Supervised Learning for Neural Networks: A Tutorial with JAVA exercises [1]

Wulfram Gerstner

[1] A chapter in the EPFL graduate volume on ‘Intelligent Systems’, edited by Mlynek and Teodorescu (preliminary)

Chapter 1

Supervised Learning for Neural Networks: A Tutorial with JAVA-exercises

Abstract This chapter gives a short introduction to the theory of neural networks in the context of supervised learning. Simple perceptrons, multilayer perceptrons, radial basis function networks, the backpropagation algorithm, and other topics are discussed. The problem of overfitting and generalization which is of eminent practical importance is emphasized. All theoretical concepts are illustrated by JAVA-applets which the reader can download from:
$http://diwww.epfl.ch/mantra/$

1.1 Introduction

Over the last twenty years neural networks have found their way into numerous applications ranging from character recognition, plant optimization, to financial prediction; see e.g. [2]. Often the term 'neural network' is used is a rather broad sense which groups together different families of algorithms and methods. On the biological end of the spectrum, the term neural network is used to describe models of computation in single neurons or whole areas of the brain. In the following we focus on artificial neural network. There at least three different classes of algorithms may be distinguished: supervised, unsupervised, and reinforcement learning. This chapter focuses exclusively on the supervised learning paradigm.

1.2 Formal Neurons

Artificial neural networks are based on a rather simple model of a neuron. Most neurons have three parts: a dendrite which collects inputs from other neurons

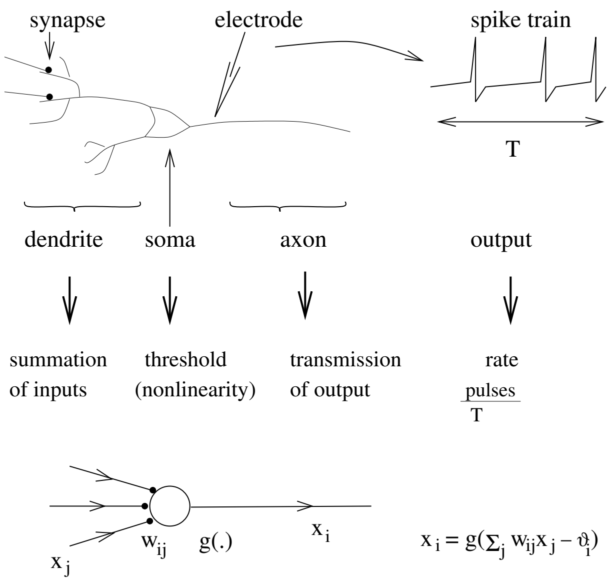


Fig. 1.1: A single artificial neuron. Input values x_j are summed with weights w_{ij} and passed through a nonlinear function $g(\sum_j w_{ij}\, x_j)$.

(or from an external stimulus); a soma which performs an important nonlinear processing step; finally an axon, a cable-like wire along which the output signal is transmitted to other neurons further down the processing chain. The connection site between two neurons is called a synapse. The signal of most real neurons consists of spikes, short pulses of electrical activity. In artificial neural networks, the spikes are replaced by a continuous variable x_j which we may think of as a temporally averaged pulse *rate*.

The formal neuron model describes the following processing steps (Fig. 1.1). The rates x_j of all neurons which send signals to neuron i are weighted with parameters w_{ij}. We may think of these weights as describing the efficacy of the connection from j to i. A weight is therefore sometimes called 'synaptic efficacy'. The output x_i of neuron i is a nonlinear transform of the summed input,

$$x_i = g\left(\sum_{j=1}^{n} w_{ij}\, x_j - \vartheta_i\right) . \tag{1.1}$$

where n is the number of input lines converging onto neuron i and ϑ is a formal threshold parameter. Let us define the total input as $h = \sum_{j=1}^{n} w_{ij}\, x_j - \vartheta_i$. The simplest choice of a nonlinearity would be a threshold function: $g(h) = 1$ for $h > 0$ and zero otherwise. We may write this in the form

$$g(h) = \mathcal{H}(h) \tag{1.2}$$

where we have used the definition of the Heaviside step function.

More generally, we could replace the strict threshold by a 'soft' threshold

$$g(h) = \frac{1}{2}[1 + \tanh(h)]\,. \tag{1.3}$$

The above choice is motivated by the fact that the output of a neuron is typicall close to zero if there is no input at all or if the input is inhibitory. The output rate approaches a maximum value (which can be normalized to unity) if the input is strong. We may use $h = \sum_j w_{ij} - \vartheta_i$ in (1.3). The value of ϑ_i determines the transition regime from low to high output. For an input $\sum_j w_{ij}\, x_j = \vartheta_i$ the output is exactly half the maximum.

Sometimes it is also useful to consider neurons with a linear transfer function

$$g(h) = h\,. \tag{1.4}$$

In particular for neurons which are at the end of the processing chains ('output neurons') this is a suitable choice.

Exercise - Single Neuron. The Java applet summarizes the mathematical concepts of a single model neuron. Play around with the parameters. Try to understand how logical AND and OR can be implemented by a single neuron.

1.2.1 Removing the threshold

In the preceding subsection, two types of parameter have been introduced, viz., weights w_{ij} and a threshold ϑ_i. We will now show that the threshold can be treated as just one extra weight. To do so we will start from (1.1)

$$\begin{aligned} x_i &= g\left(\sum_{j=1}^{n} w_{ij}\, x_j - \vartheta_i\right) \\ &= g\left(\sum_{j=1}^{n} w_{ij}\, x_j + \vartheta_i\,(-1)\right)\,. \end{aligned} \tag{1.5}$$

We may now define a new input line $x_0 \equiv -1$ with weight $w_{i0} = \vartheta_i$. This definition allows us to rewrite (1.5) in the simple form

$$x_i = g\left(\sum_{j=0}^{n} w_{ij}\, x_j\right)\,. \tag{1.6}$$

The advantage of (1.6) is that we are left with a single type of parameter.

1.2.2 Reading Neural Network Graphs

Given the above definitions of an artificial neuron we can immediately read a neural network graph as the one in Fig. 1.2. At the left end of the network, a set

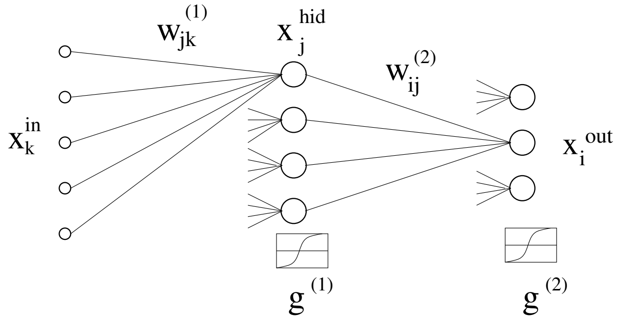


Fig. 1.2: A multilayer network. The input x^{in} is transformed to an output x^{out}.

of inputs x_k^{in} with $1 \le k \le n$ is applied. The set of outputs on the right end is x_i^{out}, $1 \le i \le m$. Let us focus on a single component x_i^{out} of the output. Its value

$$\begin{aligned} x_i^{\text{out}} &= g^{(2)}[h_i^{(2)}] \\ &= g^{(2)}[\sum_j w_{ij}^{(2)} x_j^{\text{hid}}] \\ &= g^{(2)}[\sum_j w_{ij}^{(2)} g^{(1)}(h_j^{(1)})] \\ &= g^{(2)}[\sum_j w_{ij}^{(2)} g^{(1)}(\sum_k w_{jk} x_k^{\text{in}})] \end{aligned} \tag{1.7}$$

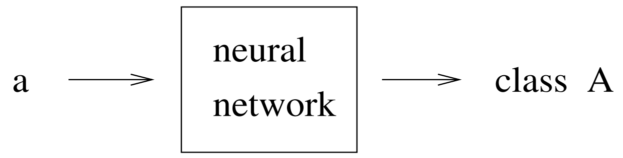
In (1.7) no lower and upper bounds have been marked. We always assume implicitly that the sums run over all corresponding input lines. They start at zero and therefore formally include the threshold. Eq. (1.7) gives the output as a function of the input x^{in}. The parameters of the input-output transformation are the weights. The weights play an important role in the theory of supervised learning.

