# Perceptrons

Peter Sussner (sussner@ime.unicamp.br)

State University of Campinas
Institute for Mathematics, Statistics, and Scientific Computing
Department of Applied Mathematics
Campinas, SP, Brazil

## ABSTRACT

Perceptrons have been on the forefront of neural network research since its beginning. This paper gives a brief review of the perceptron concept and attempts to point out some critical issues involved in the design and implementation of multi-layer perceptrons.

**Keywords —** Single-layer perceptron, multi-layer perceptron, neural network training, learning, data modelling, generalization, testing, high speed parallel VLSI, stochastic perceptron, $k$-blocking distributions

## 1. INTRODUCTION

One of the most exciting developments during the early days of *neural networks* was the *perceptron*. The perceptron which was introduced by Frank Rosenblatt [35, 36] is based on a network of binary decision units [29] which model nerve cells in the human brain. The perceptron is used to classify or recognize patterns, i.e. "to perceive". In other words, the perceptron acts as a function from a set a patterns to a set of classes. Rosenblatt's *Perceptron Convergence Theorem* provided an algorithm which enables the perceptron to learn every mapping it can represent [36, 30, 1, 18]. This result on the learning ability of perceptrons gave rise to the hope that it would be possible to construct a model of the human brain in terms of a multiple-layer perceptron.

These high expectations were crushed in the eyes of many researchers by Minsky and Papert [30] who pointed out the limitations of single-layer perceptrons. Minsky and Papert's main observation was that some very simple pattern recognition problems, namely the linearly inseparable problems, cannot be solved by means of a single-layer perceptron. Their most famous counterexample is the XOR-problem which consists of associating the binary patterns $(0, 0)$ and $(1, 1)$ with one class, and associating the patterns $(1, 0)$ and $(0, 1)$ with another class. They also addressed the scaling problem, i.e. the fact that training times increase very rapidly for certain problems as the number of input lines increases. Their criticism of neural networks is valid and mathematically accurate and it led to a highly pessimistic view of the future of neural networks at the time. Minsky and Papert did not take into account, however, that multi-layer versions of the perceptron are capable of solving an arbitrary dichotomy.

The advent of *backpropagation* in the mid 1980's renewed major interest in neural networks since it provided for a practicable algorithm to train multi- layer perceptrons [10]. The simplicity of standard backpropagation is one of the reasons why multi-layer perceptrons are still the most widely used kind of neural networks. Other factors include the adaptability, ease of implementation, and the variety of applications in pattern recognition, control, and prediction.

The organization of this review paper is as follows: First we introduce the reader to the neural network terminology as well as the concepts of single- layer and multilayer perceptrons. In Chapter 4 we discuss several training algorithms of multilayer-perceptrons. Chapter 5 deals with the neural network's ability to model the data. Chapter 6 addresses VLSI implementations of multi-layer perceptrons. Finally, we present a statistical perceptron model called stochastic perceptron.

## 2. GENERAL NEURAL NETWORK CONCEPTS

### 2.1. Introduction

Since the early days of computer science it has become evident that conventional computers lack certain abilities that every human being possesses. In particular, these machines do not display a form of intelligent behavior. There have been two approaches geared at improving this situation. One is based on *symbolism* and the other one is based on *connectionism*. The former approach models intelligence in terms of computer programs which are able to manipulate symbols given a certain amount of "knowledge" and following a certain set of rules. The connectionist approach to introducing intelligence to computer systems relies on the hope that it is possible to model the structure of the biological neural systems such as the human brain. A biological nervous system consists of a network of neurons which continually receive and transmit signals. A simple model of a biological neuron consists of a processing element receiving several inputs.

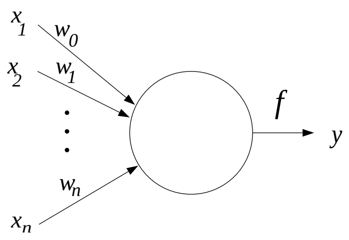


Fig. 1. A simple model of a neuron.

The symbols $x_1, \ldots, x_n$ represent the strengths of the impulses. The *synaptic weights* or connection strengths — denoted by the symbols $w_1, \ldots, w_n$ — interpret the role that the synapses play in the transmission of impulses. The output signal is represented by the symbol $y$. The dependence of the output $y$ on is given by the following rule:

$$y = f\left(\left[\sum_{i=1}^{n} w_i \cdot x_i\right] - \theta\right) , \tag{1}$$

where $\theta$ is a *threshold value* or *bias* and$f$ is the neuron's *activation function*. One of the most commonly used activation functions is the Heaviside step function given by

$$\begin{aligned} f \, : \, \mathbb{R} &\rightarrow \mathbb{R} \\ x \, &\rightarrow \begin{cases} 1 & if \;\; x \geq 0 \\ 0 & else \; . \end{cases} \end{aligned} \tag{2}$$

The neurons in an artificial neural network are sometimes also called *nodes* or *units*.

### 2.2. Neural Network Topologies

The topology of a neural network refers to it's framework and it's inter- connection scheme. In many cases the framework of a neural network consists of several layers of nodes. The literature on neural networks distinguishes between the following types of layers:

- *input layer:* A layer of neurons which receive external input from outside the network.
- *output layer:* the layer of neurons which produces the output of the network.
- *hidden layer:* a layer composed of neurons whose interaction is restricted to other neurons in the network.

A neural network is called a *single-layer neural network* if it has no hidden layers of nodes, or equivalently if it has just one layer of weights. A multi-layer neural network is equipped with one or more hidden layer of nodes. A *feedforward neural network* refers to a neural network whose connections point in the direction of the output layer. A *recurrent neural network* has connections between nodes of the same layer and/or connections pointing in the direction of the input layer. A schematic representation of an exemplar feedforward neural network is given in Figure 2.

### 2.3. Training and Learning

One of the principal components of intelligence is the capability of *learning*. Learning can be achieved in neural network by adjusting the connection weights of the network. There are two basic forms of learning in neural network: supervised learning and unsupervised learning. *Supervised learning* relies on the presentation of some input data and the corresponding target data. During the learning process a weight adjustment takes place which aims at minimizing the difference (error) between the target data and the output corresponding to the input data.

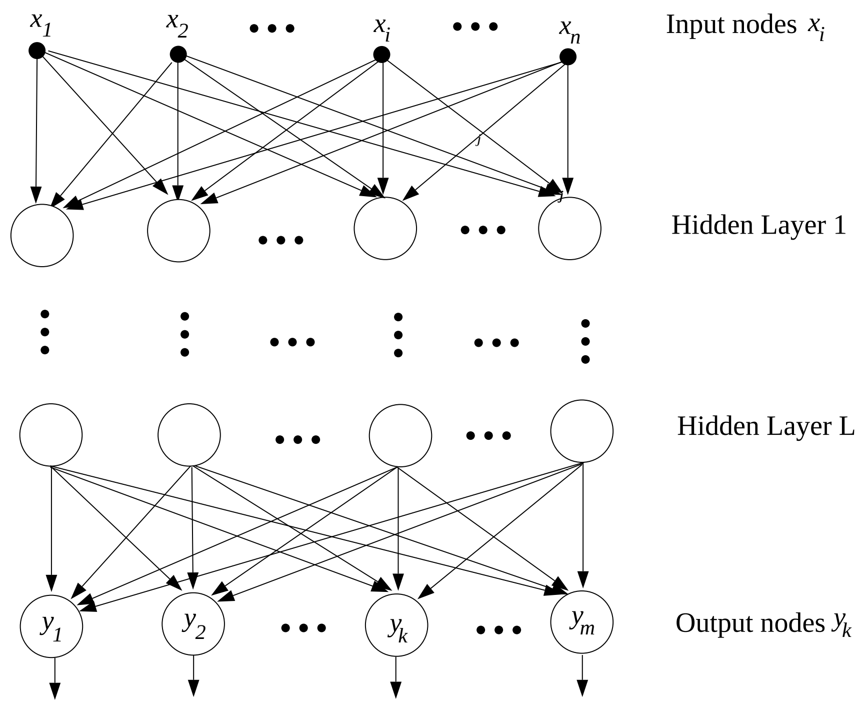


Fig. 2. Multi-layer feedforward neural network

In *unsupervised learning*, only input data are given to the network. In this setting, learning is based on grouping patterns into clusters. The weights are adjusted such that similar patterns produce the same output. *Training* refers to the actual presentation of input and possibly target data to the neural network. A neural network learns by being trained. (We have to mention that many authors prefer not to distinguish between learning and training.) The purpose of neural network training and learning is effective recall and generalization in the application phase. *Recall* consists of presenting and processing the same data which was used in the training and learning phase. Effective *generalization* is the ability of the network to perform well on new data and it is one of the main goals in the design of learning rules. The first learning rules emerged from the psychological studies of Donald Hebb and Frank Rosenblatt [17, 35]. Hebb's neuro physiological postulate stated that the synaptic connection strength between two neurons increases when one neuron repeatedly or persistently takes part in the activation of the other neuron or vice versa. Although the Hebbian learning rule represents a from of unsupervised learning, it can also be used in a supervised manner. Rosenblatt conceived a supervised learning rule for pattern recognition, where a teacher is necessary in order to indicate how to classify objects. The artificial neural network model he proposed in order to solve these problem was the perceptron.

## 3. INTRODUCTION TO PERCEPTRONS

### 3.1. Single-Layer Perceptrons

The single-layer perceptron serves as a classifier. It associates input patterns with one of two classes, say class 0 and class 1. The single-layer perceptron merely consists of an input layer and one node in the output layer. An input pattern x $= (x_1, \ldots, x_n)$ is classified as a class $1$ pattern if $\sum\limits_{i=1}^{n} w_i x_i \geq \theta$ , where w $= (w_1, \ldots, w_n)$ denotes the vector of the synaptic weights and where $\theta$ denotes the threshold parameter. The pattern x is classified as belonging to class $0$ if $\sum\limits_{i=1}^{n} w_i x_i < \theta$.

Figure 3 provides a schematic representation of a single-layer perceptron. The perceptron's activation function is the Heaviside step function of Eq.2. As a matter of convenience we used $w_0$ to denote $-\theta$. In this notation, the perceptron computes the output $y$ as $f\left(w_0 + \sum\limits_{i=1}^{n} w_i x_i\right)$ and the bias can be treated as an additional weight if we extend the input pattern x as follows: x $= (x_0, x_1, \ldots, x_n)$, where $x_0 = 1$.

