Multilayer neural networks

Jacobs, Jürgen

Publication date:
2004

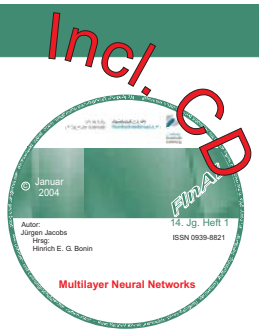
Document Version
Publisher's PDF, also known as Version of record

University of Applied Sciences

Fachhochschule Nordostniedersachsen

Lüneburg
Buxtehude
Suderburg

Arbeitsberichte aus der Informatik



Multilayer Neural Networks

Jürgen Jacobs

14. Jahrgang, Heft 1, Januar 2004, ISSN 0939-8821
Technical Reports and Working Papers
Hrsg: Hinrich E. G. Bonin
Volgershall 1, D-21339 Lüneburg
Phone: xx49.4131.677175 Fax: xx49.4131.677140

Preface

Mechanization of the human thought process is one of the key challenges of computer science. The neural network problem solving approach tries to meet with this challenge by simulating the biological mechanisms of the brain directly. To be more precise: The study of artificial neural networks *borrows* concepts and ideas from neurological research; it is *inspired* by our understanding of the brain. However, not all artificial neural networks models are developed to represent the electrochemical processes of their biological counterpart as accurate as possible. Neural networks have been applied in many other areas besides neurological research. They are used successfully in science, engineering and business – mainly for solving pattern recognition tasks. In those areas, the problem solving power of the network model used is much more important than its adherence to biological mechanisms.

The aim of these lecture notes is to give a quick introduction into neural networks, its algorithms and applications. The notes are not intended as a replacement of a comprehensive textbook. The following subjects will be covered:

- Problem situations where neural networks technology can successfully be applied.
- Similarities and differences between artificial and biological neural networks.
- Strengths and weaknesses of the neural network problem solving approach.
- Basic problem solving algorithms of feed-forward networks.
- Data preparation for neural networks.

1. The Neural-Network Paradigm

Since their appearance, digital computers have been admired because of their outstanding ability to do complex calculations precisely and rapidly. This ability also reflects the origin of the name *computer*. Let's think for example of the problem to calculate the trajectory of a rocket. Most computers do not have any difficulties to solve this problem, if they are provided with the right program (formulas). On the other hand it would be more than many human beings could cope with, even if they were given all formulas necessary. Does this make the computer more intelligent than us? As far as the ability of **algorithmic inference** – the ability to make use of formulas - is concerned, the answer might be yes. But computers are far from being able to create the necessary formulas and transform them into a program by themselves. This is still the domain of intelligent human beings. So the precondition for applying algorithmic inference is a precise quantitative understanding of system behaviour.

In some cases an algorithmic solution may be too complex (think of a motor diagnosis) or inadequate (think of finding out the right objective of an advertisement campaign). In the 1980s expert systems became popular. Expert systems try to copy the problem solving approach of human experts. Not formulas but rules are used to describe the system behaviour. For example, the two rules below could be part of a marketing expert system:

IF product is to be launched
AND product is an innovation
THEN marketing objective is to stimulate primary demand

IF marketing objective is to stimulate primary demand
AND there is no motivation to buy the product
THEN advertisement objective is to provide product information

These rules allow to deduce the advertisement objective when the three conditions 'product is to be launched', 'product is an innovation', 'there is no motivation to buy the product' are met. Expert systems make use of **deductive inference** and can cope with problems where an algorithmic approach is not possible, not known or too complex. The precondition for applying deductive inference is the ability to make knowledge explicit in the form of rules.

In many situations of everyday life, neither algorithmic nor deductive inference would be helpful. The admirable achievements of our brain cannot be restricted to our capability to use algorithms or to symbolic reasoning with complex rules. For example, how do we remember a face of a person, even if we have not seen that person for a long time? Why are we able to understand a person speaking English with an accent we have never heard before? How did we learn to write? Nobody has explained us to write the letter A with the help of rules or formulas. If so, this would have been a very complex task: Think of the many fonts which can be used. For example

A, A, $\mathcal{A}$, $\mathcal{A}$, **A**

represent the same letter. And then think of hand written characters which would make the rules and/or formulas even more complex - too complex of course to explain them to little children going to school. This example demonstrates that a new problem solving approach is required. As children learn to read and write by practising, by learning from examples, the new problem-solving paradigm should be based on **inductive inference**. Induction means generalization of examples, of special cases. By having learnt to recognize many different shapes of the letter A, it will be possible to recognise previously unseen examples of the letter

A, because they are of a ‘similar’ shape. There is no explicit knowledge describing the concept of similarity. The knowledge is in the examples; it has to be learned from the examples. Inductive inference is the problem solving paradigm most neural networks are based on. Fig. 1.1 illustrates the positioning of neural networks.

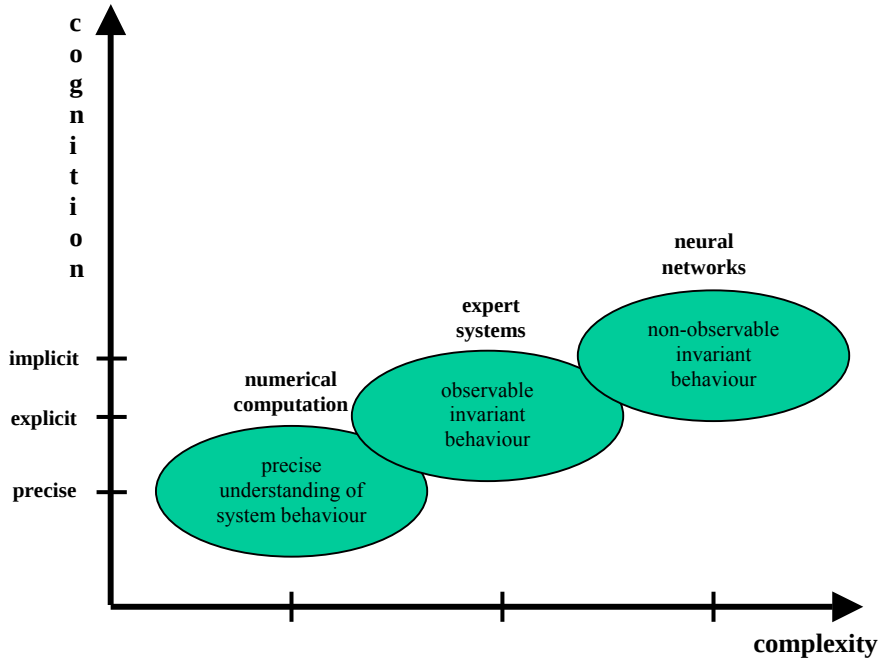


Fig. 1.1: Positioning of neural networks[1]

