T. \tag{8.46}$$

Taking the linearity of the model into account, there holds

$$\left(F\mathrm{E}\{xx^T\}F^T\right)^3 = F\left(\mathrm{E}\{xx^T\}\right)^3 F^T. \tag{8.47}$$

The solution for the mapping matrix is non-trivial, if the dimension of the input x is higher than that of the state $\bar{z}$, that is, $N < n$. It turns out that the columns of F span the *principal subspace* of the input data, that is, they are linear combinations of the N most significant principal components. It also turns out that the neocybernetic principles are enough to implement self-regulation and self-organization (in the sense of PCA), even though the local synapses only are capable of reacting to their immediate environment, knowing nothing about the global situation. From the technical point of view, it is nice that explicit covariance matrices in the assumedly high-dimensional space of x vectors is not needed — one essentially operates in the low-dimensional space of z vectors. However, the process is necessarily highly iterative as the final balances $\bar{z}$ are not known before adaptation.

Looking at the structure of the mapping matrix F, it is evident that one can implement reconstruction of the input in the least-squares sense in a straightforward way:

$$\hat{x} = \mathrm{E}\{\bar{z}x^T\}^T\mathrm{E}\{\bar{z}\bar{z}^T\}^{-1}\,\bar{z} = F\,\bar{z}. \tag{8.48}$$

This means that the internal state $\bar{z}$ can be interpreted as some kind of "mirror image" of the environment as represented by x.

Further, one can also implement normal principal component regression exploiting the principal subspace (see Fig. 8.13):

$$\hat{y} = \mathrm{E}\{\bar{z}y^T\}^T\mathrm{E}\{\bar{z}\bar{z}^T\}^{-1}\,\bar{z} = \mathrm{E}\{\bar{z}y^T\}^T\mathrm{E}\{\bar{z}\bar{z}^T\}^{-2}\mathrm{E}\{\bar{z}x^T\}\,x. \tag{8.49}$$

It seems that the dynamic systems understanding can give new intuitions for studying regression models. Also, as it turns out in what follows, understanding of the static regression models can help to better exploit the dynamic models.

Computer exercises

1. Construct data as follows:

   ```
   X = 20*rand(100,1)-10;
   Y = [cos(X/2),abs(X)/5-1];
   Xtest = [-15:0.1:15]';
   ```

   Study the behavior of the radial basis regression for different values of N (number of clusters) and σ (width of the distributions):

   ```
   [clusters] = regrKM(X,N);
   [rbfmodel] = regrRBFN(X,Y,clusters,sigma);
   Ytest = regrRBFR(Xtest,rbfmodel);
   ```

2. Assuming that you have the `Neural Network Toolbox` for `Matlab` available (version 3.0.1, for example) study the robustness of training the multilayer feed-forward perceptron network. Using the same data as above, construct the model as

   ```
   structure = [2 1 2 size(Y,2)];
   outfunc = {'tansig', 'tansig', 'tansig', 'purelin'};
   net = newff([min(X);max(X)]',structure,outfunc);
   net = train(net,X',Y');
   ```

   In principle, these commands reproduce the example presented in Figs. 8.12 and 8.11. What can you say about reliability? For model simulation use the commands

   ```
   Ytest = sim(net,Xtest');
   net.outputConnect = [0 1 0 0]; % Second layer output
   Ztest = sim(net,Xtest');
   ```

   Try also different network structures (that is, change the `structure` and `outfunc` parameters). For example, using only one hidden layer, what is the minimum number of hidden layer units that can accomplish the mapping?