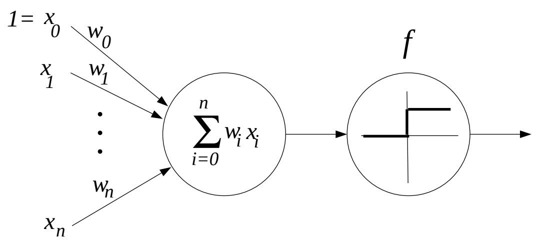


Fig. 3. Single-layer perceptron.

The equation $\sum\limits_{i=1}^{n} w_i x_i = \theta$ determines a hyperplane which is called the perceptron's decision surface. In the case where $n = 2$, the decision surface is a line. Two classes of patterns are called linearly separable if the two classes can be separated by means of a perceptron decision surface. Clearly, patterns belonging to two different classes cannot always be divided by such a decision surface. The XOR-problem provides a simple example of a situation where two classes of patterns are not linearly separable. XOR is a binary operator on $\{0,1\}^2$ such that for all $(a,b) \in \{0,1\}^2$:

$$a\;XOR\;b = \begin{cases} 0 & if\; a = b \\ 1 & else\;. \end{cases} \tag{3}$$

Thus, the XOR-operator divides the pattern space $\{0,1\}^2$ into two the subsets $C_0 = \{(0,0),(1,1)\}$ and $C_1 = \{(0,1),(1,0)\}$ .The points in the domain of the problem are plotted in Fig. 4. Open dots represent points in $C_0$. Solid dots represent points in $C_1$.

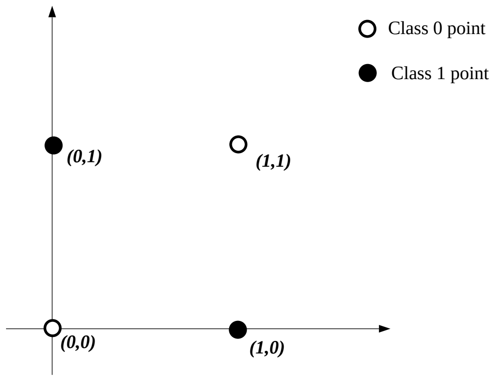


Fig. 4. Representation of domain for XOR.

### 3.2. Single-Layer Perceptron Learning

In 1962 Rosenblatt presented the perceptron convergence theorem which induces a supervised learning algorithm for solving arbitrary classification problems into two classes [36]. The algorithm modifies the weights at time $k+1$ in the direction of the current error $E(k)$ which is defined as the difference of the target output and the actual output at time $k$.

The algorithm can be described as follows. Suppose we are given a set of training patterns $\mathrm{x}^1, \mathrm{x}^2, \ldots, \mathrm{x}^m$. The order in which the patterns are read in does not matter. Initialize the step counter $k$ to be $0$ and the counter $p$ indicating the pattern number to be $1$. Let $\mathrm{w}(0) = (w_1(0), \ldots, w_n(0))$ denote the initial vector of the weights.

1. Set $\mathrm{x} = \mathrm{x}^p$ and compute the activation $y(k)$ for input pattern $\mathrm{x}$.
2. Compute the current output error $E(k)$ as follows:

$$E(k) = t^p - y(k) \ , \tag{4}$$

where $t^p$ is the target value for the pattern $\mathrm{x} = \mathrm{x}^p$ and $y(k)$ is the output value at time $k$.

3. Modify the vector $\mathrm{w}(k) = (w_1(k), \ldots, w_n(k))$ of the connection weights at time $k$ by adding the factor $\eta \cdot \varepsilon(k) \cdot \mathrm{x}$, i.e.:

$$\mathrm{w}(k+1) = \mathrm{w}(k) + \eta \cdot E(k) \cdot \mathrm{x} \ . \tag{5}$$

4. Increment the step counter $k$. Update the counter $p$ representing the pattern number as follows:

$$p \ := \begin{cases} p+1 & if \ p \leq m-1 \\ 1 & if \ p = m \ . \end{cases} \tag{6}$$

Figure 5 illustrates an application of the perceptron learning algorithm. Class 1 points have been plotted with diamonds and Class 0 points have been plotted with crosses. The lines plotted in the figure represent decision surfaces after $k = 0, 20, 40,$ and $80$ training patterns have been presented to the single-layer perceptron.

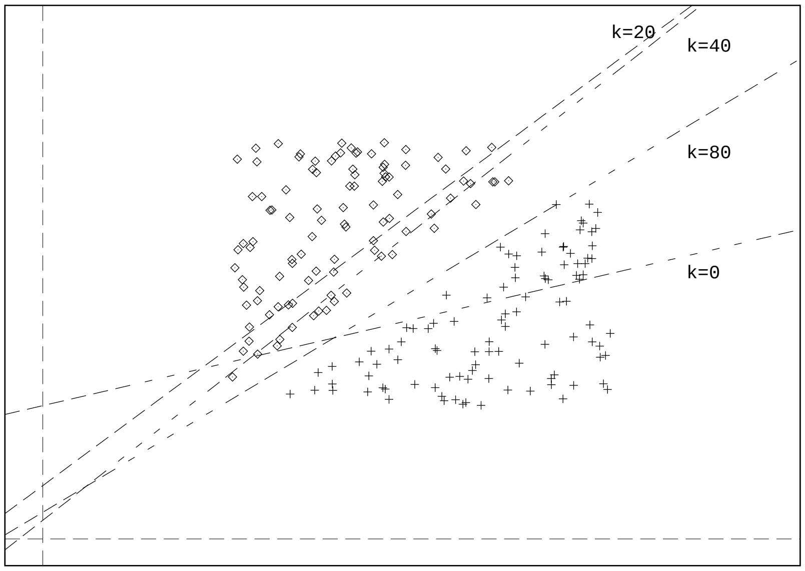
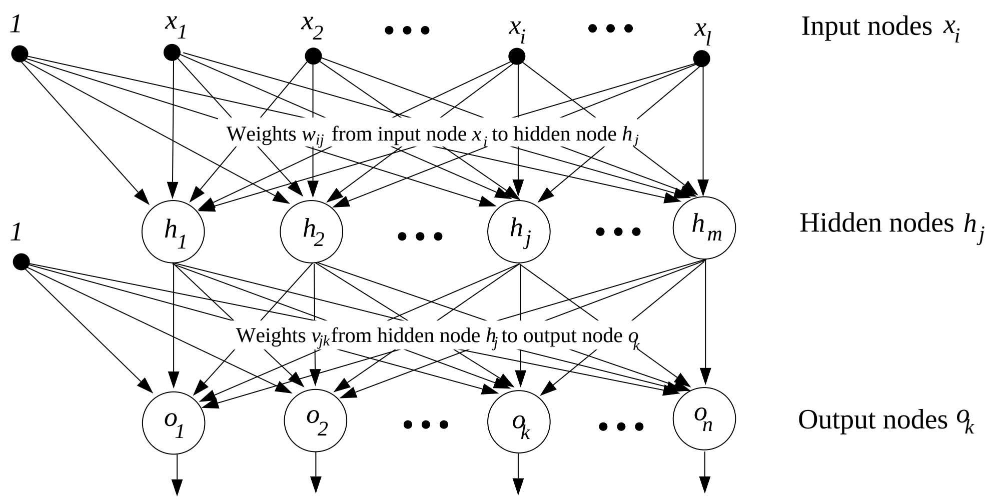


Fig. 5. The perceptron's decision surface after step $k$ of the learning algorithm.

This algorithm is guaranteed to find a weight adjustment which solves the classification problem in a finite number of steps if the given two classes are linearly separable. However, the algorithm does not converge at all when the classes are linearly inseparable and it is difficult to recognize this situation beforehand. Minsky and Papert recognized the heart of the problem: single-layer perceptrons only have one layer of adaptive weights [30]. A suitable data representation may transform an originally linearly inseparable problem into a linearly separable one. However, once chosen, the data representation is fixed.

### 3.3. Multi-layer Perceptrons

Multi-layer perceptrons are feedforward neural networks with at least one hidden layer of nodes. Thus, they have at least two layers of adaptive weights. Figure 6 illustrates the framework and the connection scheme of a two-layer perceptron. The framework of a multi-layer perceptron may include a bias parameter in every layer of nodes. As before, this situation can be modelled by extending the input vectors by an additional component of $1$.



1
$x_1$
$x_2$
$x_i$
$x_l$
Input nodes $x_i$
Weights $w_{ij}$ from input node $x_i$ to hidden node $h_j$
1
$h_1$
$h_2$
$h_j$
$h_m$
Hidden nodes $h_j$
Weights $v_{jk}$ from hidden node $h_j$ to output node $o_k$
$o_1$
$o_2$
$o_k$
$o_n$
Output nodes $o_k$

Fig. 6. Two-layer perceptron.

The activation functions of multi-layer perceptrons are either *threshold* functions or belong to the family of *sigmoidal functions* whose graphs are characterized by a monotonically increasing s-shaped curve. A generic sigmoid activation function is given as follows:

$$f(x) = \frac{a}{1 + e^{-bx+c}} + d \; . \tag{7}$$

The parameters $a$, $b$, $c$, and $d$ control the height, the slope, and the horizontal and vertical justification of the curve. The following figures represent sigmoidal functions for different parameter values.