Neural network research has been started in the late 1940s already, whereas industrial usage began in the 1990s. Since then a huge amount of successful applications in science, technology and business – mainly in the area of pattern recognition – have been arisen. The three examples below shall give a first impression of possible applications in the business area:

- Profit/loss prediction on the stock exchange (input: times series of stock prices; output: recommendation to buy or sell)
- Credit scoring (input: balance sheets; output: insolvency risk)
- Market segmentation (input: social and economical indicators like education and income; output: market segments)

The positive impact of the inductive problem solving approach is that no prior knowledge of system behaviour is required. On the other hand neural networks will act as a black boxes. They can’t explain their results, because generalization is not made explicit.

[1] adopted from Refenes, A.-P.: Introduction, in Refenes, A.-P. (ed.): Neural networks in the capital markets, Chichester, 1995, p.4

2. The Biological Inspiration

In order to better understand the components of artificial neural networks, the list below summarizes basic - sometimes simplified - facts of neuro-biology. Additionally, the basic features of biological neurons are illustrated in fig 2.1:

- The human brain is estimated to contain about 10^{11} neurons.
- Each neuron is connected with 1000 to 10000 other neurons.
- Each cell body (soma) has attached one axon. At the end, the axon forks into several branches. The axon is electrically active; it transports voltage pulses of about 10^{-1} Volt.
- At the end of the branches of an axon there are contacts – called synapses. A synapse couples the axon with the input channels – called dendrites – of other neurons. Neural systems have both excitatory and inhibitory connections.
- When the voltage pulses arrive at the soma, and their combined effect exceeds a certain critical threshold, the neuron will be activated. This means that the neuron will transmit a new voltage pulse along its axon.

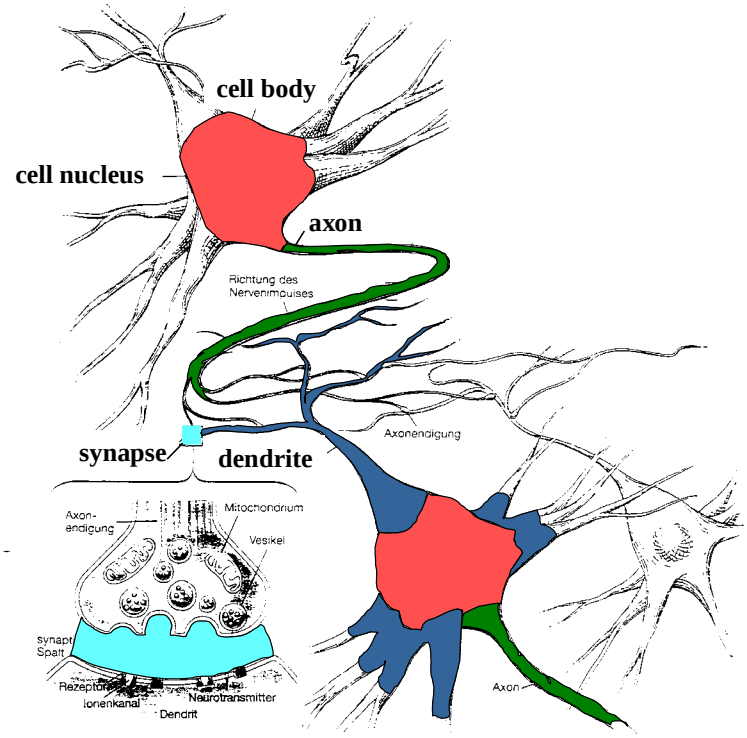


Fig. 2.1: The basic features of biological neurons[2]

Biological neurons are rather slow devices. The time to produce a voltage pulse is in the range of milliseconds, and voltage pulses are transferred with a speed of about 100 m/sec. Therefore standard personal computers with a frequency of 1 GHz are about 1 million times faster and

[2] adopted from Spektrum der Wissenschaft, Spezial 1, Gehirn und Geist, Spektrum Akademischer Verlag, Heidelberg, p. 24

signals are transmitted about 3 million times faster. The slowness as well as the relatively simple structure of individual neurons seem to be incompatible with the ability of our brain to accomplish complex tasks. A possible solution to this contradiction (at least partly) is the observation that our brain makes use of massively parallel processing. Parallel distributed processing of many neurons and not sequential processing steps of only a few neurons characterize thought processes. This aspect of neural network research is expressed more carefully (not suggesting to model all biological mechanisms exactly) by the terms **parallel distributed processing** or **connectionism**. As the contribution of a few individual neurons to the solution will not be too influential, neural network systems are regarded as **fault tolerant** systems. Damage to small parts of the system is not expected to disrupt its performance completely. Moreover, due to their inductive problem solving approach (generalization of examples), neural networks are also tolerant to corrupted versions of the original input patterns.

The above-mentioned biological facts lead us to a simplified model of a biological neuron which is depicted in fig. 2.2:

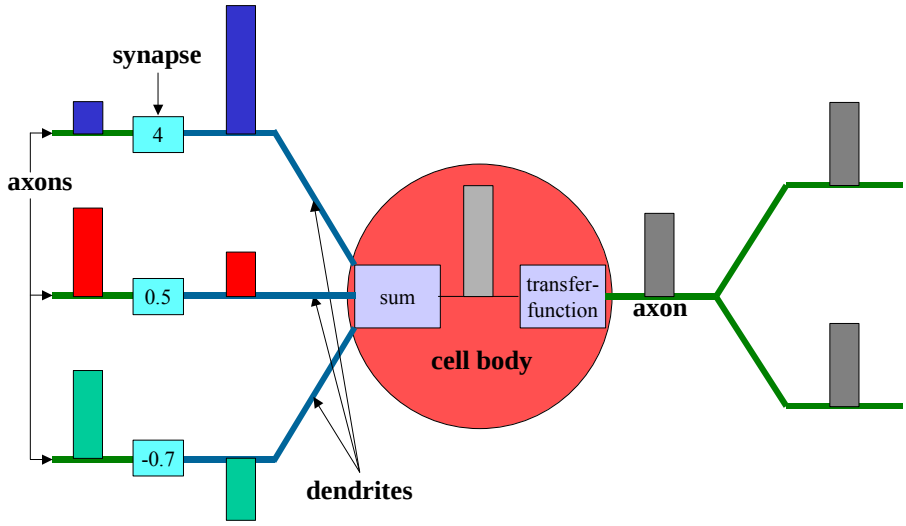
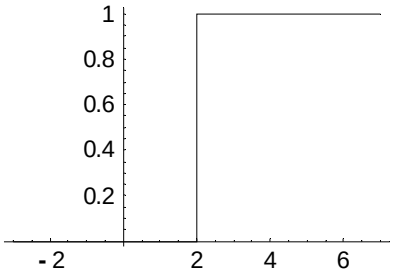
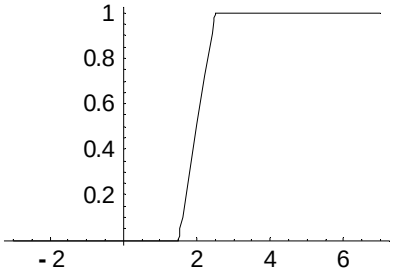