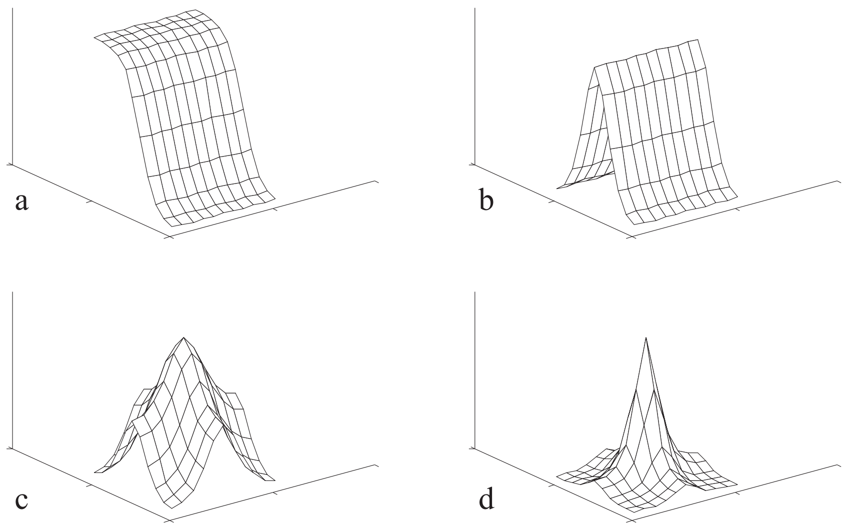


Figure 8.6: Visualization of the generality of feedforward perceptron networks as approximators of smooth functions. In this example, only two input signals is assumed (x_1 and x_2), and in the figures, the outputs of neurons are plotted as functions of these inputs in a two-dimensional (x_1, x_2) plane. First, see **a**: this kind of output function is characteristic to the nonlinear neuron; adjusting the weights of the inputs and the bias, the location, orientation, and depth of the “transition barrier” can be freely adjusted (note that $z = g(w_1x_1 + w_2x_2 + b)$). Similarly, if yet another neuron is connected to the same input using the same ratio between the input weights, the new transition barrier is parallel to the previous one, yet shifted. Figure **b** results if the outputs of these two neurons are added together — the height of the bump can be freely adjusted by changing the weights. Using another set of two neurons, another (not parallel) bump can be constructed, and if the outputs of these four neurons are added together, the result looks something like the surface in **c**. The peak can be emphasized (see **d**) if this signal is connected to a second-layer neuron. It turns out that using four first-layer neurons, a peak can be created anywhere in the plane; if there are enough neurons, a large number of such peaks can be constructed. These peaks can be applied as basis functions (compare to radial basis function networks), and any continuous function can be approximated to arbitrary accuracy. To summarize, a two-layer network with enough hidden layer neurons can approximate any function

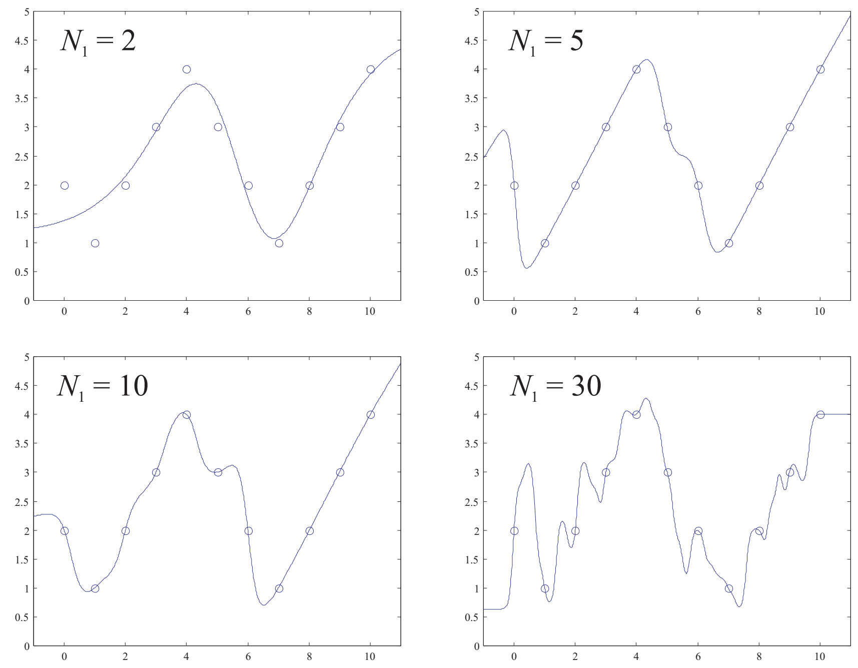
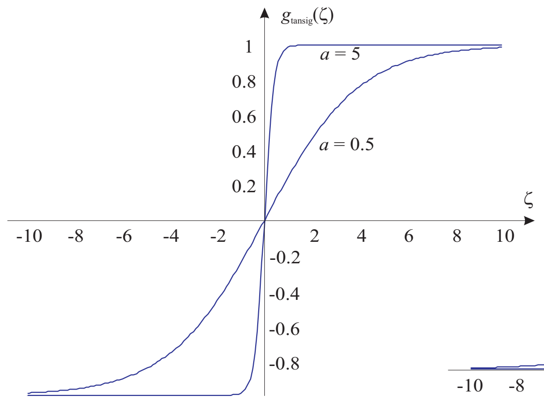