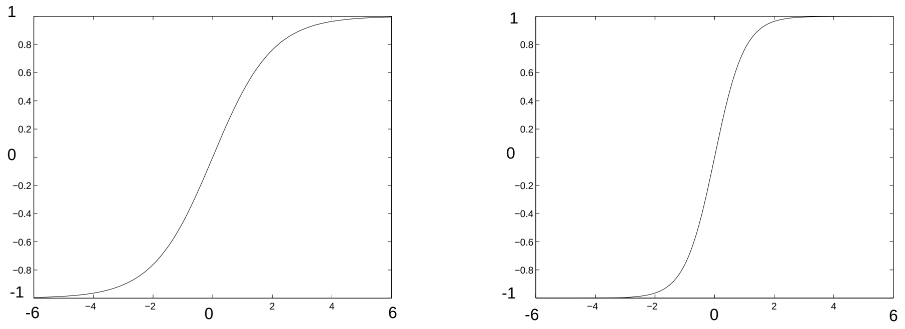


Fig. 7. A bipolar logistic function with $a = 2$, $b = 1$, $c = 0$, and $d = -1$ on the left and the hyperbolic tangent function ($b = 2$, $c = 0$, and $d = -1$) on the right:

In the previous section, we noted that single layer perceptrons can separate patterns positioned on different sides of a hyperplane. A two-layer perceptron with threshold units can form a single convex region as a decision boundary. This property is due to the fact that the output unit can be used computing a multivariable logical AND of the hidden units by setting the bias to $-m$, where $m$ is the number of hidden units. Setting the threshold parameter to $-1$ would generate a multivariable OR function. Thus, three-layer perceptrons with threshold activation functions can approximate arbitrary decision boundaries provided that the number of hidden units is variable. Formally, the approximation can be achieved in terms of a fine grid of hypercubes. Gibson & Cowan as well as Blum & Li have pointed out that two-layer perceptrons with threshold activation functions are incapable of approximating arbitrary decision regions [15, 8]. Of course, this statement does not preclude the fact that there exist examples of complex, non-convex decision regions which can be generated by two-layer perceptrons with threshold units [45, 22] . Feedforward neural networks with two layers of weights and sigmoidal activation functions are able to approximate arbitrary decision regions to arbitrary accuracy. This theorem follows from a result stating that these perceptrons are able to approximate arbitrarily well any continuous mapping from $\mathbb{R}^n$ to $\mathbb{R}$ . Feedforward neural networks with differentiable activation functions such as sigmoidal functions can be trained efficiently using the *backpropagation algorithm*.

## 4. Training and Learning in Multi-layer Perceptrons

### 4.1. Backpropagation

Training in multi-layer perceptrons is performed in a supervised form and aims at minimizing a previously defined error function. Rosenblatt's Perceptron Learning Algorithm determines a sequence of weight adjustments such that the error will vanish in a finite number of steps. The weights are modified according to their error contribution. Similar learning algorithms do not exist for feedforward neural networks with threshold activation functions and multiple layers of weights.

Fortunately, there are proven methods for the minimization of differentiable functions such as sigmoids which resemble threshold functions. *Gradient descent*, the simplest and most commonly used of these optimization methods, relies on the partial derivatives of the (error) function in order to determine a local minimum. Strictly speaking, backpropagation only refers to the calculation of the error function derivatives. The importance of backpropagation lies in the fact that the evaluation of the error function derivatives can be performed in $O(W)$ operations, where $W$ is the number of weights and biases in the network, instead of $O\big(W^2\big)$ which are required to evaluate the partial derivatives directly. Most training algorithms for multi-layer perceptrons consist of a backpropagation phase and a weight modification phase. Many authors refer to the entire training algorithm as backpropagation.

Suppose the training set consists of $P$ patterns $\mathrm{x}^1, \ldots, \mathrm{x}^P$. Each pattern vector $\mathrm{x}^p$, where $p$ ranges from $1$ to $P$, produces an output vector $\mathbf{y}^p = \left(y_1^p, \ldots, y_m^p\right)$. The total error $E$ is measured in terms of the errors $E^p$ where $E^p$ is the error generated by an individual pattern $\mathrm{x}^p$. In most cases, the error can be written as

$$E = \sum_{p=1}^{P} E^p \; . \tag{8}$$

One of the most common choices for the error measure $E^p$ is

$$E^p = \frac{1}{2} \sum_{l=1}^{m} \left(t_l^p - y_l^p\right) \; , \tag{9}$$

where $\mathbf{t}^p = \left(t_1^p, \ldots, t_m^p\right)$ is the target output for the pattern vector$\mathrm{x}^p$. The errors $E$ and $E^p$ can be viewed as functions of all the weights in the network. Due to the representation of $E$ as a sum of the individual errors $E^p$, the problem of determining the derivatives of$E$ with respect to the weights reduces to the problem of determining the derivatives of $E$ with respect to the weights.

The description of the algorithm for evaluating the derivatives of $E^p$ with respect to the weights will reveal that these derivatives can be expressed as a product of the activation of a certain node and another parameter $\delta$ corresponding to another node. The algorithm performs the following basic steps:

1. Present the pattern $\mathrm{x}^p$ to the network and compute the activations of the nodes.
2. Compute the $\delta$-parameters for the output units (Eq.15).
3. Use the $\delta$-parameters of the units in the layer $l+1$ to compute the $\delta$-parameters of the units in the $l$-th layer (Eq.17, “Backpropagation”).
4. Evaluate the required derivatives by using the product representation of the derivatives mentioned above (Eq.14).

We now describe in detail the general method for evaluating the derivatives of the error function $E^p$ in a multi-layer feedforward neural network with differentiable activation functions. Suppose that the pattern $\mathrm{x}^p$ has been presented to the network. From now on its output is simply denoted by $\mathbf{y} = (y_1, \ldots, y_m)$ instead of $\mathbf{y}^p = \left(y_1^p, \ldots, y_m^p\right)$. Let $z_i^l$ be the activation of the $i$-th node in the $l$-th layer of nodes. The weight connecting the $i$-th unit of layer $l$ to the $j$-th unit of layer $l+1$ is denoted by $w_{ji}^l$.

Each unit of a hidden layer or the output layer first computes a weighted sum of its inputs of the form

$$s_j^{l+1} = \sum_i w_{ji}^l \cdot z_i^l \; . \tag{10}$$

The activation of the $j$-th unit of layer $l+1$ is obtained by applying a differentiable activation function $g$ to the sum $s_j^{l+1}$:

$$z_j^{l+1} = g\left(s_j^{l+1}\right) . \tag{11}$$

The activation functions may vary in different layers of nodes. However, we chose to leave away this distinction so as to avoid an unnecessary clutter of notation.

Since the weights $w_{ji}^l$ only influence the error $E^p$ via the summed input $s_j^{l+1}$ to the $j$th node of layer $l+1$, an application of the chain rule gives:

$$\frac{\partial E^p}{\partial w_{ji}^l} = \frac{\partial E^p}{\partial s_j^{l+1}} \cdot \frac{\partial s_j^{l+1}}{\partial w_{ji}^l} . \tag{12}$$

By Eq.10

$$\frac{\partial s_j^{l+1}}{\partial w_{ji}^l} = z_i^l . \tag{13}$$

Denoting $\frac{\partial E^p}{\partial s_j^{l+1}}$ by $\delta_j^{l+1}$, we can write

$$\frac{\partial E^p}{\partial w_{ji}^l} = \delta_j^{l+1} \cdot z_i^l . \tag{14}$$

If $L$ represents the number of layers of neurons, then the symbols $\delta_1^L, \ldots, \delta_m^L$ denote the $\delta$-parameters of the output units. These parameters can be immediately computed as follows:

$$\delta_j^L = \frac{\partial E^p}{\partial s_j^L} = \frac{\partial E^p}{\partial y_j} \cdot g'\left(s_j^L\right) . \tag{15}$$

Using the chain rule for partial derivatives again, we obtain the following formula for the $\delta$'s of the hidden units. These parameters are denoted by the symbol $\delta_j^l$ where $l$ ranges from $2$ to $L-1$.

$$\delta_j^l = \frac{\partial E^p}{\partial s_j^l} = \sum_k \frac{\partial E^p}{\partial s_k^{l+1}} \cdot \frac{\partial s_k^{l+1}}{\partial s_j^l} = \sum_k \delta_k^{l+1} \cdot \frac{\partial s_k^{l+1}}{\partial s_j^l} . \tag{16}$$

In view of Eq.10, the partial derivative of $s_k^{l+1}$ with respect to $s_j^l$ is given by $w_{kj}^l \cdot g'\left(s_j^l\right)$. Note that pulling the factor $g'\left(s_j^l\right)$ out of the summation yields the following *backpropagation formula*:

$$\delta_j^l = g'\left(s_j^l\right) \cdot \sum_k w_{kj}^l \cdot \delta_k^{l+1} . \tag{17}$$

The term on the right hand side is computed in Step 3 of the backpropagation algorithm in order to determine the $\delta$'s of layer $l$ once the $\delta$'s of layer $l+1$ are known.

### 4.2. Multi-layer Perceptron Training

We now discuss some training algorithms for multi-layer perceptrons which utilize error back-propagation. From now on we will simply enumerate the weights in the form $w_1, w_2, \ldots, w_W$, where $W$ is the total number of weights. Thus the weights form a vector $\mathbf{w} = (w_1, w_2, \ldots, w_W)$. The gradient $\nabla E$ of an error function $E$ with respect to the weights consists of the partial derivatives $\frac{\partial E}{\partial w_1}, \frac{\partial E}{\partial w_2}, \ldots, \frac{\partial E}{\partial w_W}$.

Before training it is necessary to initialize the weights and biases. The vector of these initial weights is denoted by $\mathbf{w}(0) = (w_1(0), w_2(0), \ldots, w_W(0))$. Furthermore, a step size parameter $\eta$ must be chosen. The training algorithms alternate between a backpropagation phase and a weight modification phase. The latter phase consist of adding a vector $\Delta\mathbf{w}(k)$ to the current weight vector $\mathbf{w}(k)$ at time $k$, where the modification $\Delta\mathbf{w}(k)$ depends on the step size parameter $\eta$. The simplest of these algorithms uses *gradient descent* for weight adjustment and is described below:

Set $k = 0$ and execute Steps I and II until a stopping criterion is met.

I. Perform the backpropagation phase for all training patterns $\mathbf{x}^1, \ldots, \mathbf{x}^P$ which produces the gradients $\nabla E^1, \ldots, \nabla E^P$.

II. Update the weights as follows:

$$\begin{aligned} \mathbf{w}(k+1) &= \mathbf{w}(k) + \Delta\mathbf{w}(k)\ , \\ where\ \ \Delta\mathbf{w}(k) &= \ -\eta \sum_{p=1}^{P} \nabla E^p|_{\mathbf{w}(k)}\ . \end{aligned} \tag{18}$$

Increment the counter $k$.

Note that in the training algorithm above the weights are updated every time the whole set of training patterns has been presented to the network. The algorithm is said to operate in *batch, off-line* or *deterministic* mode. If the training set is large, this technique leads to a slow learning process since it involves a sweep through the whole training set for each weight update. An alternative method is the *real-time, on-line* or *stochastic* mode where the weights are adjusted after each pattern presentation as follows:

Set $k = 0$, $p = 1$, and execute Steps I and II until a stopping criterion is met.

I. Perform the backpropagation phase for the training pattern $\mathbf{x}^p$ yielding partial derivatives $\nabla E^p$.

II. Update the weights as follows:

$$\begin{aligned} \mathbf{w}(k+1) &= \mathbf{w}(k) + \Delta\mathbf{w}(k)\ , \\ where\ \ \Delta\mathbf{w}(k) &= \ -\eta \nabla E^p|_{\mathbf{w}(k)}\ . \end{aligned} \tag{19}$$

Increment the counter $k$. The new value of $p$ is given by $(p+1) mod P$.