Fig. 2.2: A simplified model of a biological neuron

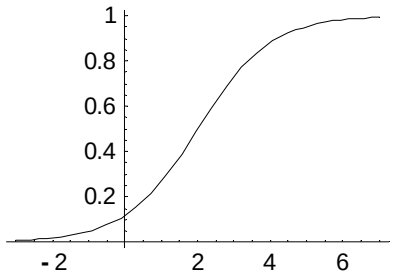
- The interconnection of two neurons via several dendrites is reduced to only one connection.
- The coupling strength of a synapse is modelled by a **weight** factor. Inhibiting synapses are modelled by negative weights, while positive weights model synapses which excite the dendrite they are connected to.
- All incoming signals are combined by a so-called **propagation function** - usually the sum of the inputs.
- The strength of the output signal is calculated by a **transfer function**. The transfer function is sometimes also called **activation function**. Sometimes, however, the activation function only determines how strongly the neuron is activated, and a separate **output function** determines the strength of the output signal due to the activation. Fig. 2.3 shows three typical (non-continuous, continuous, and differentiable) transfer functions representing a neuron with a threshold value 2.



stair-step threshold function:
f(x) = 0, x < 2
= 1, x ¥ 2



thresholded linear function:
f(x) = 0, x § 1.5
= x – 1.5, 1.5 < x < 2.5
= 1, x ¥ 2.5



logistic (sigmoid) function:
$f(x) = 1/(1 + e^{-(x-2)})$

Fig. 2.3: Typical transfer functions

Normally the task of the network designer is to choose

- an appropriate topology of the network,
- suitable propagation and transfer functions,
- suitable training examples, as well as
- a suitable learning procedure.

The learning procedure will be used to fix the remaining free parameters of the network with the help of the training examples:

- the weights for each interconnection between two neurons, and
- the threshold of each neuron.

Both, **supervised** and **unsupervised learning** algorithms have been developed. Supervised learning algorithms require training examples with known outputs. The errors between actual and desired outputs are used to adjust the weights. Unsupervised learning algorithms make use of regularities of input data. Observed output variable values are not required. In the following we will discuss supervised learning only.

As already mentioned, most artificial neural networks differ from their biological counterparts. Usually artificial neural networks

- contain only a small number of neurons (typically 100 to 1000),
- do not model time dependency (e.g. the frequency of voltage pulses), and
- use learning procedures which are mathematically and not biologically motivated.

3. Feed-forward Neural Networks

Feed-forward networks can be arranged in layers so that connections only go from one layer – starting from input neurons - to a later layer until the output neurons are reached. Fig. 3.1 shows a simple feed-forward network with an input layer consisting of two neurons connected to an output layer consisting of one neuron. We call this a **single-layer network** (although it actually has two layers of *nodes*), because there is only one layer of trainable *weights* and one layer of neurons (in our example this layer consists of one output neuron) with adaptable threshold values. The input neurons will propagate the input signals unchanged. Fig. 3.1 shows a trained network, where both weights have value 1 and the output neuron has a threshold value 0.5.

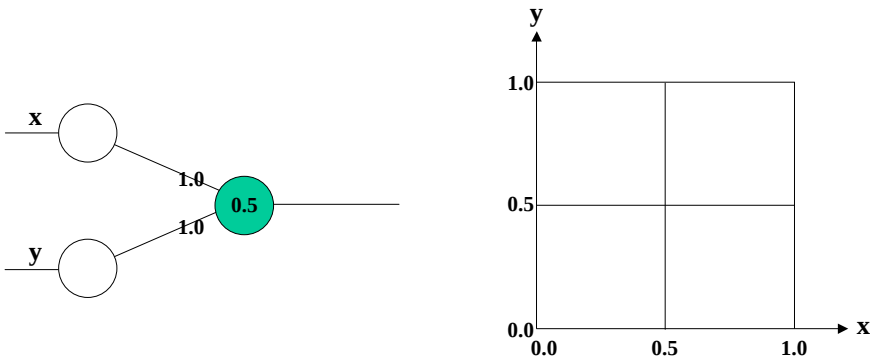


Fig. 3.1: Perceptron with two input units and one output unit

Single-layer networks with stair-step threshold transfer functions are also called **perceptron**s, because they were applied to problems of visual perception, in which the inputs were binary images of characters or simple shapes. However, perceptrons are not restricted to problems of visual perception. In our example, input x could be a measure of the annual income of a borrower (high values representing a low amount), input y could be a measure of the credit amount (high values representing a high amount), and the output could be a credit risk indicator. The output neuron will "fire" (i.e. produce the output value 1) and indicate a high credit risk, if the weighted sum of the inputs is greater than or equal 0.5, otherwise it will produce the output 0 and indicate a low credit risk.

Exercise: Give a mathematical description of the input-output relationship of the network depicted in fig. 3.1 and draw the region (decision region) within the two-dimensional input space representing all input values which will lead to the output value 1.

Perceptrons have limited capabilities. They allow only for straight lines as decision boundaries. Multi-layer perceptrons can overcome this limitation. Fig. 3.2 shows a two-layer perceptron.