Figure 8.7: Number of neurons — matter of expertise. A two-level feed-forward perceptron network (hyperbolic tangent / pure linearity) with different numbers N_1 of hidden layer neurons has been trained using the dotted points as training samples. As the number of free parameters grows, the matching error becomes smaller, but, at the same time, the curve outlook becomes less predictable. Note that the results can vary from simulation to simulation



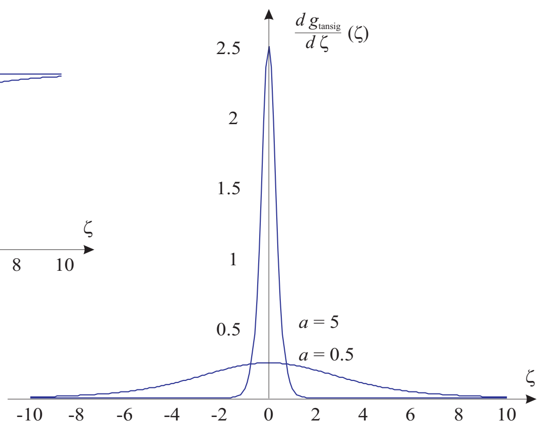


Figure 8.8: Hyperbolic tangent Figure 8.9: ... and its derivative

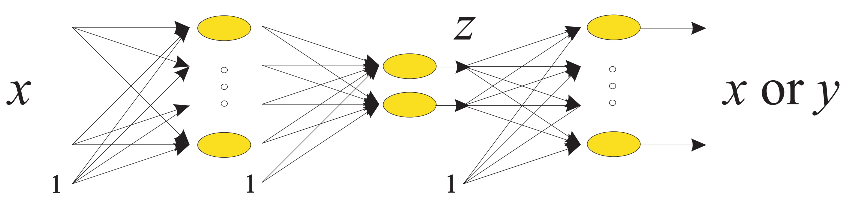
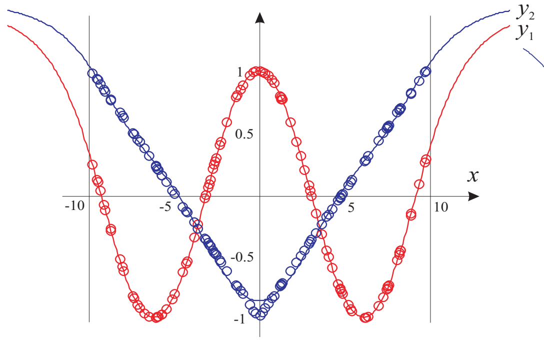


Figure 8.10: Neural network based "nonlinear PCA" (output x) and "nonlinear PLS" (output y). After training, the impact from the input to the output gets channelled through the variables z of lesser dimension. If the mapping from x to x or y still can be done, it must be so that the information has been successfully compressed



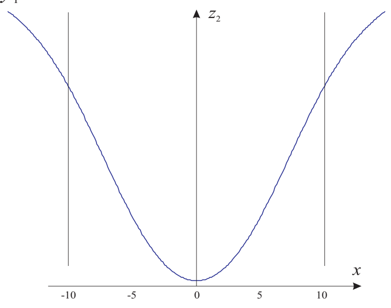


Figure 8.11: Two functions to be modeled (shown as circles), approximations shown using solid line type (see Fig. 8.12)

Figure 8.12: An intuitively reasonable nonlinear latent variable behavior, recognizing the symmetry of the signals in Fig. 8.11

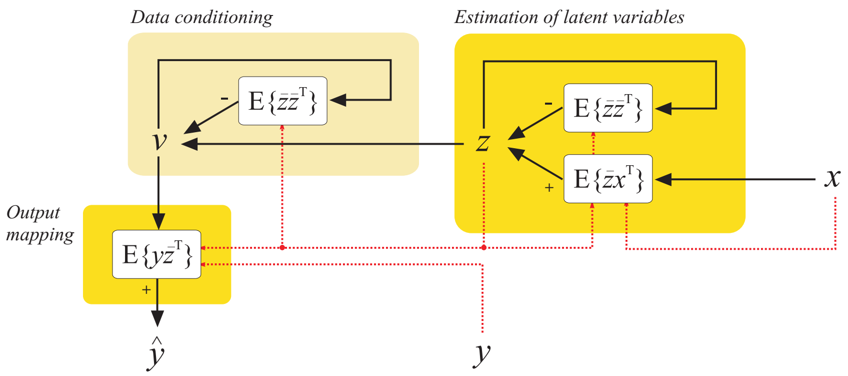


Figure 8.13: "Structure of "cybernetic regression". Note that the notations differ from those employed in [**?**]