Since both versions of this algorithm are based on gradient descent, they only implement a search for a *local minimum*. The chances for attaining the global minimum of the error function can be increased by executing several independent training procedures with randomly initialized weights. Another possibility would be to chose a more complex architecture with a larger number of weights, since the local minima are usually lower in this case.

In many instances, multi-layer perceptron learning requires a huge number of sweeps through the whole training set or *epochs* until the error function reaches an acceptably low value. The principal reason for the phenomenon is that the error surface often has narrow ravines, i.e. regions whose curvature is very large in one direction and rather small in the orthogonal direction. In this situation, the choice of the learning parameter $\eta$ is problematic. A large step size may cause divergent oscillations across the ravine. A small value for $\eta$ will lead to a slow learning process since the weight vector will first slowly converge to the bottom of the ravine before it crawls along the bottom and finally reaches a local minimum. The technique presented in the next section constantly adapts the step size in order to improve the learning speed of the multi-layer perceptron.

### 4.3. Adaptive Step Size Technique

In this technique each weight $w_i$, where $i = 1, \ldots, W$, has an individual step size parameter $\eta_i$ which is changed at every iteration. The rate of change depends on the signs of successive gradient components. In real-time mode, we obtain the following equations:

$$\eta_i(k+1) = \begin{cases} \rho \cdot \eta_i(k) & if \ \frac{\partial E^p}{\partial w_i}|_{\mathbf{w}(k+1)} \ and \ \frac{\partial E^p}{\partial w_i}|_{\mathbf{w}(k)} \ have \ the \ same \ sign \\ \sigma \cdot \eta_i(k) & else \ , \end{cases}$$
$$where \ \ \rho > 1 \ \ and \ \ \sigma < 1 \ . \tag{20}$$

By this simple strategy, an individual step size $\eta_i$ will be increased if the current weight update is performed in the same direction as the previous one, i.e. further weight updates in this direction are required. The step size parameter $\eta_i$ is decreased if the current weight adjustment is performed in a different direction than the previous one, i.e. the weight $w_i$ was previously changed by a too large amount. The adaptive step size technique is able to deal particularly well with ravines which are more or less parallel to some axis corresponding to an individual weight $w_{ji}$. If the error surface forms a ravine that is oblique to all axes, Silva and Almeida suggest a combination of this technique with the momentum technique presented below [39, 40].

### 4.4. Momentum Technique

In the momentum technique, an additional term $\alpha \cdot w_i(k)$ is added to each weight update term $\Delta w_i(k)$ [33]. In batch mode, this procedure results in the following weight update equation:

$$\begin{gathered} \mathbf{w}(k+1) = \mathbf{w}(k) + \Delta\mathbf{w}(k) \ , \\ where \ \ \Delta\mathbf{w}(k) = \ -\eta \sum_{p=1}^{P} \nabla E^p|_{\mathbf{w}(k)} + \alpha \cdot \mathbf{w}(k) \\ and \ \ 0 \leq \alpha < 1 \ . \end{gathered} \tag{21}$$

Clearly, $\Delta\mathbf{w}(k) = \ -\eta \cdot \nabla E^p|_{\mathbf{w}(k)} + \alpha \cdot \mathbf{w}(k)$ in real-time mode.

The terms $\alpha \cdot w_i(k)$ is called the *momentum terms*. This term has a cumulative effect if successive weight updates occur in similar direction. On the other hand, the contributions from successive momentum terms will tend to cancel out if the weight vector oscillates from one ravine wall to another ravine wall. The intended effect of the momentum technique is to lead the weight vector faster in the direction of the local minimum. The choice of the momentum parameter $\alpha$ is crucial to achieve this goal. A small parameter $\alpha$ will normally introduce little improvement compared to the regular gradient descent algorithm. A large choice of $\alpha$ may drive the weight vector up the ravine wall (and possible out of the ravine) at the location of a bend in the ravine, particularly if a large amount of momentum has previously been acquired.

### 4.5. Enhanced Gradient Descent

As we have seen, simple gradient descent with momentum does not guarantee convergence — not even to a local minimum. Several methods exist to enhance the robustness of gradient descent. One of the major problems concerning simple gradient descent with momentum is the fact that its effectiveness depends on an appropriate choice for the step size parameter $\eta$ and the learning parameter $\alpha$, both of which have to be chosen by trial and error. Instead of adopting this time-consuming random approach we might prefer to choose the parameters $\eta$ and $\alpha$ automatically.

One of these automatic approaches is called *bold driver technique* [43, 3]. The first step is to check how the error function has changed after each step of the gradient descent.

1. *Increase in error:* The weight vector is reset to its previous value, the step size parameter is multiplied by a number $\sigma < 1$ (typical choice: $0.5$), and the momentum coefficient $\alpha$ is set to zero. [The local minimum must have been overshot. Therefore, a new attempt to reduce the error is made with a smaller step size and without momentum.]

2. *Decrease in error:* The weight change is accepted. and the step size parameter $\eta$ is multiplied by a number $\rho > 1$ (typical choice: $1.1$) [A decrease in error suggests that the algorithm is on its way towards a local minimum.]

**4.6. Line Search**

All the techniques for weight adjustment we have discussed so far proceeded in a certain direction with a certain step size given by the learning parameters. The procedure of *line search* is based on the following idea: Once the direction of the next step is fixed, the optimal reduction of an error function $E$ can be achieved by minimizing $E(\mathbf{w}(k) + \lambda \mathbf{d}(k))$ with respect to $\lambda \in \mathbb{R}$.

Line Search can be employed when training multi-layer perceptrons both in batch mode and in real-time mode. Choose an initial weight vector $\mathbf{w}(0)$ and set $k = 0$. Perform the following steps until a stopping criterion is met:

1. Determine a search direction $\mathbf{d}(k)$.
2. Minimize $E(\mathbf{w}(k) + \lambda \mathbf{d}(k))$ with respect to $\lambda$. Let $\lambda_\circ$ be the variable where the minimum is adopted.
3. Update the weights by setting $\mathbf{w}(k+1) = \mathbf{w}(k) + \lambda_\circ \mathbf{d}(k)$ and increment the counter $k$.

Successive gradient vectors seem to provide the best choice for the sequence of search directions at first glance. However, practical examples show that successive steps in the opposite direction of the gradient will usually take many iterations to arrive at the minimum. The last sections of this chapter will deal with different training algorithms which are based on line search [28].

**4.7. Conjugate Gradient (CG) Method**

A better choice for the search directions $\mathbf{d}(k)$ is the socalled conjugate gradient direction [19]. A version of the general line search algorithm outlined above is called *conjugate gradient algorithm* if $\mathbf{d}(1) = -\nabla E|_{\mathbf{w}(1)}$ and $\mathbf{d}(k)$ is in the socalled conjugate gradient direction for all $k > 1$. The conjugate gradient directions satisfy

$$\mathbf{d}(k+1) H \mathbf{d}(k) = 0 \; , \tag{22}$$

where $H$ denotes the Hessian matrix. An explicit evaluation of the Hessian matrix is unnecessary. The Hestenes-Stiefel formula, the Polak-Ribiere formula or the Fletcher-Reeves formula provide ways to compute the new conjugate gradient direction $\mathbf{d}(k+1)$ using only $\mathbf{d}(k)$ and gradient information [34]. Backpropagation can be employed again for finding the gradients.

If the error function is a quadratic polynomial, this algorithm is guaranteed to find a minimum of a quadratic error function in $W$ steps. In the case of a general non-quadratic error function the algorithm makes use of an approximation in terms of a quadratic error function in the neighborhood of a given point. These approximations are usually updated after a sequence of $W$ iterations. Due to the difference between the actual error function and the quadratic approximation, the algorithm needs to be run for many iterations until a stopping criterion is reached.

## 4.8. Newton's Method

Newton's method selects $-\big(H^{-1} \cdot \nabla E\big)|_{\mathbf{w}}$, where $H$ denotes the Hessian matrix, as a search direction. The vector $-\big(H^{-1} \cdot \nabla E\big)|_{\mathbf{w}}$, known as the Newton direction or the Newton step, points directly towards the minimum of the error surface, if the error function is a quadratic polynomial. In the general case, a quadratic approximation of the error function is chosen and the Newton step, involving the evaluation of the Hessian is applied iteratively. This approach involves several problems:

1. If the Hessian is not positive definite, the Newton step is not guaranteed to move towards a local minimum. It may move towards a local maximum or a saddle point instead. The *model trust region* approach resolves this problem by adding a suitably large multiple of the identity matrix to the Hessian, yielding a positive definite matrix as a result [28]. A closer look reveals that hereby a compromise between Newton's method and the standard gradient descent method is formed.
2. The stability of Newton's Method is affected if the step size, computed by a line search, takes the weight vector outside the validity of the quadratic approximation. This problem can be counteracted by forming a new quadratic approximation in the neighborhood of the current point.
3. The Hessian must be evaluated and inverted at each iteration of the algorithm. The evaluation of the Hessian costs $O\big(PW^2\big)$ steps and its inversion costs $O\big(W^3\big)$ steps in terms of the number of patterns $P$ and the number of weights and biases $W$. In order to avoid the execution of these computationally expensive operations, we might simply neglect all off-diagonal terms. This approximation of the Hessian reduces the computational cost significantly since the diagonal terms can easily be computed by means of backpropagation and the inversion of a diagonal matrix is trivial. However, this approach has turned out to be unsuccessful for many practical neural network applications where the Hessian is far from diagonal. Socalled Quasi-Newton Methods represent a practical approach to circumvent the direct calculation of the Hessian matrix.

### 4.9. Quasi-Newton (QN) Methods

Quasi-Newton methods are derived from Newton's method and adopt $-\left(H^{-1} \cdot \nabla E\right)|_{\mathbf{w}}$ as a search direction as well. However, instead of directly calculations the Hessian matrix and computing its inverse, Quasi-Newton methods iteratively construct an approximation of $H^{-1}$, using only first-order information in the process. The current method of choice for this construction is the Broyden-Fletcher-Goldfard-Shanno-Method [13].

### 4.10. Comparison of CG and QN Methods

QN methods are computationally more stable than CG methods. In contrast to CG methods, it is not necessary in these algorithms to perform the line searches with great accuracy in order to obtain a reduction of error. This property leads to a faster convergence of QN methods compared to CG methods.