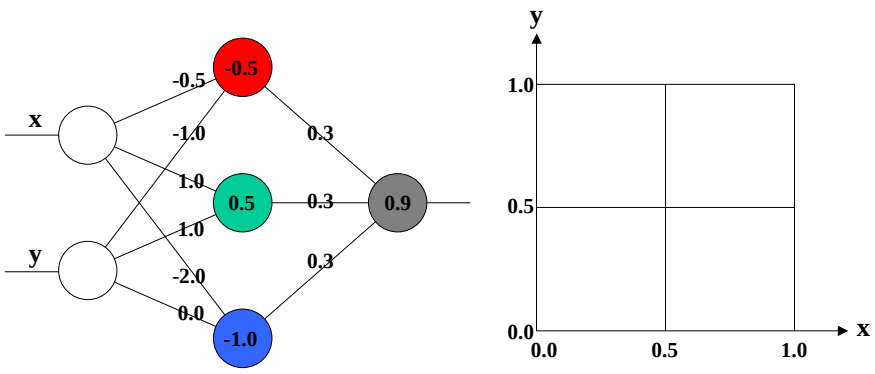


Fig. 3.2: Two-layer perceptron with two input units, three hidden units, and one output unit

The layer in between the input layer and the output layer is called hidden layer. Layered networks have one ore more successive hidden layers, and there are only connections from every unit in one layer to every unit in the *next* layer. The term multi-layer perceptron is also used for multi-layered feed-forward networks with transfer-functions other than the stair-step function. General feed-forward networks allow for connections from one layer to any later layer, not necessarily the next layer.

Exercise: Draw the decision region within the two-dimensional input space representing all input values which will lead to the output value 1. Use fig. 3.3 to decompose the task into several steps.

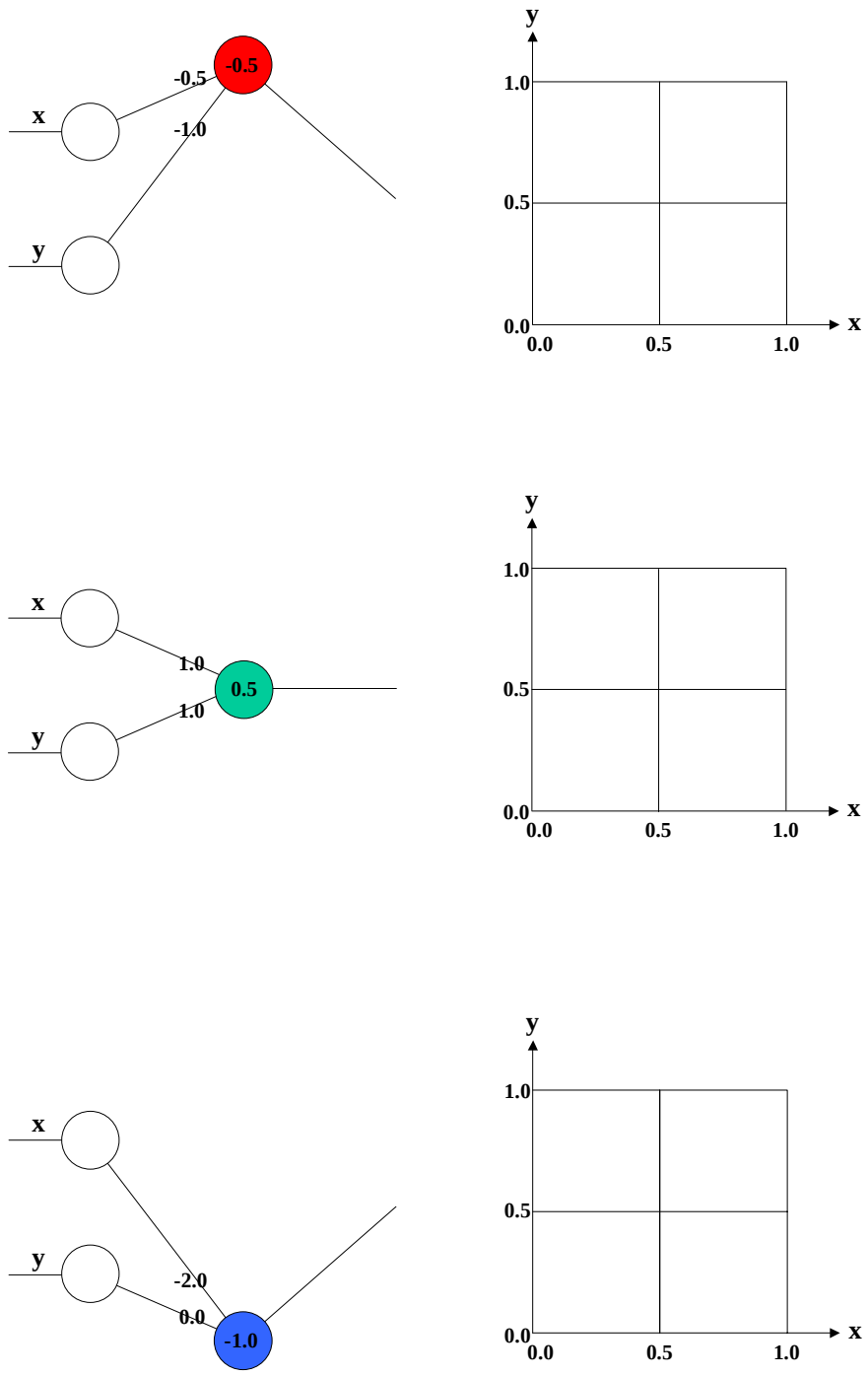


Fig. 3.3: Components of the network depicted in fig. 3.2

In general, networks with two layers of trainable weights generate decision regions which are convex (i.e. a straight line connecting any two points within the region does not cross the boundary of the region). Three-layer networks can generate more complex decision regions which may be non-convex and disconnected, as illustrated in fig. 3.4 (the blue region consists of two parts, and its bigger part is non-convex).