1.3 Supervised Learning

1.3.1 Motivation

To motivate the paradigm of supervised learning, let us consider a classification problem (Fig. 1.3). We want to have a machine which classifies input patterns into different classes. The input patterns could, for example, be handwritten characters as they frequently occur on cheques or postal addresses. The output should be the correct label of the character. For example, a machine which recognizes A's should give a 1 in the output if the input is an A and zero otherwise. Of course, the machine is not perfect and makes occasionally errors.

For the moment, we may treat the classification machine as a black box which represents some input-output transform. Such a transform will have some parama)



b)

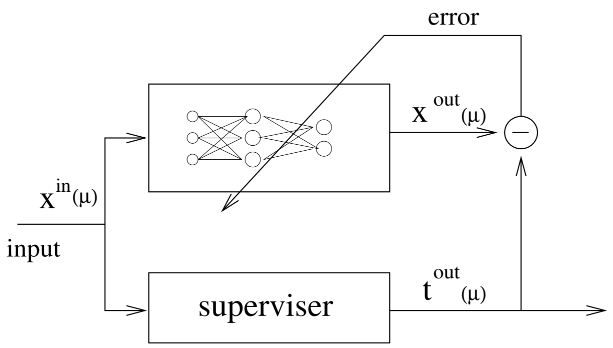


Fig. 1.3: a) The task of automatic classification. An input, for example a pixel pattern of a handwritten character, is to be classified as either belonging to class **A** or not. b) The parameters of the classifier are adapated using training data where the superviser knows the correct output $t^{\text{out}}(\mu)$. The error $x^{\text{out}}(\mu) - t^{\text{out}}(\mu)$ is fed back to the neural network.

eters which we can change. We want to optimize those parameters so that the machine makes as few errors as possible.

How can we optimize these parameters? In supervised learning we always assume that there is a data base χ which contains a number of examples where the input is given *together* with the correct output. A data base χ in supervised learning consists of P input-output pairs $\mathbf{x}^{\text{in}}(\mu), \mathbf{t}^{\text{out}}(\mu)$ which are numbered by the index μ:

$$\chi = \left\{ \left(\mathbf{x}^{\text{in}}(\mu), \mathbf{t}^{\text{out}}(\mu) \right); 1 \leq \mu \leq P \right\} . \tag{1.8}$$

The output value $\mathbf{t}^{\text{out}}(\mu)$ is often called the *target value* which motivates the letter $\mathbf{t}^{\text{out}}$ as a shorthand. The data base χ can be used to adapt the parameters of the input-output transform.

Of course, the final task of the machine will be to give the correct answer to new input data which it has never seen before. It is therefore not sufficient to look simply at the performance of the machine for the known input-output pairs in the data base. What we want is that the machine 'learns' the essential aspects of the input output mapping and then 'generalizes' to new data.

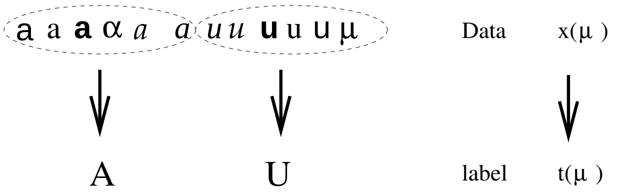


Fig. 1.4: Example of a data base for supervised learning. Each input pattern, for example the pixel pattern of a character, is associated with the correct label, e.g., **A** for a.

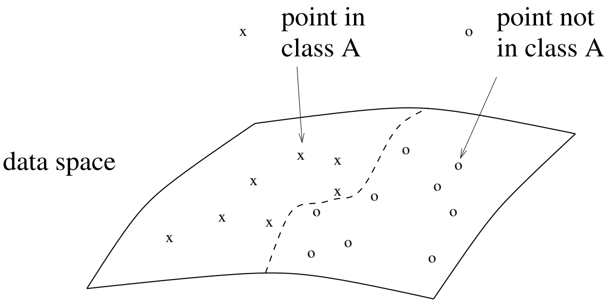


Fig. 1.5: The dashed line separates the crosses from the zeros.

1.3.2 Geometrical visualization

In general the input data $\mathbf{x}^{\text{in}}(\mu)$ can be high dimensional vectors $\mathbf{x}^{\text{in}} \in \mathbb{R}^n$ with components x_k^{in}, $1 \leq k \leq n$. In the case of handwritten character recognition, we could try to work directly with the pixel patterns which might have a resolution of 16×16. Thus $n = 256$. We may imagine the data vectors as points in the n-dimensional space (Fig. 1.5). Some of the input patterns have a target value $t^{\text{out}}(\mu) = 1$. They are labeled as members of the class and marked by crosses. Others have $t^{\text{out}}(\mu) = 0$ and are marked by zeros. In the geometrical picture, the task of generalization corresponds to finding a surface which separates the elements of class A from the non-elements.

We can rephrase this statement mathematically: we want to find a function $d_A(x^{\text{in}})$ so that $d_A(x^{\text{in}}) > 0$ for all point $x^{\text{in}}(\mu)$ which are elements of class A, and $d_A(x^{\text{in}}) < 0$ for points which do not belong to class A. The separating surface is the set of points with $d_A(x^{\text{in}}) = 0$. For any new data point x^{in} we may predict the classification simply by evaluating the function $d_A(x^{\text{in}})$. If $d_A(x^{\text{in}}) > 0$ we guess that the point belongs to class A and vice versa. The function d_A would be called a discriminant function for class A.

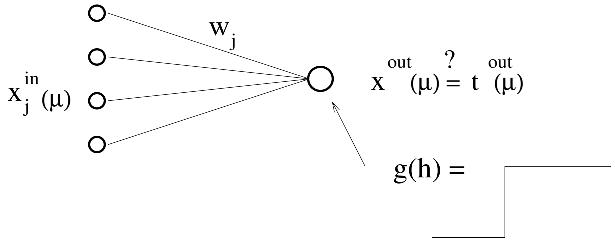


Fig. 1.6: The simple perceptron consists of a single neuron with a threshold nonlinearity $g(h) = \mathcal{H}(h)$.

1.4 Simple Perceptrons

Let us start with the analysis of a single neuron which receives input from a number of input channels $x_k^{\rm in}$ with $1 \leq k \leq n$ This is called a simple perceptron. If we apply input pattern μ, the output of the neuron is

$$x^{\rm out}(\mu) = \mathcal{H}\left(\sum_{k=1}^{n} w_k x_k^{\rm in}(\mu) - \vartheta\right) . \tag{1.9}$$

Note that the sum starts at $k = 1$ and the threshold has been made explicit. Since there is only a single neuron, there is no need to keep the index of the neuron i in the weights. We have therefore made the replacement $w_{ik} \longrightarrow w_k$. We can think of the weights w_k, $1 \leq k \leq n$ as a vector $\mathbf{w}$ which has the same dimensionality as the input vectors $\mathbf{x}^{\rm in}(\mu)$. The classification task consists in optimizing the weights so that the output of the network $x^{\rm out}(\mu)$ is equal to the target value $t^{\rm out}(\mu)$.

1.4.1 Linear Separability

Let us study (1.9) in more detail. Whenever the argument of the Heaviside function $d(x^{\rm in}) \equiv \sum_{k=1}^{n} w_k x_k^{\rm in} - \vartheta > 0$, the neuron will have an output of $+1$. That is, the neuron will classify the input as an element of the class. Similarly, whenever the argument is negative an input will be classified as not belonging to the class. Thus $d(x^{\rm in})$ is a discriminant function. The set of critical points in the input space which separate the class members from the non-members is given by $d(x^{\rm in}) = 0$, hence

$$0 = \sum_{k=1}^{n} w_k x_k^{\rm in} - \vartheta\,. \tag{1.10}$$

Eq. (1.10) is a linear equation and defines a hyperplane in the input space. Thus we have the result that a simple perceptron can only solve problems which can be separated by a hyperplane. These problems are called linearly separable. Fig. 1.7 illustrates the idea of linear separability in two dimensions.