On the other hand, the construction of the matrix approximating of $H^{-1}$ entails storage requirements of $O\left(W^2\right)$. Since CG methods only require $O(W)$ storage, they are preferred for large-scale problems involving a multitude of weights.

Recently a number of researchers have devised several low-storage QN methods which combine the speed advantages of CG methods with the linear storage requirements of QN methods [32, 14, 3].

## 5. Generalization

In the last chapter, we gave the impression that training only serves the purpose of effectively minimizing the error function, which measures the performance of the multi-layer perceptron on some set of training data. However, the most important role of training is to condition the network such that it generalizes well and *models all the data*. As mentioned in Chapter 2, the ability to generalize represents the most important component of the network's learning ability. *Generalization* refers to the network's performance in the application phase. Since it is either impossible or computationally prohibitive to include all problem data in the training process, the network should aim at predicting the structure of the problem data by detecting some structure in the training data. There are several techniques for measuring and improving the network's generalization performance. Many of these methods are geared at optimizing the size of the network which is an influential factor in the generalization capabilities of the network. Therefore, we consider it appropriate to make a few remarks on this topic beforehand.

### 5.1. Network Size

Supervised training with training data is analogous to fitting a curve through a number of data points reminiscent of polynomial curve fitting. The function corresponding to this curve is of a form which is determined by the architecture of the network. A multi-layer perceptron computes a function from $\mathbb{R}^n$ to $\mathbb{R}^m$ which is given by a concatenation of multiplications, additions , and sigmoidal or hardlimiting functions.

The function has a number of free parameters which correspond to the weights of the network. An insufficient amount of free parameters leads to a poor fit through the given data points yielding poor recall. By increasing the number of free parameters the curve can be better fitted through the given data points. For example, a polynomial of degree $n$ or higher can achieve a perfect fit to $n+1$ data points. However, by choosing a function with many free parameters to represent the data one risks to *over-fit to the data*, i.e. the curve will reveal large oscillations from one data point to another. Fig. 8 illustrates this principle with a polynomial of degree $10$ used for interpolation of $6$ data points. A polynomial with smaller degree would model the data points reasonably well without exhibiting oscillations. Good generalization results can be achieved if the curve which is the outcome of training not only lies in the vicinity of the training data points but also in the vicinity of the problem data points. Since the problem data points are unknown, their location has to be predicted based on the location of the training data points. In most cases, a smoother function with a smaller amount of free parameters provides a better basis for predicting the location of new, unknown data points.

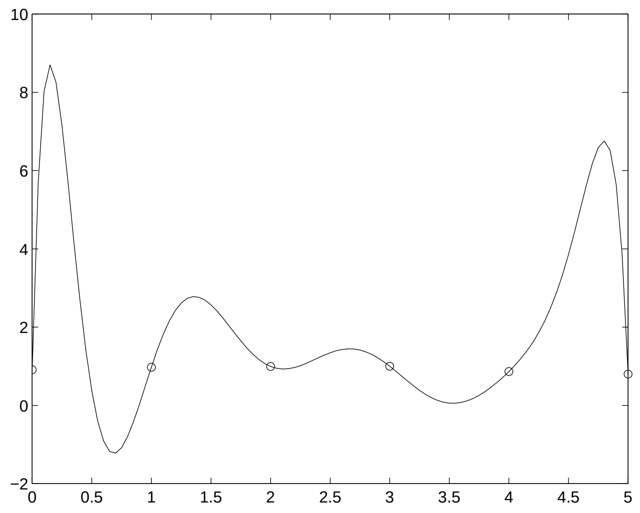


Fig. 8. Interpolation of data points by polynomial of degree $10$.

The preceding remarks indicate that a neural network with a sufficiently large number of weights can be trained to attain perfect recall of training data. The drawback is that a multitude of weights will lead to bad generalization behavior. Such a network will tend to detect non-existent regularities in the data. In many neural network applications, the training data is subject to some form of noise. Training a network with too many weights will have the undesired effect of modelling the noise instead of the structure of the problem data. The two basic alternatives to resolve this situation are:

- Reducing the size of the network;
- Increasing the problem complexity.

## 5.2. Network Pruning and Network Growing

Generally speaking, the optimal network topology is the "smallest" network that trains well. Of course, the simplest approach to finding the optimal network topology is to experiment with networks of different sizes. Although this approach is computationally very expensive it is still used in practice. The computational effort can be reduced by restricting the set of networks under consideration. For example, one might consider only two-layer perceptrons in the training experiments since networks of this form have the capability to approximate an arbitrary decision region.

*Network pruning* and *network growing* algorithms are more sophisticated approaches for optimizing the size of a neural network. Pruning techniques start with a relatively large networks which is iteratively reduced in size either by removing connections or complete units. The algorithms alternate between a training phase which consists of applying a standard training algorithm to the network and a (connection or node) removal phase. The removal phase involves the computation of the saliency, a measure of the importance of the weights, the nodes respectively. In each iteration some of the low-saliency objects are deleted. Network pruning techniques include *optimal brain damage* [9], *optimal brain surgeon* [16] as well as the skeletonizing algorithm of Mozer and Smolensky [31].

Network growing algorithms adopt a bottom-up approach: Starting from a small network, nodes are added to the network until a sufficiently small training error is reached. The most famous ones of these techniques is called *cascade correlation* [12]. The term cascade correlation is derived from the architecture of the network this algorithm constructs. The resulting networks have sigmoidal hidden unit and both feedforward and recurrent connections.

## 5.3. Regularization

In the previous section we remarked that neural networks can be viewed as functions and neural network training can be viewed as function interpolation. Among two neural networks producing similar error for the same set of training data the neural network corresponding to the smoother function tends to generalize best. Regularization is a technique which enhances the smoothness of the interpolating function by the addition of a *regularization term* to the error function $E$:

$$\tilde{E} = E + \rho C \; . \tag{23}$$

The function $C$ is a penalty term whose influence on the total cost function $\tilde{E}$ is controlled by the parameter $\rho \geq 0$. The total error $\tilde{E}$ to be minimized provides for a compromise between the fit the training data and function smoothness. Small $\rho$'s favor fitting the training data while small $\rho$'s favor smoothing out the interpolating function.

Choosing a regularizer of the form

$$C = \frac{1}{2} \sum_{p=1}^{P} \sum_{i=1}^{n} \sum_{k=1}^{m} \left( \frac{\partial y_k}{\partial x_i^2} \right)^2 |_{\mathbf{x}^p} \tag{24}$$

will directly penalize the curvature of the interpolating function, since curvature is measured in terms of second-order derivatives [6, 7].

A more simple and more common regularization term, called *weight decay*, is given by half of the sum of all the weights and biases:

$$C = \frac{1}{2} \sum_{i=1}^{W} w_i^2 \; . \tag{25}$$

Note that this choice of a penalty term forces the weights to become small. Small weights will cause the weighted sums of the inputs at a certain node to be small as well. Thus, the sigmoid is predominantly applied in a neighborhood of the origin where its behavior is almost linear. Only larger weights would lead the sigmoid to a region of larger curvature. Therefore, the function represented by the $n$-layer perceptron resembles a polynomial of degree $n$. Since multi-layer perceptrons typically have a small number of layers, the resulting function will be rather smooth.

The use of the term weight decay becomes clear when considering simple gradient descent for weight modification in the training algorithm. The weights are changed as follows:

$$\begin{gathered} \mathbf{w}(k+1) = \mathbf{w}(k) + \Delta \mathbf{w}(k) \; , \\ where \;\; \Delta \mathbf{w}(k) = \; -\eta \, \nabla \tilde{E}|_{\mathbf{w}(k)} = \; -\eta \left( \nabla E|_{\mathbf{w}(k)} \; + \rho \, \mathbf{w}(k) \right) \; . \end{gathered} \tag{26}$$

In the absence of $\nabla E$ we can write

$$\frac{\partial \mathrm{w}(k)}{\partial k} = -\eta\, \rho\, \mathrm{w}(k)\;, \tag{27}$$

since $\Delta \mathrm{w}(k)$ is the discrete form of the derivative of $\mathrm{w}$ with respect to $k$. Eq.27 has the unique solution

$$\mathrm{w}(k) = \mathrm{w}(0) \cdot e^{-\eta\, \rho\, \mathrm{w}(k)}\;, \tag{28}$$

yielding that all the *weights decay exponentially* to zero.

Some regularizers, which are similar to weight decay, are capable of acting as *weight pruners* by pushing the subset of weights which is least important in the reduction of the original error $E$ towards zero while leaving other weights large. This property leads to an algorithm which prunes all the weights at each iteration that fall below a certain threshold. Examples include *weight elimination* and *linear decay* [44, 46].

Recently, some authors have proposed the choice of a penalty function $C$ which is tailored to achieve *robust classifications* and good generalization performance [24, 11]. In this case, robustness refers to lack of sensitivity with respect to small perturbations in the input space, e.g. due to noise. The robustness of the network mapping will also lead to smoothness of the interpolating function the property which is responsible for generalization capabilities. The sensitivity of the multi-layer perceptron is given by the derivatives of the output $\mathrm{y}$ with respect to the input $\mathrm{x}$. If $\tilde{\mathrm{y}}$ denotes the output corresponding to a specific input $\tilde{\mathrm{x}}$, we obtain the sensitivity component $\frac{\partial y_i}{\partial x_k}$ at location $\tilde{\mathrm{x}}$ as follows:

$$\frac{\partial y_i}{\partial x_k}|_{\tilde{\mathrm{x}}} = \sum_{j_1,\ldots,j_{L-1}} w^{L}_{ij_{L-1}} w^{L-1}_{j_{L-1}j_{L-2}} \ldots w^{1}_{j_1 k}\, f'(\tilde{y}_i) f'\left(\tilde{s}^{L-1}_{j_{L-1}}\right) \ldots f'(\tilde{s}^{1}_{j_1})\;, \tag{29}$$

where $\tilde{s}^l_j$ denotes the weighted sum computed at the $j$-th node of $l$-th hidden layer. Adding the sums of squares of all sensitivity components at all training patterns $\mathrm{x}^1, \ldots, \mathrm{x}^p$ to the original error function would encourage robust classification, but is computationally very expensive. Note that small sensitivity components can be obtained by producing either small weights like in weight decay or small derivatives of the hidden layer activations. A short look at the bipolar sigmoidal function and its derivative indicates that both goals cannot be accomplished at the same time. (Also note that a combination of medium-sized weights and derivatives will lead to a relatively large product.)