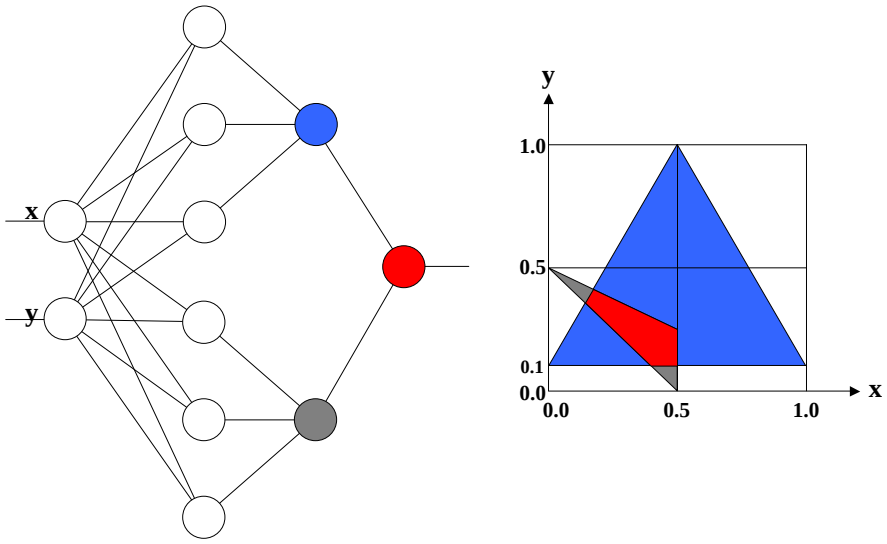


Fig. 3.4: Output regions of a three-layer preceptron

Exercise: Put in appropriate values for the weights and thresholds of the network shown in fig. 3.4 which will generate the red decision region. What will have to be changed to obtain the blue decision region?

Several researchers have investigated the question which input-output relationships can be represented by neural networks, and several variants of **universal approximation** theorems could be proven[3]. These theorems state that an arbitrary function of a certain type (e.g. a continuous function) can be approximated with arbitrary accuracy. For example, it can be shown that any continuous function can be approximated (uniformly on compacta) to arbitrary accuracy by a network with one hidden layer of sigmoid neurons and an output layer of (unthresholded) linear neurons[4]. However, this interesting result (as well as all other results concerning the universal approximation capability of neural networks) is of limited practical value, because it does not tell us the number of neurons required[5]. Moreover, the result does not say anything about the efficiency of the approximator. Perhaps a network with more than one layer of hidden neurons could achieve the same approximation result for a given problem more efficiently, i.e. with less neurons altogether.

[3] Tikk, D., Kóczy, L. T., Gedeon, T. G.: A survey on universal approximation and its limits in soft computing techniques, Research Working Paper RWP-IT-012001, School of Information Technology, Murdoch University, Perth, W.A., 2001, p. 20, see http://www.mft.hu/publications/tikk/Univ_appr.pdf

[4] Bishop, C. M: Neural Networks for Pattern Recognition, New York, 1995, pp. 130 – 132

[5] There are also some results on the number of neurons required to achieve a specified accuracy of approximation, but these results do not advise us to choose the 'right' number of neurons in any practical problem.

4. Backpropagation Learning Algorithm

The (biologically motivated) **Hebbian learning rule** says that the connection strength between two neurons will be increased, when the pre- and post-synaptic neurons are activated simultaneously. This rule can be expressed by the following formula:

$\Delta w_{ij} = \eta \cdot o_i \cdot o_j$

where η is a positive constant called the **learning rate**, o_i (o_j) is the output generated by neuron i (j), and Δw_{ij} denotes the increase of the weight w_{ij} connecting neuron i and j (cf. fig. 4.1).

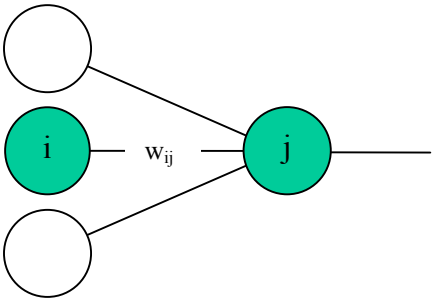


Fig. 4.1: Pre-synaptic neuron i and post-synaptic neuron j are activated simultaneously

One problem associated with this learning rule is that it doesn't take into account, how much the generated output deviates from the target output. The formula below removes this problem:

$\Delta w_{ij} = \eta \cdot o_i \cdot (t_j - o_j)$

where t_j denotes the target output of neuron j. This learning rule is called **delta rule** or **Widrow-Hoff rule**. It is mathematically motivated: the smaller the training error (i.e. the difference $t_j - o_j$) is, the smaller the weight changes will be. The delta rule can only be applied, if the target outputs of all training patterns are known. However, target outputs can only be known (if it all) for output neurons. Therefore this rule is not applicable to multi-layer networks like the one illustrated in fig. 4.2.

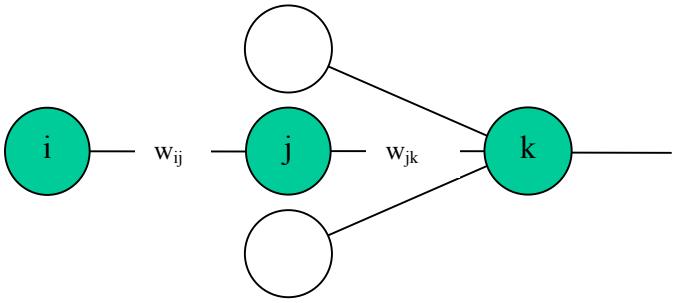


Fig. 4.2: Weight changes of weight w_{ij} can't be calculated with the delta rule

The (mathematically motivated) **backpropagation algorithm** (also known as the **generalized delta rule**) removes this problem. It is based on the iterative **gradient descent technique** which can be used to find local minima of differentiable functions of several variables. The error observed at each output neuron can be regarded as a function of several variables – the weights of all neurons which are connected to the output neuron. It can be easily shown that the threshold of an arbitrary transfer function of neuron n can be represented by an additional input neuron (the so called **bias unit**) which is connected to neuron n and has a permanent output value of 1. Therefore the task to find the right threshold of neuron n can be treated as the task to find the weight of the connected bias unit.

The error $E_k = t_k - o_k$ (t_k denotes the target value, o_k denotes the calculated output value of neuron k) in fig. 4.2 can be expressed as a function of weight-variables easily:

(1) $o_k = f_k(\Sigma_j w_{jk} \cdot o_j)$

where f_k denotes the transfer function of neuron k. The summation index j denotes all neurons (including the bias unit) which are connected to neuron k. The output o_j of the hidden neuron j is given by

(2) $o_j = f_j(\Sigma_j w_{ij} \cdot o_i)$

where f_j denotes the transfer function of neuron j. The summation index i denotes all neurons (including the bias unit) which are connected to neuron j. Combining (1) and (2), we obtain the formula

$E_k = t_k - o_k = t_k - f_k(\Sigma_j w_{jk} \cdot f_j(\Sigma_j w_{ij} \cdot o_i))$.