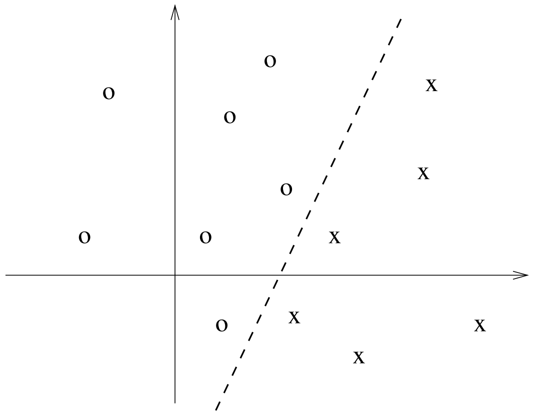


Fig. 1.7: Linear separability: The crosses and the zeros can be separated by a straight line.

We emphasize that we have used a notation in n dimension and an explicit threshold. The threshold is related to the distance of the hyperplane from the origin. If we include the threshold in the weights, we work in $n+1$ dimension and the hyperplane must go through the origin.

1.4.2 Perceptron Algorithm

The simple perceptron can solve problems which are linearly separable - but how can we find the position of the optimal hyperplane? The idea of the perceptron algorithm is that we test one data point $\mathbf{x}^{\mathrm{in}}(\mu)$ after the other. Testing a data point means that we apply $\mathbf{x}^{\mathrm{in}}(\mu)$ at the input layer of the perceptron and compare the neuronal output $x^{\mathrm{out}}(\mu)$ with the desird output $t^{\mathrm{out}}(\mu)$. If the output of the perceptron is correct, we don't take any action. If the output is incorrect $x^{\mathrm{out}}(\mu) \neq t^{\mathrm{out}}(\mu)$, the weight vector $\mathbf{w}$ is changed $\mathbf{w} \longrightarrow \mathbf{w} + \Delta\mathbf{w}$. The process of weight adaptation is called learning. The learning rule is

$$\Delta\mathbf{w} = \eta \left[t^{\mathrm{out}}(\mu) - x^{\mathrm{out}}(\mu)\right] \mathbf{x}^{\mathrm{in}}(\mu) \tag{1.11}$$

where $\eta > 0$ is some small parameter. It is useful to introduce a variable $\delta(\mu) = t^{\mathrm{out}}(\mu) - x^{\mathrm{out}}(\mu)$. The learning rule then becomes

$$\Delta\mathbf{w} = \eta\, \delta(\mu)\, \mathbf{x}^{\mathrm{in}}(\mu) \tag{1.12}$$

which makes the geometrical structure of the learning process more transparent. If the actual output $x^{\mathrm{out}}(\mu)$ is correct, $\delta(\mu)$ is zero and the weight vector is not changed. Let us consider a data point $\mathbf{x}^{\mathrm{in}}(\mu)$ with target value $t^{\mathrm{out}}(\mu) = 1$ for which the perceptron gives the false output $x^{\mathrm{out}}(\mu) = 0$. In this case $\delta(\mu)$ is positive and a change $\Delta\mathbf{w}$ of the weight vector *in direction* of $\mathbf{x}(\mu)$ is applied.

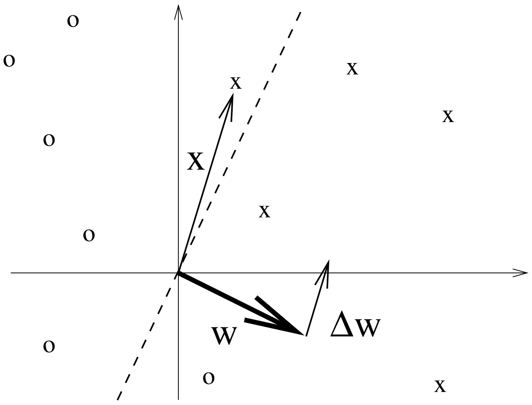


Fig. 1.8: Perceptron algorithm. If a point is misclassified like the cross (vector $\mathbf{x}$) on the left-hand side of the separating line (dashed), the weight vector $\mathbf{w}$ is changed in direction of $\mathbf{x}$. This rotates the seprating line in the desired direction.

This rotates the hyperplane around the origin. Note that the algorithm as it is presented here supposes that the threshold is absorbed in an additional weight component. We will see in the following that nearly all of the learning rules have a structure analogous to (1.12).

Exercise - Perceptron Learning Rule. Choose the threshold nonlinearity and define a 'AND' or 'OR' problem. Watch how the perceptron learning rule finds the correct separation.

1.4.3 Gradient descent

Even if we are interested in a classification task, we may nevertheless consider a neuron with a continuous output function $g(h)$. For example, we may take the sigmoidal function $g(h) = (1/2)[1 + \tanh(h)]$ introduced in (1.3). The advantage of the continuous output is that we can apply the general idea of gradient descent to the learning process. The total quadratic error that the neural network makes on all training samples in the data base χ is

$$E(\mathbf{w}) = \frac{1}{2} \sum_{\mu} [t^{\text{out}}(\mu) - x^{\text{out}}(\mu)]^2 \,. \tag{1.13}$$

Since $x^{\text{out}}(\mu) = g[h(\mu)] = g[\sum_k w_k x_k^{\text{in}}(\mu)]$ depends on the weights w_k, the error is a function of the weight vector $\mathbf{w}$. Changing the weights may increase or decrease the error. We may therefore adjust the weights so as to minimize the error. A simple method to reduce the error is gradient descent. The weights are changed in direction of the negative gradient

$$\Delta w_k = -\eta \, \frac{dE}{dw_k} \tag{1.14}$$

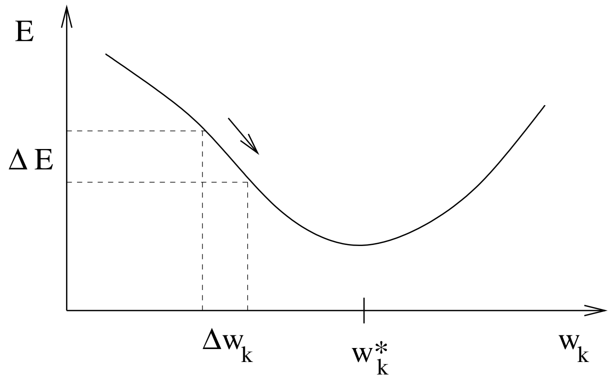


Fig. 1.9: Gradient descent. A change of the weight in direction of the negative gradient $\Delta w_k = -\eta dE/dw_k$ lowers the error by an amount ΔE.

where η is some small parameter.

The gradient dE/dw_k can be calculated using the chain rule. We set $h(\mu) = \sum_j w_j x_j^{\text{in}}(\mu)$ and find

$$\begin{aligned}\frac{dE}{dw_k} &= \sum_\mu \frac{dE}{dh(\mu)} \frac{dh(\mu)}{dw_k} \\ &= -\sum_\mu \left[t^{\text{out}}(\mu) - x^{\text{out}}(\mu)\right] g'|_{h(\mu)}\, x_k^{\text{in}}(\mu) \\ &= -\sum_\mu \delta(\mu)\, x_k^{\text{in}}(\mu) \qquad (1.15)\end{aligned}$$

The factor g' is the derivative of the gain function $g(h)$ evaluated at $h = h(\mu)$. Note that (1.15) contains the sum over all patterns. If we put (1.15) in (1.14) we have a learning rule where we change the weights only after we have tested all the patterns and calculated the term $\delta(\mu)$ for $1 \leq \mu \leq P$. Such a learning rule would be called a 'batch' rule. In practice it is easier to use learning in an on-line mode. The weights are changed immediately after a single pattern has been applied,

$$\Delta\mathbf{w} = \eta\, \delta(\mu)\, \mathbf{x}^{\text{in}}(\mu) \qquad (1.16)$$

where we have switched to the vector notation. The form of the learning rule is identical to (1.12), except that the factor δ is now $\delta(\mu) = [t^{\text{out}}(\mu) - x^{\text{out}}(\mu)]\, g'|_{h(\mu)}$. The only difference compared to the perceptron learning rule (1.11) is the additional factor g'.