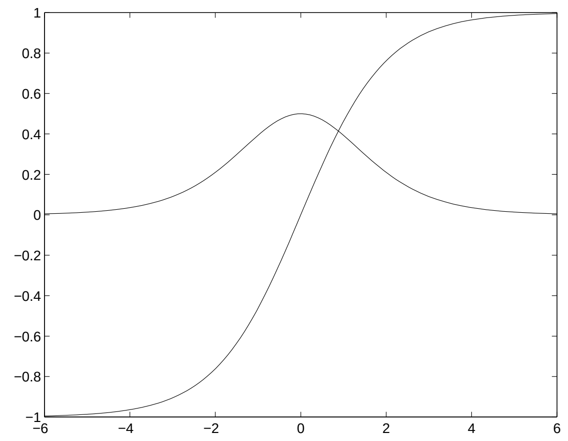


Fig. 9. $a = 1$, $b = 1$, $c = 0$, and $d = 0$: A logistic function.

Jeong and Lee choose to force the hidden layer activations into their saturation range by imposing a penalty in the form of the sums of all hidden layer activations. The learning process reveals some features of Hebbian learning when using a standard gradient descent method with error backpropagation. Initializing the weights with very small values provides a successful learning process in simulation experiments.

Drucker and Le Cun take a direct path to converting the original error function $E$ into a robust error function . The new error function $\tilde{E}$ consists of the sum of $E$ and a regularizer given by a multiple of the following term at a particular location $\tilde{\mathrm{x}}$:

$$C = \frac{1}{2}\left(\frac{\partial E}{\partial x_1}\right)^2 + \frac{1}{2}\left(\frac{\partial E}{\partial x_2}\right)^2 + \ldots + \frac{1}{2}\left(\frac{\partial E}{\partial x_n}\right)^2 . \tag{30}$$

Drucker and Le Cun's experiments yield an improved generalization performance over standard gradient descent with backpropagation, but also an increased computational effort due to the fact that calculating the appropriate derivatives requires two backpropagation phases.

### 5.4. Validation and Testing

Validation methods have been designed in order to select a network model with an optimal generalization performance. The choice of the neural network model may for example be between networks differing in the number of hidden units or between neural networks having different regularization parameters $\rho$. Validation methods require a set of *training data* and an independent set of data called *validation data*. The following steps are executed for all neural network models under consideration:

1. Train the neural net with the set of training data. The set of weights which minimizes the error function is fixed.

2. Check the generalization performance of the current model by evaluating the error function using the validation data as inputs.

The neural network model having the smallest error with respect to the validation data is selected.

Often the results of these approaches are confirmed by presenting yet another set of data called *test data* to the network. Use of this technique is meant to safeguard against overfitting to the validation data.

In practice, an independent set which can be designated to be a validation set is often not available. The method of *cross-validation* partitions a single data set into distinct subsets which serve as training data or validation data in different iterations of the general validation method described above. If the original data set is denoted by $S$, we have

$$\begin{aligned} S = S_1 \cup \ldots \cup S_n \\ and\ \ S_i \cap S_j = \varnothing\ \ \forall\, i = j\ . \end{aligned} \tag{31}$$

The validation method is executed $n$ times. In iteration $i$, the set give by the set difference $S - S_i$ acts as the training data set and the set $S_i$ acts as the validation set. If $|S_i| = 1$ for all $i = 1, \ldots, n$, we speak of the *leave-one-out* method.

### 5.5. Stopped Training

Stopped Training is a validation method which does not require a run through the whole set of training data. Instead, the quality of the current network is tested at each iteration by means of validation data. The goal is to select the network which performs best on the validation data. After the goal is met with some certainty, the training is halted. This strategy which is illustrated in Fig. avoids an overly tight fit to the training data.

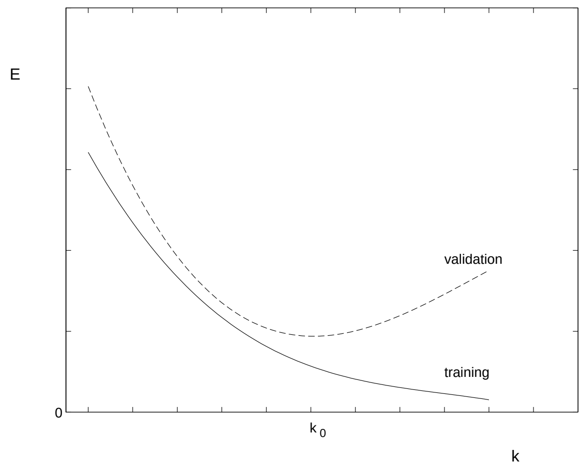


Fig. 10. Early stopping of the training process. The network found at iteration $k_0$ is selected.

### 5.6. VC-Dimension

In this section, we restrict our attention to multi-layer perceptrons with hard-limiting activation functions and binary inputs and outputs. Vapnik and Chervonenkis developed the concept of Vapnik-Chervonenkis-dimension, short VC-dimension, which provides an estimate of the generalization performance of a neural network in the worst-case scenario [42]. They express the network's generalization performance, denoted by $g$, in terms of the probability that an arbitrary pattern of the same distribution as the training patterns is classified correctly.

Vapnik and Chervonenkis provided an upper bound for the probability that the network's generalization performance differs by more than $\varepsilon$ from the fraction of patterns in the training set which are classified correctly. Note that for a perfect fit of the training patterns, we obtain an upper bound for the probability that $g < 1-\varepsilon$, or equivalently a lower bound for the probability that $g \geq 1 - \varepsilon$.

The upper bound mentioned above can be expressed in terms of the network's VC-dimension, which is denoted by $\dim_{VC}$. The quantity $\dim_{VC}$ is the largest number of patterns $P$ such that the neural network can solve every binary classification problem of $P$ patterns.

Let $U$ be the total number of units in a multi-layer perceptron with threshold activation functions. Baum and Haussler [5] showed that

$$\dim_{VC} \leq 2W \log_2 (eU) \ . \tag{32}$$

From this estimate they derived the following statement for $0 < \varepsilon \leq \frac{1}{8}$. Let $g_T$ denote the fraction of training patterns which is classified correctly. If the network has been trained with at least $\frac{W}{2}\log_2\left(\frac{U}{\varepsilon}\right)$ patterns such that $g_T \geq 1 - \frac{\varepsilon}{2}$ then the network will correctly classify a fraction $1-\varepsilon$ of future patterns with high probability.

They also proposed the estimate $P_{\min} \simeq \frac{W}{\varepsilon}$ for the minimal number of training patterns required to correctly classify at least a fraction of $1-\varepsilon$ patterns by means of a two-layer perceptron with threshold units. Thus, the number of training patterns needed in the case $\varepsilon = \frac{1}{10}$ amounts to $10W$. In practice we hope to encounter more favorable ratios than in this worst-case estimate, particularly after applying the techniques for improving the generalization performance we discussed in this chapter.

## 6. Hardware Implementations

The massively parallel structure of neural networks, in particular multi-layer perceptrons, cannot be exploited by means of software running on serial machines. Therefore, software implementations on conventional computers lack the speed requirements for many real-time applications such as high energy physics. General purpose parallel machines certainly provide the necessary speed and parallelism, but have a high price tag. More cost-effective alternatives are analog and digital VLSI implementations. We briefly review these methods with a special emphasis on multi-layer perceptron implementations.

### 6.1. Analog VLSI Implementations

In analog implementations, signals are modelled by physical variables such as a voltage, a current charge, a frequency, or a time duration. This analog representation of neural network parameters has various advantages and drawbacks:

On the one hand, analog NN hardware obtains high processing speeds and high densities of components by exploiting the physical properties of analog signals to perform neural network operations. On the other hand, this representation is characterized by very poor absolute precision, since it is very susceptible to outside influences such as variations in temperature, power supply, components, etc.. Thus, analog chip design is a very difficult task, which is further complicated by problems of weight storage and the need for a multiplier linear over a wide range. Synthesis tools such as CAD do not exist for analog hardware design.

Due to these problems, working analog neural network implementations are still limited. They mostly serve in elementary applications, for example as the front ends of perception systems.

### 6.2. Digital Integrated Circuit Implementations

Digital VLSI is a proven and mature technology which has been used for many years in conventional computers. In contrast to analog NN chips, digital neurocomputers are sold by a large number of manufacturers for a reasonable price. Moreover, digital neural network implementations tend to be able to solve a larger variety of tasks than analog implementations. An algorithm can easily be mapped onto a digital system in a top-down approach. Multi-layer feedforward neural network is the prevalent neural network design implementation. A number of CAD systems is available to support the designer’s work.

In digital VLSI technology, the weights can be simply stored in RAM. The accuracy of digital VLSI hardware is given by the number of bits of the operands and accumulators. However, digital VLSI hardware is subject to serious constraints in chip area. A large number of neurons, high numerical precision, and high speed elements are very area consuming. Thus a compromise has to be found between accuracy, processing speed, and the number of neurons on the chip. Typically digital VLSI chips have a higher precision, but lower speed and density than analog chips.

A digital neural network implementation can either consist of a single VLSI chip or multiple chips can be composed to form a neural network architecture. Multiple chip architectures include slice architectures and Radial Basis Function Networks. SIMD and systolic arrays are built using multi-processor chips. Arrays of SIMD chips are particularly suited for the implementation of multi-layer feedforward neural networks since all processors on one particular chip execute the same instruction in parallel but on different data. Common control and data buses can combine multiple chips. For example, the Adaptive Solutions CNAPS system forms a SIMD array using Inova N6400 chips. Systolic arrays are based on the concept of pipe-lining: After performing a single calculation a processor passes the result on to the next processor. Siemens MA-16 chips can be employed to build systolic arrays.

### 6.3. Hybrid VLSI Implementations

Hybrid implementations try to form a compromise between digital and analog VLSI technologies by taking the best of both worlds. Usually some or all of the internal processing is performed in analog fashion while the communication with the outside environment is digital to facilitate the incorporation into digital systems. For example, the AT&T ANNA Artificial Neural Network ALU operates internally with capacitor charge to store the weights, but has digital inputs and outputs. The Neuroclassifier chip of the University of Twente is a two-layer, fully interconnected network with 70 analog inputs, six hidden-layer neurons and 1 to 6 analog outputs, whose 5-bit digital weights are stored on on-chip SRAM. It has been successfully applied to the classification of high energy physics particles and to real-time image processing.