If index i denotes hidden neurons, the outputs o_i can be expressed as a function of weight-variables with a formula analogous to (2). Otherwise o_i denotes the strength of the input signal.

The overall error E_p for one training pattern p has to take into account all output neurons k. Usually it is calculated by summing up the *squared* errors E_k^2, because these errors can't become negative:

$E_p = \frac{1}{2} \Sigma_k E_k^2$.

The factor ½ is introduced, because it will be eliminated by differentiating the squared errors. The total error E for all training patterns is calculated by

$E = \Sigma_p E_p^2$.

If the network consists of one input and one output neuron only, E is a function of two weight-variables and can be represented graphically (cf. fig. 4.3 and 4.4). In this example there are two training patterns: (1, 0) and (0, 1), i.e. input x = 1 shall result in output $o_2 = 0$, and input x = 0 shall result in output $o_2 = 1$. The transfer function used is the sigmoid function $f_2(x) = 1/(1 + e^{-x})$.

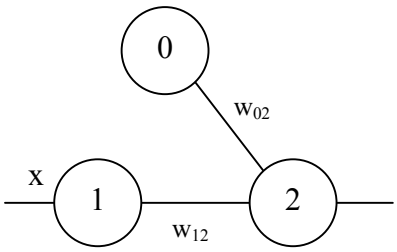


Fig. 4.3: Input neuron 1 and bias unit 0 are connected to output neuron 2.

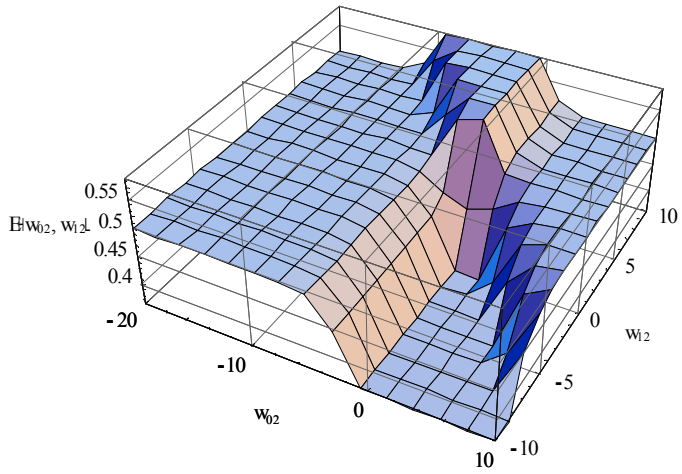


Fig. 4.4: Two-dimensional error surface $E(w_{02}, w_{12})$

Starting with a randomly chosen point on the error surface (i.e. with randomly chosen weights), the negative gradient vector

$-(\partial/\partial w_{02}\, E(w_{02}, w_{12}), \partial/\partial w_{12}\, E(w_{02}, w_{12}))$

shows into the direction of the steepest descent. In fig. 4.5 the gradient vector field of E is visualized by drawing errors representing the vectors.

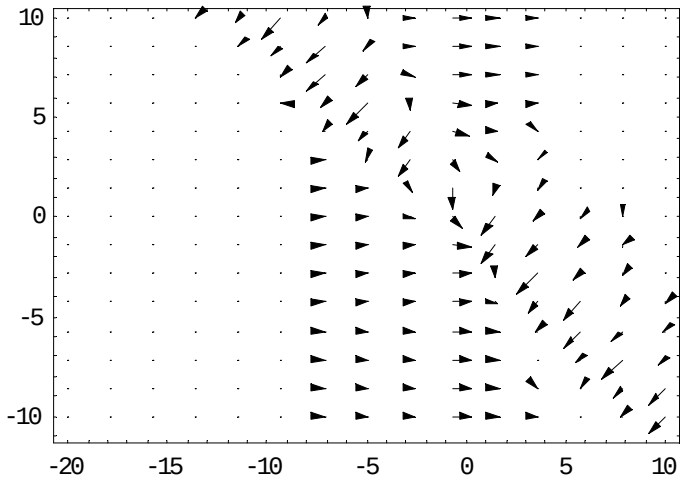


Fig. 4.5: Gradient vector field corresponding to fig. 4.4 (w_{02} drawn on the horizontal axis).

Each arrow points into the direction of the steepest descent at its base point, and its length is proportional to the gradient.

Exercise: Calculate the negative gradient:
$-(\partial/\partial w_{02}\, E(w_{02}, w_{12}), \partial/\partial w_{12}\, E(w_{02}, w_{12})) =$

The backpropagation algorithm implements the gradient descent technique. It tries to find the minimum of the error function by following the path of the steepest descent on the error surface. The vector of the weight changes after each **training epoch** (one training epoch incorporates the presentation of the complete set of training patterns to the network) is proportional to the negative gradient of the overall error function E. More precisely, this is called the **offline** or **batch version** of the backpropagation algorithm. The **online version** – also called **vanilla backpropagation** – calculates the weight changes after the presentation of each individual training pattern p (i.e. the vector of the weight changes is proportional to the negative gradient of the error function E_p). If the patterns are chosen randomly from the training set, it is called the **stochastic version**. The online as well as the stochastic version are only approximations to true gradient descent. Nevertheless they frequently used, because there are some advantages associated with weight updates after each pattern presentation (see below). In this case, the learning rule is given by

$\Delta w_{ij} = \eta \cdot o_i \cdot \delta_j$

where

$\delta_j = f_j'(net_j) \cdot (t_j - o_j)$, if neuron j is an output neuron (f' denotes the first derivative of f),

$\delta_j = f_j'(net_j) \cdot \Sigma_k\, w_{jk} \cdot \delta_k$, if neuron j is a hidden neuron, and

$net_j = \Sigma_i\, w_{ij} \cdot o_i$ denotes the net-input to neuron j.

A precondition for the application of the gradient descent technique is (of course) that all transfer functions f_j are differentiable. In order to calculate the error component δ_j, one has to start with the output layer and then to go backwards until the first layer of the network is reached. The error component δ_j is *back propagated* from the output of the network to its input.

Online and stochastic backpropagation are more efficient in dealing with redundant data than the batch approach. If the training set contains many duplicate or near duplicate patterns, the average over a small portion of the patterns will provide a good approximation to the complete set. Therefore vanilla and stochastic backpropagation will descend approximately the same way down the error surface after having processed this small portion of the patterns as the batch version will descend after having processed the complete pattern set.