In the online mode, it is no longer guaranteed that the error E decreases at each step. A change $\delta\mathbf{w}$ which is good for the present pattern may be bad for several other patterns and therefore increase the total error. If patterns μ are chosen randomly from the data base χ, the online mode corresponds to a stochastic gradient descent algorithm.

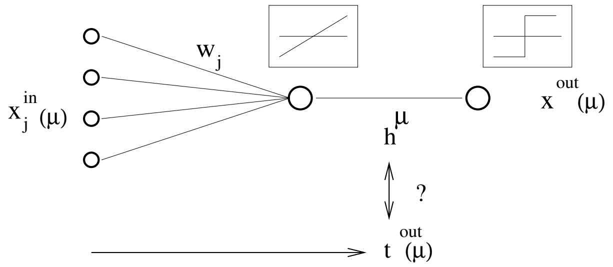


Fig. 1.10: Adaline. The target value $t^{\text{out}}(\mu) = \pm 1$ is compared with the *linear* output $h^\mu = \sum_n w_k x_k^{\text{in}}(\mu)$ to calculate the error. The threshold $x^{\text{out}}(\mu) = g(h^\mu)$ is disregarded during the learning process.

1.4.4 Adaline algorithm

In the fifties and sixties, Widrow and Hoff had worked with a variant of a gradient descent algorithm; see [1]. The error was measured directly after the linear summation step, before the threshold; cf. Fig. 1.10. The name is short for ADAptive LInear NEuron or, more generally, ADAptive LINEar Element.

To understand the principle, define $t^{\text{out}}(\mu) = +1$ if input pattern μ belongs to the class and $t^{\text{out}}(\mu) = -1$ otherwise. If the summed input $h(\mu) = \sum_n w_k x_k^{\text{in}}(\mu)$ is positive, the threshold step will give an output $x^{\text{out}}(\mu) = +1$; if $h(\mu) \leq 0$ the output will be $x^{\text{out}}(\mu) = -1$. For a pattern with $t^{\text{out}}(\mu) = +1$, a summed input of $h(\mu) = +1$ will guarantee the correct output. Similarly, the requirement $h(\mu) = -1$ for patterns $t^{\text{out}}(\mu) = -1$ is sufficient for a correct output. Note that $h(\mu) = +1$ is not a necessary requirement for an output of one. Any value $h(\mu) > 0$ would give the same result $x^{\text{out}}(\mu) = 1$. But if we take the stronger requirement that $h(\mu)$ should be as close as possible to $t^{\text{out}}(\mu)$, we can define an error

$$E_{\text{Ada}}(\mathbf{w}) = \frac{1}{2} \sum_\mu \left[t^{\text{out}}(\mu) - h(\mu)) \right]^2 . \tag{1.17}$$

The error is zero if $t^{\text{out}}(\mu) = h(\mu)$.

Since $h(\mu) = \sum_j x_j^{\text{in}}$, the error (1.17) is a quadratic function in the weights. It follows that $E_{\text{Ada}}(\mathbf{w})$ has a unique minimum which can either be approached by gradient descent or calculated explicitly. The minimum $\mathbf{w}^*$ is given by $dE/dw_k = 0$ for $0 \leq k \leq n$. Thus

$$\sum_\mu t^{\text{out}}(\mu)\, x_k^{\text{in}}(\mu) = \sum_j w_j^* \sum_\mu x_j^{\text{in}}(\mu)\, x_k^{\text{in}}(\mu) . \tag{1.18}$$

This is a linear equation for $\mathbf{w}^*$ which can be solved by standard methods. The advantage of the linear element used for the learning step in Adaline is that direct mathematical methods are available. The disadvantage with view to the classification task is that we are optimizing the wrong error function. In the end we are interested in a correct classification *after* the threshold step and not in the best linear approximation.

Exercise - ADALINE, Perceptron, and Backpropagation. In this applet, you can define your own classification problem. Start with a linearly separable problem. Define one set of 'red' and one set of 'blue' points by clicking on the surface. Use the Perceptron algorithm, ADALINE, or gradient descent to solve the problem (gradient descent is called Backpropagation in this Applet). Can you define a problem which is correctly solved by the Perceptron algorithm, but where ADALINE does not find a solution?

1.4.5 Optimal Perceptron

Let us suppose that we have indeed a problem which is linearly separable. If there exists one solution, there are almost always many others. Which one is the best? One possibility is to choose the solution with the maximal safety margin. Fig. 1.11 illustrates the idea. The dashed line correctly separates the zeros from the crosses, but some of the data points are close to the separating line. The optimal separation is given by the thick solid line. Data points on both sides have at least a distance of d. Note that several points will lie directly on the margin. These points are called the support vectors.

Let us assume that we work in $n+1$ dimensions with the threshold ϑ integrated as the first component of the weight vector. Formally the optimal separation can then be defined as the line where the closest point has a maximal distance

$$d = \max_{\mathbf{w}} \left\{ \min \left\{ \frac{\mathbf{w}\,\mathbf{x}^{\mathrm{in}}(\mu)}{\|\mathbf{w}\|} \;, 1 \leq \mu \leq P \right\} \right\} . \tag{1.19}$$

Optimizing the margins should make the system relatively stable with respect to noise.

1.4.6 Discussion

The theory of simple perceptrons had been developed in the fifties during the first big vague of neural networks. The simple perceptron has one obvious disadvantage: it can only solve linearly separable problems. Obviously there are many problems which are not linearly separable. The prototype of a non-separable problem is the XOR-problem.

What can we do with those problem which are not linearly separable? One idea is that we may be able to find a preprocessing scheme which recodes the problem so that the remaining problem is linearly separable. The idea is illustrated in Fig. 1.12. The input vector $\mathbf{x}^{\mathrm{in}}(\mu)$ is pre-processed in a first layer and yields a set of features with coefficients $\phi_1(\mathbf{x}^{\mathrm{in}}(\mu)) \ldots, \phi_n(\mathbf{x}^{\mathrm{in}}(\mu))$. These are then treated as input to a simple perceptron $x^{\mathrm{out}}(\mu) = \mathcal{H}\left[\sum_{k=1}^{n} w_k\, \phi(\mathbf{x}_k^{\mathrm{in}}(\mu))\right]$.

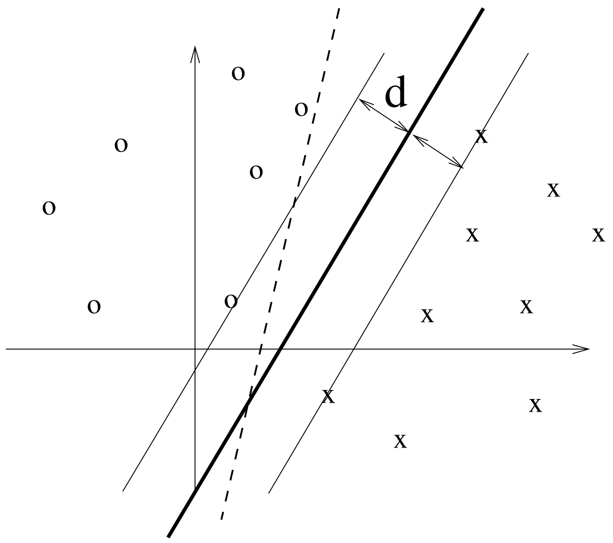


Fig. 1.11: The optimal perceptron. The thick line provides the separation with the maximum margin. The dashed line gives a sub-optimal separation.