### 6.4. Training Modes for Neural Network Hardware

We distinguish between different training modes in hardware depending on the location where all or parts of the training phase is realized. As we will point out this issue is closely related to the precision of the weight representation which is required to establish successful learning. A categorization of training modes for multi-layer perceptrons using error backpropagation can be given as follows:

*Off-chip learning:* The whole training process takes place on a separate computer with high precision. The results are quantized and loaded onto the chip. Only recall is performed on-chip. Practical experiments have revealed that low on-chip accuracy suffices to achieve successful learning. For example, the ANNA chip which does not have any on-chip learning capabilities has been successfully used foe high-speed character recognition although it only uses a 6–bit weight resolution and a 3–bit resolution for inputs and outputs [37].

*Chip-on-the-loop learning:* In this approach, the forward propagation part of the training process is realized on-chip while the error backpropagation and the weight updates are performed off-chip on a high-precision computer. The resulting floating point representations of the weights are discretized using a staircase-shaped multiple-threshold function and then the forward propagation pass of the training phase is repeated.

*On-chip learning:* In the event that the complete training process is executed on-chip, we speak of on-chip learning. Consequently, only limited accuracy is available for weight training. Several simulations indicate that weight training with standard backpropagation only leads to successful leaning if the weights have a precision of at least 16 bits [2, 21]. This requirement is due to the fact that the weight quantization step often exceeds the weight updates which prevents the weights from changing. A number of weight discretization algorithms and hardware friendly training algorithms such as weight propagation are capable of alleviating this problem [4, 23, 26, 47].

## 6.5. Performance Evaluation of VLSI Implementations

Ideally, a neural network hardware implementation should incorporate the basic principles in the design of powerful biological neural nets while being adapted to perform biologically inspired as well as other applications. The most important of these principles are:

- *A large number of neurons:* The human brain has about $10^{12}$ neurons.
- *A large number of interconnections:* There are about $1000$ synapses per neuron in the brain.
- *Learning capability:* This requires changeable weights.
- *High processing speed*

The speed of implementations of multi-layer perceptrons is typically rated in connections per second (CPS) and connection updates per second (CUPS). The CPS value measures the rate of multiplications and accumulate operations in the recall phase. The CUPS value provides the rate of weight updates in the training phase. This value usually refers to weight training of multi-layer perceptrons using error backpropagation, but it can be given for other algorithms and other neural networks as well.

We now provide a comparison of several VLSI chips used for multi-layer neural network implementations with respect to the total number of neurons and synapses, learning capability, and accuracy [27].

The precision of node activations and weights is measured in bits. BP abbreviates back-propagation and PE abbreviates processing element.

| **Type** | **Name** | **Learning** | **Precision** | **Neurons** | **Speed** |
|---|---|---|---|---|---|
| *Analog* | Intel ETANN | — | 6 b x 6 b | 64 | 2 GCPS |
| *Digital* | Philips L-Neuro 2.3 | — | 16 b - 32 b | 12 PE | 720 MCPS |
| | MCE MT19003 | — | 13 b | 8 | 32 MCPS |
| | Hitachi WSI | BP | 9 b x 8 b | 576 | 250 MCPS, 64 MCUPS |
| *Hybrid* | AT&T ANNA | — | 3 b x 6 b | 16 - 256 | 2.1 GCPS |
| | Neuroclassifier | — | 6 b x 5 b | 6 | 21 GCPS |
| | Ricoh RN-200 | BP | na | 16 | 3.0 GCPS |

## 7. Stochastic Perceptrons

### 7.1. Stochastic Perceptrons and Probabilistic Concepts

The belief that biological neurons are probabilistic devices has motivated an extension of the perceptron concept. The *stochastic perceptron* is a classifier like the conventional perceptron. The functionality of the stochastic perceptron is similar to the functionality of the perceptron as illustrated in Fig.3 although the activation function $f$ is not necessarily a threshold function but an arbitrary function into the interval $[0, 1]$. Furthermore, the stochastic perceptron does not produce a deterministic decision which associates a pattern $\mathrm{x}$ with class $0$ or with class $1$. Instead, it assigns class membership with probability given by the weighted sum of its inputs. If $y$ denotes an output value, the probability that the stochastic perceptron assigns an input pattern $\mathrm{x}$ to class $1$ is given by

$$P(y = 1|\mathrm{x}) = f\left(\sum_{i=1}^{n} w_i x_i\right) . \tag{33}$$

Note that a threshold parameter $\theta$ does not need to occur in this formulation since it can be incorporated in the definition of the function $f$. In the following discussion we restrict ourselves to monotonically increasing activation functions $f$. Thus, our discussion includes the sigmoidal activation functions used in multi-layer perceptrons. The input patterns only adopt values in the Boolean domain $\mathcal{I}^n = \{+1, -1\}^n$. The input space has an unknown underlying distribution denoted by $D$. The notation $p_D(\mathrm{x})$ is used for the probability of observing vector value $\mathrm{x}$ under the distribution $D$.

The class of stochastic perceptrons can be embedded into the class of *probabilistic concepts* (p-concepts) [25]. A p-concept consists of a function $c : \mathcal{I}^n \to [0, 1]$ and probabilistic device which generates an output of $y = 1$ with probability $c(\mathrm{x})$ for input $\mathrm{x}$.

### 7.2. PAC Learning Criterion

For each classification of an input space with underlying distribution $D$ there exists a p-concept called target p-concept which provides an exact model. A learning algorithm for stochastic perceptrons must be geared at finding a good approximation of the target p-concept in terms of a stochastic perceptron. Note that the adaptive parameters of a stochastic perceptron not only include the weights $w_i$ but also the activation function $f$. Thus, given a set of training patterns a learning algorithm determines a set of $w_i$ and a activation function $f$ yielding a stochastic perceptron approximating the target p-concept. Following general statistical nomenclature, this stochastic perceptron is called the hypothesis and is denoted by $h$.

The success of a learning algorithm can be expressed in terms of a version of the *"Probably Approximately Correct" (PAC) learning criterion* [41]. This formulation depends on a error measure $E$ which is defined as follows:

$$E(h, c) = \sum_{\mathrm{x}} p_D(\mathrm{x}) \left| h(\mathrm{x}) - c(\mathrm{x}) \right| , \tag{34}$$

where $h$ denotes the hypothesis and $c$ denotes the target p-concept. The error measure $E$ is called *variation distance*.

### 7.3. $k$-blocking Distributions

Marchand and Hadjifaradji presented a learning algorithm which PAC learns the class of stochastic perceptrons under a certain class of distributions called k-blocking distributions where "PAC learns" means the following: If the target p-concept is a stochastic perceptron and the underlying distribution is k-blocking, the algorithm then the algorithm will find for any $0 < \varepsilon, \delta < 1$ a hypothesis $h$ such that $E(h, c) < \varepsilon$ with confidence $1 - \delta$.

The algorithm is based on the fact that a weight $w_i$ of the target stochastic perceptron can be detected by changing the variable $x_i$ while assigning a fixed value to a certain set of other variables. This set depends on the index $i$ and is called blocking set. It is required to have the property that "the distribution on all the remaining variables is unaffected by the setting of $x_i$ whenever all the variables of the blocking set are set to a fixed value." Formally, we have:

$$\begin{gathered} p_D(\mathbf{x}_U|\mathbf{x}_B = \mathbf{b},\ x_i = +1) = p_D(\mathbf{x}_U|\mathbf{x}_B = \mathbf{b},\ x_i = -1) \\ \forall\ \mathbf{b},\ \forall\ \mathbf{x}_U\ . \end{gathered} \tag{35}$$

Here $U$ denotes the complement of $B \cup \{x_i\}$ in $\{x_1, \ldots, x_n\}$. The symbols $\mathbf{x}_U$ and $\mathbf{x}_B$ stand for the restriction of $\mathbf{x}$ on $U$ and $B$, respectively. The symbol $\mathbf{b}$ denotes an assignment for $B$. We say that $B$ is a minimal blocking set if there is no subset of $B$ which is a blocking set.

If all the variables are statistcally independent from each other the empty set forms a blocking set for every variable $x_i$. In this case, the influence of $w_i$ on the probability that $y = 1$ can be estimated by fixing $x_i$ at value $+1$ or at value $-1$ [38]. The algorithm of Marchand and Hadjifaradji satisfies the PAC learning criterion for the more general case of $k$-blocking distributions which are defined as follows.

A distribution $D$ is called *$k$-blocking* if $|B_i| \leq k\ \forall\ i = 1, 2, \ldots, n$ whenever $B_i$ is a minimal blocking set for variable $x_i$. A standard calculation shows that all Markov distributions of $k$th order belong to the class of $2k$-blocking distributions. Thus, the $k$-blocking family comprises many distributions found in practice.

### 7.4. Learning Stochastic Perceptrons

As noted before, the weight $w_i$ of a hypothesis stochastic perceptron can be derived by fixing a blocking set of a variable $x_i$ at a certain value. This idea gives rise the definition of the *blocked infuence* of $x_i$:

$$\mathrm{Binf}(x_i|\mathbf{b}_i) = P(y = 1|\mathbf{x}_{B_i} = b_i, x_i = 1) - P(y = 1|\mathbf{x}_{B_i} = b_i, x_i = -1)\ , \tag{36}$$

where $B_i$ is a blocking set for variable $x_i$ and $\mathbf{b}_i$ is an assignment for $\mathbf{x}_{B_i}$. Note that $\mathrm{Binf}(x_i|\mathbf{b}_i)$ not only depends on $x_i$ but also on the choice of the blocking set $B_i$ and the vector $\mathbf{b}_i$. The main importance of $\mathrm{Binf}(x_i|\mathbf{b}_i)$ lies in the fact that, regardless of the choice of $B_i$ and $\mathbf{b}_i$, we have the following relations whenever the target p-concept is a stochastic perceptron.

$$\mathrm{Binf}(x_i|\mathbf{b}_i) \begin{cases} \geq 0 & if\ w_i = +1 \\ = 0 & if\ w_i = 0 \\ \leq 0 & if\ w_i = -1 \end{cases} \tag{37}$$

This relationship gives rise to a simple rule for finding the weights $w_i$ of the target stochastic perceptron provided a blocked influence $\mathrm{Binf}(x_i|\mathbf{b}_i)$ can be determined.

The search for a blocking set may potentially be too expensive, even under the assumption that the distribution $D$ is $k$-blocking. In most real-world applications, we can restrict ourselves to searching for a blocking of size $k$ in a neighborhood of $x_i$.