Another potential advantage of the online and stochastic version is the belief that they are superior in avoiding **local minima**. One problem of the backpropagation learning algorithm is that it does not guarantee to find the best set of weights. It's true that except in pathological cases, the path of the steepest descent always leads to at least a local minimum on the surface, but the minimum will not necessarily be a global one. The online as well as the stochastic algorithm alleviate this problem, because sequential weight-updates descend different error

surfaces (one for each training pattern) with different local minima. Therefore it will be less likely to end up in any one of them.

Another means to avoid local minima is to add a **momentum term** to the gradient descent formula:

$$\Delta w_{ij}(s+1) = \eta \cdot o_i \cdot \delta_j + \alpha \cdot \Delta w_{ij}(s)$$

where s denotes the number of the update step, and α is a positive constant specifying the strength of the momentum. The momentum term adds inertia to the motion on the error surface. It tends to keep the motion on the error surface in the same direction from one update step to the next, and therefore can help to jump over small local minima in the error surface. Moreover, it will speed up convergence towards the minimum by gradually increasing the weight-changes in regions, where the direction of the downward movement is unchanging (successive $\Delta w_{ij}(s)$ have the same sign). This is especially important in flat regions where the gradient is close to zero.

Iterative procedures like the backpropagation algorithm need a starting point as well as a stopping rule. Usually the starting point consists of randomly chosen weights. However, the weights should not become too large (see next chapter). The starting point may have considerable impact on the number of training steps required (think of starting in a flat region far away from the minimum vs. starting in a steep region close to the minimum). Therefore it is common practice to train a network several times using different starting points.

Several **stop-training rule**s have been proposed. Possible choices are:

- Stop after a given number of training steps.
- Stop when the error drops below a given level.
- Stop when the relative change in the error drops below a given value over a given number of steps.

Of course, the error-based stopping rules seem more satisfying than the first rule. However, the problem with the second rule is that the given error value may never be reached (e.g. if the number of hidden units is too small, so that the universal approximation capability is not given). The problem with the third rule is that it may lead to premature termination (e.g. in large flat areas).

Moreover, it is not always advisable to train the network until the smallest error possible has been reached. Usually we are not really interested to minimize the error on the training set, but we want to have small errors when *new* cases are submitted to the network. We are interested in the **generalization** performance of the network. **Over-learning** (**over-fitting**) may result in **memorization** of the training examples. The weights are tuned to fit every detail of each training example, including noise or inconsistent data (i.e. details or complete examples which are not representative of the general distribution of the data). One approach to overcome this problem is to split the available data into a training set and **validation set**. The training will be stopped when the error over the validation set (which consists of unseen examples) starts to increase. Because both, the training set and the validation set are used to fit the weights, neither the training set error nor on the validation set error will give an unbiased estimate of the generalization error. For this purpose, some part of the available data which must not be used in any way during training has to be reserved as a **test set** (**hold-out set**).

Sometimes sample data is rare. Splitting the few examples into two or even three sets may become a problem, because over-learning is most severe for small training sets. In these cases a **cross-validation** approach can be used. For instance, if there are n examples available, we can partition them into k disjoint subsets, each of size n/k approximately. Each of these subsets will become a validation set with the union of the remaining subsets building the training set. In this way we obtain k pairs of training and validation sets and can perform the training procedure k times. Each training will determine its own number of steps s and its own validation error. The mean of these estimates for s can be used as the stopping value in a final training on all n examples. The mean validation error will give an estimate for the generalization performance of the final network.

Memorization of training examples may not only be caused by too many training cycles. The size of the network has also a significant impact on its generalization performance. The danger of over-fitting increases, if we have too many adjustable parameters to train the network. Therefore large networks with a large number of hidden neurons and connections should be avoided. There are basically two approaches to find the 'right' number of neurons/connections: **pruning algorithms** and **growing algorithms**. Pruning algorithms start with a relatively large network and remove those connections or neurons, which have the lowest relevance to the performance of the network. Growing algorithms follow the opposite strategy. They start with a relatively small network and allow new connections or neurons to be added as long as these connections or neurons improve the performance of the network.

5. Data Preparation

Neural networks can only process numeric input and will produce numeric output. It is obvious that data preparation has to be performed, in order do deal with non-numeric input and output variables. But even if all data is numeric, some data preparation may be necessary. Typical transfer functions like the sigmoid function accept numeric input in any range, but they are sensitive to changes of the input only in a fairly limited range around the threshold value. Far away from this value there is a saturation effect so that changes of the input will result in (almost) no change of the output. This means that learning will become very slow. In order to avoid this effect input values are normally **scaled** into a sufficiently small range. For the same reason the initial values of the weights (see previous chapter) should be chosen out of a limited interval. Output variables may also have to be scaled in order to adjust them to the transfer functions of the output neurons. For example, the sigmoid function can only approximate output variables in the range of 0 to 1, and the boundary values 0 and 1 will never be produced exactly. Therefore target values outside the *open* interval (0, 1) should be avoided. If the training examples contain **outliers** (i.e. examples with extreme values of the input-output variables), scaling has to be performed carefully. Linear scaling using the extreme values as upper or lower bounds would force the variable values of the majority of the examples into very small ranges.

In case of non-numeric variables, raw data has to be **encoded** appropriately. For example, many-state nominal variables like direction of movement (left, right, straight, up, or down) could be encoded by assigning numbers to each state, e.g. left → 1, right → 2 etc. However, this might induce some ordering (more or less, better or worse) into the variable. Therefore **one-of-N** encoding might be a better choice. This encoding transforms the many-state variable into several binary variables (one for each state). In our example, the variable "direction of movement" would be replaced by 5 binary variables, and the state "right" would be transformed to the values 0, 1, 0, 0, 0 (assuming that the state "right" is assigned to the second variable). Unfortunately, one-of-N encoding increases the number of input neurons and therefore the network size. This may become a severe problem if the number of possible states is large and the number of the training examples is small.

Occasionally there are **missing values** for some of the features of the training examples. It is not always adequate to eliminate those examples from the training set, because this may lead to very small and/or biased training sets. In some cases it is appropriate to replace the missing values by 'typical' values, e.g. the average over observed values (excluding extreme values) in case of numerical variables and the most frequently observed value in case of nominal variables[6]. In other cases, the fact that data is missing may be informative in itself (e.g. a refusal to answer an awkward question or a missing output signal of an overloaded sensor), and a better choice would be to code the missing value as a specific value of the feature than to take the average of the available data.