Unfortunately, however, there exists no universal preprocessing method which is *local* in the input space and would transform any difficult problem in a linearly separable one. In fact, Minsky and Papert gave the example of task where figures have to be classified into connected ones (consisting of a single line) and those which are not connected. Since this is a global question which can not be decided locally, no local preprocessing can simplify the problem - the remaining task that has to be solved in the classification step is always of the XOR type and can therefore not be solved by a simple perceptron. The book of Minsky and Papert [7] has been very influential and their critique of the perceptron basically killed the field of neural networks for a long period. Today it is often stated that simple perceptrons are of historical interest only.

This is, however, not completely true. More recently the idea of preprocessing followed by a classification by simple perceptrons (this idea in fact never completely died) has come back in the context of 'Support Vector Machines' (SVM). Preprocessing by convolution with a set of kernels maps the problem into a high (potentially infinite) dimensional feature space. In the feature space the separation with maximum margin is chosen. The classification approach is therefore exactly the one used in the 'optimal perceptron'.

Using a smart mathematical formulation of the preprocessing step, it is possible to avoid the explicit calculation of the features by convolution. The problem of finding the optimal separation can be formaulated directly as a quadratic programming problem. For details see, e.g., [11].

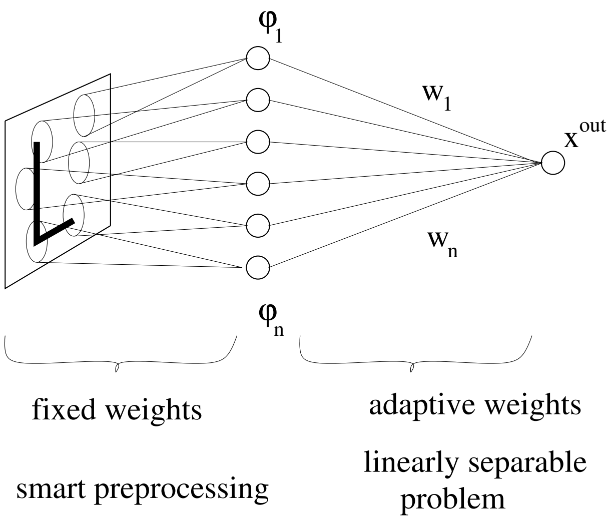


Fig. 1.12: The idea of the perceptron. A preprocessing step is followed by a simple perceptron.

1.5 Multilayer Perceptrons (MLP)

Multilayer perceptrons are organized in layers of neurons and implement a feed-forward processing chain. The layers between the input nodes and the output neurons are called hidden layers. Multilayer Perceptrons are more flexible than simple perceptrons. The most important learning rule for network training is the backpropagation algorithm.

1.5.1 The XOR problem

In the preceding section we have seen that the simple perceptron can only solve linearly separable problems. The prototype of an example which is not linearly separable is the XOR problem. It is straightforward to construct a two-layer perceptron with two neurons in a hidden layer which solves the XOR-problem; Fig. 1.13. The first neuron in the hidden layer uses a linear separation which splits off the OFF/OFF input. The second neuron separates the ON/ON input from the rest. If both neurons in the hidden layer are inactive, then the input is either ON/OFF or OFF/ON. What remains for the neuron in the output layer is now a linearly separable problem. The above arguments show that a multilayer perceptron is more flexible and can, in principle, solve problems that a simple perceptron cannot solve. The construction does not, however, tell us how to find the optimal weights in a general, potentially much more complicated situation. To get weights automatically, we need a learning rule.

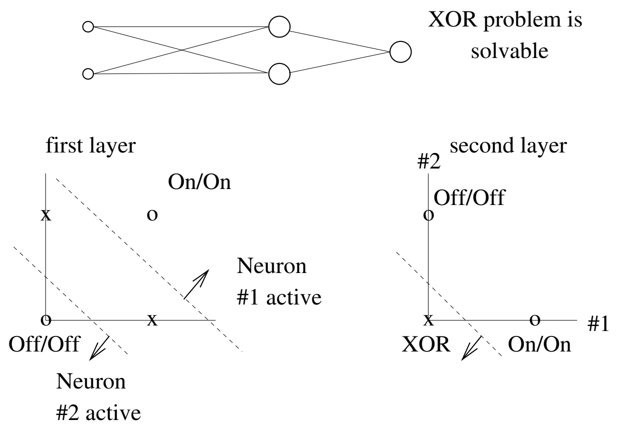


Fig. 1.13: Solution of the XOR problem by a 2-layer perceptron (+ layer of input nodes). There are two neurons in the hidden layer.

1.5.2 Backpropagation Algorithm

The Backpropagation algorithms is the gradient descent rule applied to a multilayer perceptron. A clever use of the chain rule allows us to implement the algorithm rather efficiently.

We use the notation introduced in Fig. 1.2 and Eq. (1.7). As in the section on gradient descent, the aim is to minimize the quadratic error

$$E(\mathbf{w}^{(1)}, \mathbf{w}^{(2)}) = \frac{1}{2} \sum_{\mu} \sum_{i} \left[t_i^{\text{out}}(\mu) - x_i^{\text{out}}(\mu) \right]^2 \tag{1.20}$$

where the sum over i runs over all neurons in the output layer. In order to apply the gradient descent rule

$$\Delta w_{ij}^{(k)} = -\eta \, \frac{dE}{dw_{ij}^{(k)}} \tag{1.21}$$

we need to calculate the derivative $dE/dw_{ij}^{(k)}$. The superscript k refers to the layer. For the sake of simplicity, we introduce the total input to neuron i in layer k as $h_i^{(k)}(\mu) = \sum_j w_{ij}^{(k)} x_j^{(k-1)}(\mu)$ where $x_j^{(0)}(\mu) = x_j^{\text{in}}(\mu)$. We use the chain rule and write

$$\frac{dE}{dw_{ij}^{(k)}} = \sum_{\mu} \frac{dE}{dh^{(k)}(\mu)} \, \frac{dh_i^{(k)}(\mu)}{dw_{ij}^{(k)}} \, . \tag{1.22}$$

Let us define $\delta_i^{(k)}(\mu) = -dE/dh_i^{(k)}$. This yields a learning rule of the form

$$\Delta w_{ij}^{(k)} = \eta \sum_{\mu} \delta_i^{(k)}(\mu) \, x_j^{(k-1)} \tag{1.23}$$

which we recognize; cf. Eqs. (1.12,1.16). The δ factors are

$$\delta_i^{(n)} = \left[t_i^{\text{out}}(\mu) - x_i^{\text{out}}(\mu) \right] \, g'^{(n)}|_{h_i^{(n)}(\mu)} \tag{1.24}$$

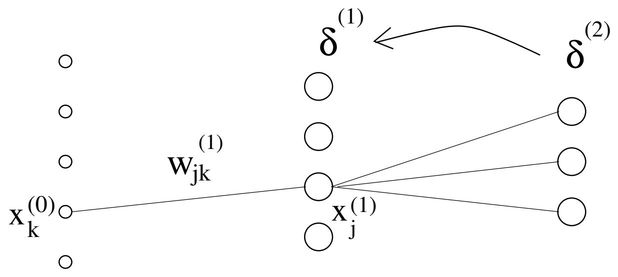


Fig. 1.14: Backpropagation of the δ values.

for the output layer $k = n$ and

$$\delta_j^{(k-1)} = g'^{(k-1)}|_{h_j^{(k-1)}}(\mu) \sum_i w_{ij}^{(k)} \delta_i^{(k)} \tag{1.25}$$

for all other layers $1 \leq k \leq n-1$. The final equation (1.25) gave rise to the name 'backpropagation of errors' for the algorithm. The δ-factors are seen as local errors. To calculate the error at layer $k-1$, we need the error at layer k which is therefore propagated backwards. Even though our discussion has focussed on networks with a single hidden layer, the same algorithm can also be used for networks with several hidden layers.

A gradient descent algorithms converges to local minima. It is not guaranteed that it finds a global minimum. It is therefore necessary in applications to restart the algorithm several times with different initial conditions. The online version of the algorithm which introduces a certain degree of stochasticity helps to escape from small local minima, but there may be deep local minima in which the algorithm is trapped.