Once $B_i$ is found and set to an arbitrary value $\mathbf{b}_i$, an empirical estimate of $\mathrm{Binf}(x_i|\mathbf{b}_i)$, denoted by $\widehat{\mathrm{Binf}}(x_i|\mathbf{b}_i)$, can be calculated based on the training set. Hoeffding's inequality yields a number of training patterns which suffices to guarantee a good estimate [20]. If $\mathrm{Binf}(x_i|\mathbf{b}_i)$ is very small, this number is prohibitively large. A lemma shows that the variables $x_i$ whose blocked influence $\mathrm{Binf}(x_i|\mathbf{b}_i)$ is very small for all $\mathbf{b}_i$ can be ignored. The corresponding weights $w_i$ can be set to zero without losing much accuracy in the approximation of the target stochastic perceptron $c$. For all other $x_i$, the weight $w_i$ is set to $+1$ if $\max_{\mathbf{b}_i}\{\mathrm{Binf}(x_i|\mathbf{b}_i)\}$ is positive and the weight $w_i$ is set to $-1$ if $\max_{\mathbf{b}_i}\{\mathrm{Binf}(x_i|\mathbf{b}_i)\}$ is negative. If $s$ denotes the weighted sum of the components of a vector $\mathbf{x}$ we know with large confidence that the following equation holds for the target $c$:

$$c(\mathbf{x}) = P\left(y = 1 | \sum_{i=1}^{n} w_i x_i = s\right) . \tag{38}$$

The number of training patterns needs to be large enough for yet another application of Hoeffding's inequality which ensures with large confidence that, disregarding insignificant vector values $\mathbf{x}$, a good approximation of $c(\mathbf{x})$ is obtained by

$$h(\mathbf{x}) = \widehat{P}\left(y = 1 | \sum_{i=1}^{n} w_i x_i = s\right) . \tag{39}$$

## 8. REFERENCES

1. M.A. Arbib. *Brains, Machines, and Mathematics*. Springer-Verlag, New York, NY, 1987.
2. K. Asanovic and N. Morgan. Experimental evaluation of precision requirements for back-propagation training of artificial neural networks. In U. Rueckert and J.A. Nossek, editors, *Proceedings of the 2nd International Conference on Microelectronics for Neural Networks, Evolutionary and Fuzzy Systems*, pages 9–15, Muenchen, Germany, 1991.
3. R. Battiti. Accelerated backpropagation learning: Two optimization methods. *Complex Systems*, 3:331–342, 1989.
4. R. Battiti and G. Tecchiolli. A digital processor for neural networks and reactive tabu search. In *Proceedings of the 4th International Conference on Microelectronics for Neural Networks, Evolutionary and Fuzzy Systems*, pages 17–25, Turin, Italy, 1994.
5. E.B. Baum. What size of neural net gives valid generalization? *Neural Computation*, 1(4):151–160, 1989.
6. C.M. Bishop. Curvature-driven smoothing: A learning algorithm for feedforward neural networks. *IEEE Transactions on Neural Networks*, 4(5):882–884, 1993.
7. C.M. Bishop. *Neural Networks for Pattern Recognition.* Clarendon Press, Oxford, UK, 1995.
8. E.K. Blum and L.K. Li. Approximation theory and feedforward networks. *Neural Networks*, 4(4):511–515, 1991.
9. Y. Le Cun, B. Boser, J.S. Denker, D. Henderson, R.E. Howard, W. Hubbard, and L.D. Jackel. Optimal brain damage. In D.S. Touretzky, editor, *Advances in Neural Information Processing Systems*, volume 2, pages 598–605, San Mateo, CA, 1990. Morgan Kaufmann.
10. G.E. Hinton D.E. Rummelhart and R.J. Williams. Learning internal representations by error propagation. *Parallel Distributed Computing: Explorations in the Microstructure of Cognition*, 1: Foundations:318–362, 1986.
11. H. Drucker and Y. Len Cun. Improving generalization performance using double backprop-agation. *IEEE Transactions on Neural Networks*, 3(6):991–997, 1992.
12. S.E. Fahlmann and C. Lebiere. The cascade-correlation learning architecture. In D.S. Touretzky, editor, *Advances in Neural Information Processing Systems*, volume 2, pages 524–532, San Mateo, CA, 1990. Morgan Kaufmann.
13. R. Fletcher. *Practical Methods of Optimization.* John Wiley, New York, NY, 1987.
14. R. Fletcher. Computational solution of non-linear systems of equations: Low storage methods for unconstrained optimization. In E.L. Allgower et al., editor, *Lectures in Applied Mathematics*, volume 26, pages 165–179. American Mathematical Society, 1990.
15. G.J. Gibson and C.F.N. Cowan. On the decision regions of multilayer perceptrons. In *Proceedings of the IEEE,* volume 78, pages 1590–1594, 1990.
16. B. Hassibi and D.G. Stork. Optimal brain surgeon. In S.J. Hanson, J.D. Cowan, and C.L. Giles, editors, *Advances in Neural Information Processing Systems*, volume 5, pages 164–171, San Mateo, CA, 1993. Morgan Kaufmann.
17. D. O. Hebb. *The Organization of Behavior*. John Wiley, New York, 1949.

18. J. Hertz, A. Krogh, and R.G. Palmer. *Introduction to the Theory of Neural Computation*. Addison-Wesley, Redwood City, CA, 1991.

19. M.R. Hestenes and E. Stiefel. Methods of conjugate gradients for solving linear problems. *Journal of Research of the National Bureau of Standards*, 49(6):409–436, 1952.

20. W. Hoeffding. Probability inequalities for sums of bounded random variables. *Journal of the American Statistical Association*, 58(301):13–30, 1963.

21. J.L. Holt and J-N Hwang. Finite precision error analysis of neural network hardware implementations. *IEEE Transactions on Computers*, 42:1380–1389, 1993.

22. W.Y. Huang and R.P. Lippmann. Neural net and traditional classifiers. In D.Z. Anderson, editor, *Neural Information Processing Systems*, pages 387–396. American Institute of Physics, New York, NY, 1988.

23. Jabri. Practical performance and credit assignment efficiency of multi-layer perceptron perturbation based training algorithms. Sedal technical report 1–7–94, Sydney University Electrical Engineering, Sydney, Australia, 1994.

24. D. Jeong and S. Lee. Merging backpropagation and hebbian learning rules for robust classification. *Neural Networks*, 9:1213–1222, 1996.

25. M.J. Kearns and R.E. Schapire. Efficient distribution-free learning of probabilistic concepts. *Journal of Computer and Systems Sciences*, 48:464–497, 1994.

26. P.H.W. Leong and M.A. Jabri. A low-power vlsi arrrhythmia classifier. *IEEE Transactions on Neural Networks*, 6:1435–1445, 1995.

27. C.S. Lindsey and T. Lindblad. Review of hardware neural networks: A user's perspective. Plenary talk given at the Third Workshop on Neural Networks: From Biology to High Energy Physics, 1994.

28. D.G. Luenberger. *Linear and Nonlinear Programming*. Addison-Wesley, Reading, MA, 1984.

29. W.S. McCulloch and W. Pitts. A logical calculus of ideas immanent in nervous activity. *Bulletin of Mathematical Biophysics*, 5:115–133, 1943.

30. M.L. Minsky and S.A. Papert. *Perceptrons*. MIT Press, Cambridge, MA, 1969.

31. M.C. Mozer and P. Smolensky. Skeletonization: A technique for trimming the fat from a network via relvance assessment. In D.S. Touretzky, editor, *Advances in Neural Information Processing Systems*, volume 1, pages 107–115, San Mateo, CA, 1989. Morgan Kaufmann.

32. J. Nocedal. Udating quasi-newton matrices with limited storage. *Mathematics of Computation*, 35:773–782, 1980.

33. D. Plaut, S. Nowlan, and G.E. Hinton. Experiments on learning by back propagation. Technical Report TR CMU-CS-86–126, Department of Computer Science, Carnegie Mellon University, Pittsburgh, PA, 1986.

34. W.H. Press, S.A. Teukolsky, W.T. Vetterling, and B.P. Flannery. *Numerical Recipes in C: The Art of Scientific Computing*. Cambridge University Press, 1992.

35. F. Rosenblatt. The perceptron: a probablistic model for information storage and retrieval in the brain. *Psych. Rev*, (65):386–408, 1958.

36. F. Rosenblatt. *Principles of Neurodynamics: Perceptrons and the Theory of Brain Mechanisms*. Spartan, Washington, DC, 1962.

37. E. Saeckinger, B.E. Boser, J. Bromley, Y.Le Cun, and L.D. Jackel. Application of the anna neural network chip to high-speed character recognition. *IEEE Transactions on Neural Networks*, 3:498–505, 1992.
38. R.E. Schapire. *The Design and Analysis of Efficient Learning Algorithms*. MIT Press, Cambridge, MA, 1992.
39. F.M. Silva and L.B. Almeida. *Acceleration Techniques for the Backpropagation Algorithm*, pages 110–119. Neural Networks. Springer, Berlin, 1990.
40. F.M. Silva and L.B. Almeida. *Speeding up Backpropagation*, pages 151–160. Advanced Neural Computers. Elsevier, Amsterdam, 1990.
41. LG. Valiant. A theory of the learnable. In *Communications of the ACM*, volume 27, pages 1134–1142. 1984.
42. V.N. Vapnik and A.Y. Chervonenkis. On the uniform convergence of relative frequencies of events to their probabilities. *Theory of Probability*, 16(2):264–280, 1971.
43. T.P. Vogl, J.K. Mangis, A.K. Rigler, W.T. Zink, and D.L. Alkon. Accelerating the convergence of the backpropagation method. *Biological Cybernetics*, 59, 257–263.
44. A.S. Weigend, B.A. Hubermann, and D.E. Rummelhart. Predicting the future: A connectionist approach. *International Journal of Neural Systems*, 1(3):193–209, 1990.
45. A. Wieland and R. Leighton. Geometric analysis of neural network capabilities. In *Proceedings of the First IEEE Conference on Neural Networks*, volume 3, pages 385–392, San Diego, CA, 1987.
46. P.M. Williams. Bayesian regularization and pruning using a laplace prior. Cspr-312, University of Sussex, School of Cognitive and Computing Sciences, Brighton, UK, 1994.
47. Y. Xie and M.A. Jabri. Training limited precision feedforward neural networks. In *Proc. 3rd Australian Conference on Neural Networks*, pages 68–71, 1992.