Exercise - Multilayer perceptron. Define a problem of the 'XOR'-type by distributing about twenty points close to the four corners. Take for example a network with a single hidden layer that contains four hidden neurons. Try to optimize the learning parameters. How often gets learning stuck in a local minimum? How often does it find a correct separation of the points?.

1.5.3 Momentum terms

In practical applications the pure form of a gradient descent algorithm is rarely used. There are several tricks and methods to speed up convergence. One standard variation of gradient descent is the addition of a momentum term. At each iteration, the weight change keeps a little bit of the direction of the previous weight change. Thus the weights behave as if they had some inertia or 'momentum'.

Let us assume that the change of weight $w_{ij}^{(k)}$ in the last time step was $\Delta w_{ij}^{(k)}(t-1)$. For the change at time t pure gradient descent would give $\Delta w_{ij}^{(k)} =$

$-\eta dE/dw_{ij}^{(k)}$. Adding a momentum term means that we use instead

$$\Delta w_{ij}^{(k)}(t) = -\eta dE/dw_{ij}^{(k)} + \alpha\,\Delta w_{ij}^{(k)}(t-1) \tag{1.26}$$

with some parameter $0 < \alpha < 1$. The momentum terms helps to avoid oscillations when a minimum is approached.

1.6 Radial Basis Function Networks

Not all neural networks are Multilayer Perceptrons. The term neural networks is used these days more generally. An important class of neural networks are the so-called Radial Basis Function Networks (RBF). There are two layers of neurons. The first one calculates a nonlinear function $\phi_j(x) = \mathcal{G}(|\mathbf{x}^{\text{in}} - \mathbf{c}_j|)$ where $\mathbf{c}_j$ is the center of the radial basis function ϕ_j. A common example of a Radial Basis Function is the Gaussian

$$\mathcal{G}(|\mathbf{x}^{\text{in}} - \mathbf{c}|) = \exp\left\{-\left(\frac{\mathbf{x}^{\text{in}} - \mathbf{c}}{2\sigma^2}\right)^2\right\} \tag{1.27}$$

The second layer contains the output neurons. They have a linear transfer function. Thus the output is

$$x_i^{\text{out}} = \sum_{j=1}^{n} w_{ij}\,\phi_j(x^{\text{in}}) + w_0\,\phi_0. \tag{1.28}$$

where $\phi_0 \equiv -1$ is a constant. The term $w_0\phi_0$ plays the same role as the threshold in the case of perceptrons. Formally, it can be included in the summation if we start the sum at $j = 0$.

1.6.1 Linear Optimization

Often the centers $\mathbf{c}_j$ of the basis functions (and similarly the variance σ of the Gaussians) are chosen by simple heuristic arguments. For example we may arbitrarily chose n out of the $P > n$ data samples in our data base and set $c_j = x^{\text{in}}(\mu_j)$ for $1 \le j \le n$. The center σ of the Gaussians can be chosen so that each basis function touches one of the centers of the others. More involved schemes could make use of some unsupervised learning procedure.

Once the centers and the other parameters of the radial basis functions $\phi_j(\mathbf{x})$ have been chosen, the only parameters to optimize are the weights w_{ij}. But this is now a *linear* optimization problem. Define $A_{j\mu} = \phi_j\left(x^{\text{in}}(\mu)\right)$. For simplicity we discuss the case of a single output unit and suppress the index i. The quadratic

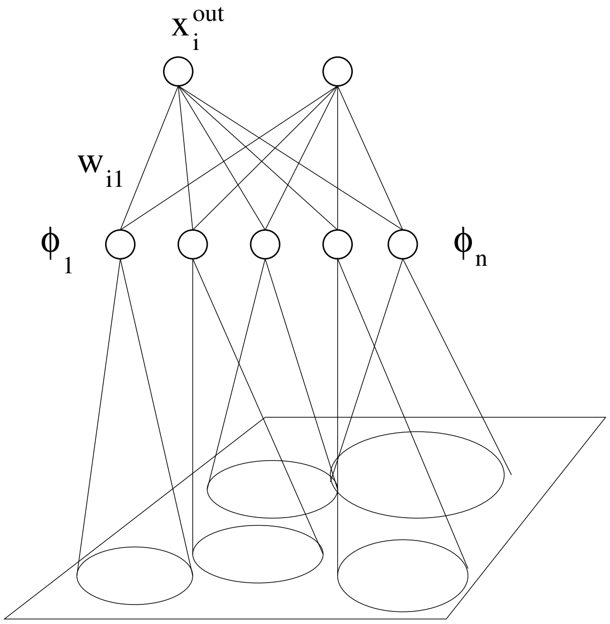


Fig. 1.15: A radial basis function network (RBF).

error is

$$
\begin{aligned}
E(\mathbf{w}) &= \frac{1}{2}\sum_{\mu=1}^{p}\left[x^{\mathrm{out}}(\mu) - t^{\mathrm{out}}(\mu)\right]^2 \\
&= \frac{1}{2}\sum_{\mu=1}^{p}\left[\sum_{j=0}^{n} w_j A_{j\mu} - t^{\mathrm{out}}(\mu)\right]^2 \qquad (1.29)
\end{aligned}
$$

The condition $dE/d\mathbf{w} = 0$ yields a *linear* equation for $\mathbf{w}$ which can be solved by direct methods. The same linear optimization problem has already been encountered in (1.18) in the context of ADALINE.

Exercise - Radial Basis Function Network. In this applet a RBF network is used for function approximation. The centers of the functions are equally distributed over the one-dimensional input space. Define a set of about fifteen input-output pairs. Watch the output of the network. Now reduce the number of data points while keeping the number of basis functions constant. What happens?

1.6.2 Nonlinear Optimization

More generally we may wish to optimize the centers $\mathbf{c}_j$ and the parameters of the basis functions by a supervised learning procedure. Following the same arguments as for the derivation of the backpropagation algorithm, we can apply gradient descent to $\mathbf{c}_k$ and to the variance σ. The derivation of the formulas is straightforward, but tedious. Another possibility of parameter optimization is given by the Expectation Maximization algorithm. In low dimensions it is also possible to use, instead of radial kernels, Gaussians with arbitrary covariance matrix.

Exercise - Gaussian Mixture Model/EM. Define a set of about 20 data points which yo may want to distribute into two clusters. Set the number of kernels to two. Watch how the two kernels move to the clusters and adjust there covariance matrix so as to optimally model the data distribution. What happens if the number of kernels does not match the number of clusters?

1.6.3 Discussion

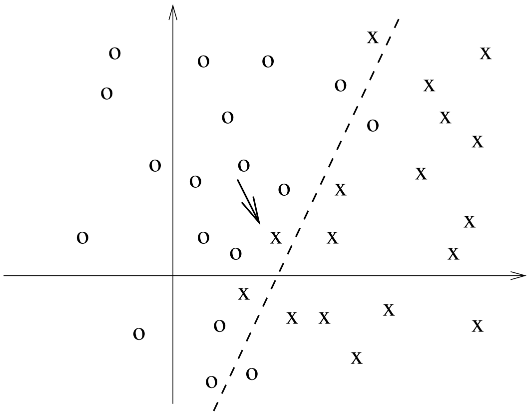
The framework of the radial basis function network makes the connection between neural networks and classical approximation techniques in statistics more transparent. In fact, methods which use a superposition of several Gaussians are commonly used in density estimation. In statistics these methods are also called mixture models or, more specifically, a mixture of Gaussians. The main difference is that radial basis function network usually assume that several output units can share the same basis functions $\phi_k(\mathbf{x})$, $1 \leq k \leq n$. The standard approach in density estimation would be to use for each class an independent set of basis functions. See the book of Bishop for an in-depth discussion of the relation [4].

Another advantage of radial basis functions is that they allow more easily an interpretation of what the network is doing. This goes in two directions. First, expert knowledge can be easily implemented and used for an initialization of parameters. Second, after training the mapping that the network has found can be analyzed. In many cases it is possible to extract the knowledge in terms of rules. The relation between neural networks and expert knowledge is exploited in the context of 'Neuro-Fuzzy-Networks'.

1.7 The Problem of Overfitting and Generalization

1.7.1 Motivation

We have seen in the preceding sections that a simple perceptron can only solve linearly separable problems whereas a multilayer perceptron is more powerful. The question arises whether it is always desirable to have a flexible network. The answer is negative. In some cases it is better to have network which does not perform perfectly on the data set. To understand why, let us recall the task of supervised learning. The data set is used to train a network to perform a desired a)



b)

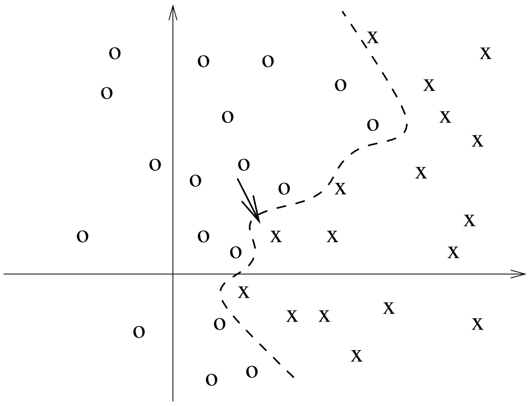


Fig. 1.16: The separation of a data set a) by a simple perceptron and b) by a multilayer perceptron (schematic). The multilayer perceptron performs perfectly on the data set. What matters, however, in applications is the prediction for new data points where the answer is not known, like the point indicated by the arrow.

input-output mapping, e.g., for automatic classification. The final network will be used on *new* data which are not part of the training data base. The performance of the network has therefore to be measured on new data which has not been used during training.

The basic idea is illustrated in Fig. 1.16. In a) the data set has been separated by a simple perceptron. Since the data set is not linearly separable a few errors remain. The more flexible network in b) is capable of a perfect separation. Let us now see how the network performs on new data, for example the data point indicated by an arrow in the input space. The network in b) would a predict a classification as a cross, the one in a) as a zero. The problem is that a priori we don't know which one of the two classifications is correct. For example it may be that the cross which caused the tongue in the separation boundary is the results of a noisy measurement and that the likelihood of zeros is much higher in that region than the one for crosses. Then the prediction of the simple network would in fact be better than the one of the more flexible network. It follows that in particular for noisy data, we have to control the flexibility of our network. Otherwise the network may learn the 'noise' instead of the underlying structure of the input-output mapping. Learning the specific data sample including the noise is known as the problem of overfitting.

The same problem is also known in function approximation. Let us suppose we have a data base which contains noisy input-output pairs $(x^{\text{in}}(\mu), t^{\text{out}}(\mu))$ with $x^{\text{in}}, t^{\text{out}} \in \mathbb{R}$. The task of function approximation is to find the underlying structure of the data distribution. A fit by a polynomial with as many free parameters as data points allows one to interpolate the data perfectly, but may cause arbitrarily large oscillations. The output of the function for new input points x^{in} is then meaningless. A good fit should contain few free parameters which forces the function to average away part of the noise.

In neural networks, the weights w_{ij} play the role of the fit parameters. We must therefore control the effective number of weights. More or less systematic methods of the control of model flexibility are usually summarized under the title of regularization procedures. Some of these methods are discussed in the next subsections.

Exercise - Radial Basis Function Network. Go back to the Radial Basis Function Network and study how the oscillations in the representation increase if the number of fit paramters approaches the number of data points. Define a data set which does not cover the input space evenly, for example, you could choose to have more points towards the borders than in the middle of the interval. What happens?

1.7.2 Training and validation set

Whatever the specific regularization method, we always split the data set χ into two parts, χ_1 and χ_2. If the data set contains a total of $P = 1000$ data points, we may for example select stochastically $P_1 = 500$ points for data set χ_1 and use the remaining $P - P_1$ points for χ_2.

Only data set χ_1 is used during learning. It is called the training set. At each step of the learning procedure, we take a pattern $\mu \in \chi_1$, and calculate the weight update. After P_1 updates (one *epoch*), we interrupt learning and test the performance of the current network on the $P - P_1$ samples of data set χ_2. The set χ_2 plays the role of 'new' data which the network has not seen during training. It is therefore called validation set. We also calculate after each epoch the error E_{learn} on the training set. Both errors are plotted as a function of learning time, measured by the number of epochs.

During the initial phase of learning both training and validation error usually

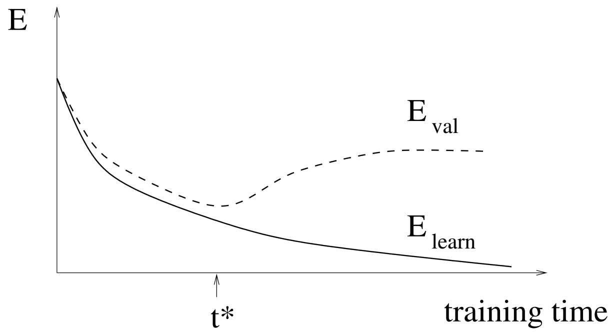


Fig. 1.17: Overfitting. During training with gradient descent rules, the learning error E_{learn} decreases (solid line). The validation error E_{val} (dashed), however, may increase again after some time.

decrease. This means that the network learns the structure of the data as it should. If the network is too flexible, then the validation error E_{val} starts to increase again after a certain number of epochs while the training error continues to decrease. This means that the network now learns the specific examples of the (noisy) data set. The generalization to new data, however, gets worse. The network starts to 'overfit' the data; cf. Fig. 1.17.

Exercise - Generalization in Multilayer Perceptrons. By clicking on the surface, you may define a learning set and a validation set. Make sure the two sets are of the same size, e.g., 30 points each. Take a network with 8 hidden neurons and study the evolution of learning and validation error. Take a small learning step and continue learning for several thousand iterations. Does the validation error rise after some time?

1.7.3 Control of network architecture

Overfitting occurs if the effective number of free parameters in the network is too large. In neural networks, the number of parameter increases with the number of neurons in the hidden layer. A simple procedure to control the flexibility of the network is therefore to test various network architectures with different number of neurons. If the validation error increases consistently towards the end of learning, then the network is too flexible. In this case, a smaller network with a reduced number of neurons should be tried. On the other hand, if the network is too small, then both the learning and validation error are unnecessarily large at the end of learning. The optimal size of the network is achieved if validation and learning error are low after convergence of the learning procedure.

1.7.4 Regularization by early stopping

Changing the network architecture repeatedly is a tedious procedure. Moreover it is rather coarse in the sense that each additional neuron means several new parameters w_{ij}. Thus it is often not completely clear what the optimal network size really is.

A simpler way of flexibility control is given by a method called early stopping. To use this method, we take a network which is slightly too large so that the validation error starts to increases after some time t^*. The best set of weights is therefore the one at t^* - and not the one at the end of learning. During the learning process we therefore keep the set of weights at t^* in a separate memory. These are the weights which we will choose for the final application of the network.

During learning the validation error may fluctuate slightly over several epochs. A first minimum of E_{val} may be followed by a second, deeper minimum much later in the learning process. A correct use of early stopping therefore requires that learning and validation errors are watched over a long time. Learning should not be stopped the first time that the validation error increases.

Early stopping does not formally reduce the number of free parameters. The reason it works is that, at the beginning, the network does not yet use all the flexibility provided by the large number of parameters. At the beginning, several neurons in the hidden layer learn approximately the same weight vector. Only after some time they specialize to detect different features. By early stopping we therefore control the *effective* number of free parameters used by the network.

1.7.5 Regularization by penalty term

Regularization by penalty terms is probably the most systematic way to control the flexibility of the network. The idea is the replace the simple quadratic error function that we have used earlier by a new error criteria

$$E = \frac{1}{2} \sum_{\mu} \left[t^{\text{out}} - x^{\text{out}}(\mu) \right]^2 + \lambda \left\{ \text{penalty} \right\} . \tag{1.30}$$

The extra term (which is specified below) is intended to penalize solutions which are too flexible. The error function (1.30) depends on the parameter λ and as before on the value of the weights $w_{ij}^{(k)}$.

How can we characterize solutions which are too flexible? In function approximation with $x^{\text{in}}, t^{\text{out}} \in \mathbb{R}$ we could give a penalty to solutions with high values of curvature. A good choice of the penalty term would therefore be the mean quadratic curvature $\int [d^2 x^{\text{out}}/(dx^{\text{in}})^2]^2 dx^{\text{in}}$ where the integral runs over the part of the input space that we are interested in.

In case of neural networks with high-dimensional input data, the curvature measure is not so easy to handle. But we may just as well take a different penalty

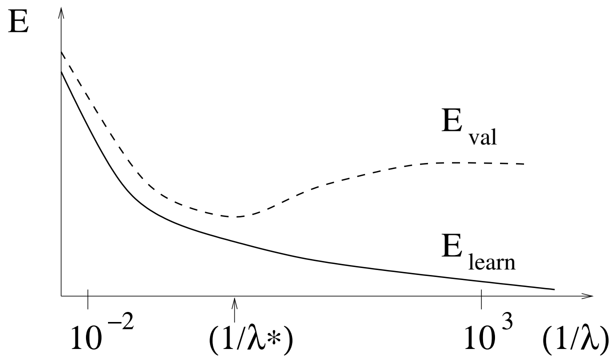


Fig. 1.18: Learning and validation error as a function of the regularization parameter λ.

term. A penalty term which is easy to use is

$$\text{penalty} = \frac{1}{2}\sum_{i,j,k}\left[w_{ij}^{(k)}\right]^2 . \tag{1.31}$$

The sum starts at $i = 1, j = 1, k = 1$ and runs over all weights in all layers. The summation does not include the thresholds (which would correspond to terms with indices $w_{i,0}^{(k)}$).

The motivation behind (1.31) is that a solution with lots of curvature can only be constructed with large weight values. Punishing large weights will therefore control the effective flexibility of the network. On the other hand, threshold values only shift the bias and should therefore be treated differently. Large threshold values are therefore not penalized. It can be shown that for linear optimization problems (that is, the error is quadratic in the weights) a penalty term (1.31) and early stopping are equivalent.

To minimize (1.30) we may use gradient descent, $\Delta w_{mn}^{(k)} = -\eta\, dE/dw_{mn}^{(k)}$. The derivative of the first term in (1.30) gives the standard term that has been discussed previously in the context of backpropagation. The derivative of the penalty term (1.31) is $w_{mn}^{(k)}$. The new learning rule is therefore

$$\Delta w_{mn}^{(k)} = \{\text{standardterm}\} - \eta\lambda\, w_{mn}^{(k)} \tag{1.32}$$

which is the old learning rule plus a *weight decay* term. The penalty term (1.31) is therefore called regularization by weight decay.

The penalty term (1.31) focuses on the mean quadratic weight. Let us denote the total number of weights by M. A penalty term (1.31) does not distinguish between a situation where all weights have the same low value of $1/\sqrt{M}$ and another one where a single weight has a value of 1 and all the others are zero. In fact, in the latter case all weights which are zero could be removed without changing the result. It is therefore argued that the number of parameters used by the network in the second case is smaller than the one in the first case. Therefore the penalty for a homogeneous weight distribution should be larger than the one for a distribution with several zero-weights. A term which does this is

$$\text{penalty} = \sum_{ijk} \frac{\left[w_{ij}^{(k)}\right]^2}{c + \left[w_{ij}^{(k)}\right]^2} \tag{1.33}$$

as can be checked by some examples. The parameter c is a positive constant. A penalty term of the form (1.33) is called *weight elimination.* The name comes from the idea that weights which are close to zero could be eliminated so as to reduce the total number of parameters of the network. Methods which remove weights or neurons are summarized under the title of pruning algorithms. The penalty term (1.33) is, however, useful even if connections with small weights are not removed. The penalty term controls as usual the *effective* number of parameters - a reduction of the total number of parameters by physically removing the connection is not necessary.

All regularization terms are added with a parameter λ. The choice of λ controls the flexibility of the network. In order to optimize λ we can use a method completely analogous to the one used for early stopping. We measure the learning error and the validation error for different values of λ. For each λ we start learning with different initializations and wait till the learning procedure has fully converged. Training and validation errors are plotted as a function of $1/\lambda$. In a sense, $1/\lambda$ plays the role of the number of epochs in early stopping. For large λ the network mainly tries to minimize the penalty term and hardly cares about the data. For small λ, the network is too flexible and overfits the data. The learning error is small, but the validation error large. The optimal value λ^* is at the minimum of the validation error.

Exercise - Generalization in Multilayer Perceptrons. Repeat the previous exercise but include a non-zero weight decay factor.

1.7.6 Discussion

In all applications, neural networks will be used on new data which are not part of the data base. The relevant performance measure is therefore not the training error, but the validation error. Even then it may be argued that the data set χ_2 on which the validation error is measured is not completely new, since it has already been used to optimize the parameter λ of the regularization term or, similarly, the stopping time for early stopping.

It order to have a performance measure which is absolutely neutral, it would therefore be necessary to split the data base χ into three parts, one for learning, a second one for validation, and a third one for testing. This is, however, a luxury which can be afforded only if the data base is very large, e.g., several thousands of samples. In practice, the number of examples is rather too small than too large. Then even a separation of the data base into two parts is difficult. The question arises of how many of the samples should be reserved for the validation set. It is, of course, not necessary to keep half of the data for validation, but how many? This is difficult to decide. In the extreme case, we may opt for a take-one-out cross-validation. One of the samples is picked for validation, the remaining $P-1$ are used for training. The training procedure is restarted P times, each time with a different sample in the validation set. Such a cross-validation procedure is useful for very small data sets.

1.8 Further Reading

The book of Bishop [4] emphasized the relation of supervised learning in Neural Networks. to classical statistical methods which are used in the context of pattern recognition problems. As a first introduction to Neural Networks the book of Rojas [9] can be recommanded for its use of graphical illustrations which help the reader to visualize the theorectical concepts. A comprehensive treatment of Neural Networks is given in the book of Haykin [5]. The now classical textbook of Hertz, Krogh, and Palmer [6] is still highly recommandable.

People who are interested in the history of the field should consult Minsky and Papert [7] as well as the collections of classical papers [1], [8]. The original paper of Rosenblatt on Perceptrons and the one of Widrow and Hoff on ADALINE are, for example, reprinted in [1]. The PDP-book shows the enthousiasm of neural network research in the eighties [10]. An overview of the state of the art of neural networks as models of biological phenomema is given in the Handbook of brain theory [3]. All of the cited books contain extensive lists of references to the original literature.

Bibliography

[1] J A Anderson and E Rosenfeld, editors. *Neurocomputing: Foundations of research.* MIT-Press, Cambridge Mass., 1988.

[2] Everyday applications of Neural Networks. Special issue. In *IEEE Transactions in Neural Networks*, volume 8, pages 825–964, 1997.

[3] M. A. Arbib. *The handbook of brain theory and neural networks.* MIT Press, Cambridge, MA, 1995.

[4] C. M. Bishop. *Neural Networks for Pattern Recognition.* Clarendon Press, Oxford, 1995.

[5] S. Haykin. *Neural Networks.* Prentice Hall, Upper Saddle River, NJ, 1994.

[6] J Hertz, A Krogh, and R G Palmer. *Introduction to the theory of neural computation.* Addison-Wesley, Redwood City CA, 1991.

[7] M. L. Minsky and S. A. Papert. *Perceptrons.* MIT Press, Cambridge Mass., 1969.

[8] G. Palm. *Brain Theory.* Springer, Berlin, 1986.

[9] R. Rojas. *Neural networks: a systematic introduction.* Springer, Berlin, Heidelberg, 1996.

[10] D. E. Rumelhard, J.L. McClelland, and the PDP research group. *Parallel distributed processing: Explorations in the microstructure of cognition. Vol. 1: Foundations.* MIT Press, Cambridge Mass., 1986.

[11] V. Vapnik. *The Nature of Statistical Learning Theory.* Springer-Verlag, New York, 1995.