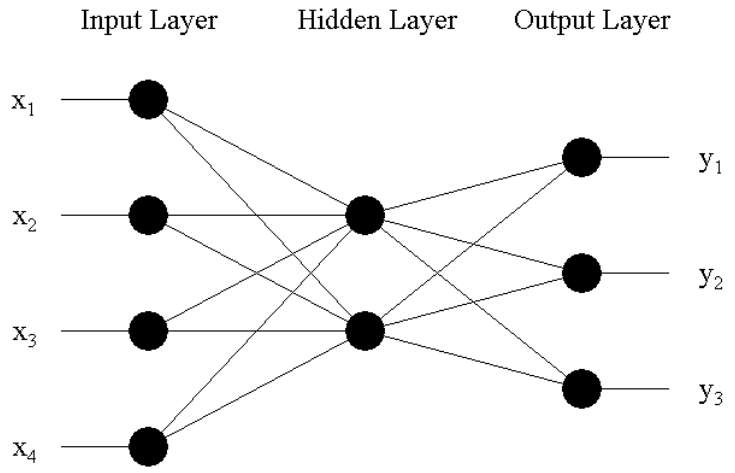
[6] sometimes more elaborated procedures will be needed, see e.g.: Bishop, C. M: Neural Networks for Pattern Recognition, New York, 1995, pp. 301 - 302

6. Exercises

1. Show that the threshold ϑ_j of an arbitrary transfer function $f_j(\Sigma_i w_{ij} \cdot o_i - \vartheta_j)$ of neuron j can be represented by an additional input neuron (the so called bias unit) which is connected to neuron j and has an output value of 1.
(o_i denotes the signal strength at the output of neuron i, w_{ij} denotes the weight of the connection between neurons i and j.)

2. Consider a neural net with stair-step threshold functions.
a. Suppose that you multiply all weights and thresholds by a constant. Will the behaviour change?
b. Suppose that you add a constant to all weights and thresholds. Will the behaviour change?

3. Describe the input-output relationship of the following net as a function F: $\mathbf{R}^4 \rightarrow \mathbf{R}^3$, where **R** denotes the set of real numbers; i.e.: Express the output $(y_1, y_2, y_3) = F(x_1, x_2, x_3, x_4)$ with the help of the transfer functions of the hidden layer and the output layer as well as with the weights connecting the layers. Use the following notation:
f_i^h: transfer function of the ith hidden neuron;
f_i^o : transfer function of the ith output neuron;
w_{ij}^h : weight connecting the ith input unit to the jth hidden unit;
w_{ij}^o : weight connecting the ith hidden unit to the jth output unit.



4. Give a rigorous proof of the XOR-problem: Show that the XOR-function can't be expressed by a perceptron without a hidden layer.

5. Consider the sigmoid transfer function $f(x) = 1/(1 + e^{-x+\vartheta})$.
a. Show how f'(x) (f'(x) denotes the first derivative of f(x)) can be calculated as an expression of f(x) and 1– f(x).
b. What does **a.** mean for the backpropagation algorithm?

6. Does it make any difference whether a) the network is trained completely with the input-output vectors of one class and then switching to another class or b) the input-output vectors are selected randomly from the training set?

7. Why do backpropagation nets produce similar outputs for similar inputs? How do the weights influence this behaviour?

8. Consider the following real valued function f: [-1, 1] → [0, 1].

$f(x) = 0.1;\ x \leq -0.8$
$f(x) = x + 0.9;\ -0.8 \leq x \leq 0$
$f(x) = 0.9;\ 0 < x \leq 0.4$
$f(x) = -2 \cdot x + 1.7;\ 0.4 < x \leq 0.8$
$f(x) = 0.1;\ x > 0.8$

a. Define a suitable training set containing 13 examples for a neural net which shall learn to approximate the function f.
b. Design a suitable topology of a feed-forward net for function f.

9. Suggest a reasonable input-output coding for a neural net for character recognition where the raw data consists of images with a size of 256*256 pixels.

10. The data set below contains 17 training examples - one example per line. Each example consists of five data items with the following meaning (read from left to right): weather outlook, temperature (F), humidity (%), windy, recommendation whether to play tennis or not. The question mark '?' denotes a missing value.

sunny,85,85,FALSE,no
sunny,80,90,TRUE,no
overcast,83,86,FALSE,yes
rainy,70,96,FALSE,yes
rainy,68,80,FALSE,yes
rainy,65,70,TRUE,no
overcast,64,65,TRUE,yes
sunny,72,95,FALSE,no
sunny,69,70,FALSE,yes
rainy,75,80,FALSE,yes
sunny,75,70,TRUE,yes
overcast,72,90,TRUE,yes
overcast,81,75,FALSE,yes
rainy,71,91,TRUE,no
rainy,70,90,TRUE,no
?, ?, 90, TRUE, no

The training data shall be presented to a feed-forward neural network. Transform the data appropriately and define a suitable network topology.

In der Reihe FINAL sind bisher erschienen:

1. Jahrgang 1991:

1. Hinrich E. G. Bonin; Softwaretechnik, Heft 1, 1991 (ersetzt durch Heft 2, 1992).
2. Hinrich E. G. Bonin (Herausgeber); Konturen der Verwaltungsinformatik, Heft 2, 1991 (überarbeitet und erschienen im Wissenschaftsverlag, Bibliographisches Institut & F. A. Brockhaus AG, Mannheim 1992, ISBN 3-411-15671-6).

2. Jahrgang 1992:

1. Hinrich E. G. Bonin; Produktionshilfen zur Softwaretechnik --- Computer-Aided Software Engineering --- CASE, Materialien zum Seminar 1992, Heft 1, 1992.
2. Hinrich E. G. Bonin; Arbeitstechniken für die Softwareentwicklung, Heft 2, 1992 (3. überarbeitete Auflage Februar 1994), PDF-Format (Passwort: arbeiten).
3. Hinrich E. G. Bonin; Object-Orientedness --- a New Boxologie, Heft 3, 1992.
4. Hinrich E. G. Bonin; Objekt-orientierte Analyse, Entwurf und Programmierung, Materialien zum Seminar 1992, Heft 4, 1992.
5. Hinrich E. G. Bonin; Kooperative Produktion von Dokumenten, Materialien zum Seminar 1992, Heft 5, 1992.

3. Jahrgang 1993:

1. Hinrich E. G. Bonin; Systems Engineering in Public Administration, Proceedings IFIP TC8/ WG8.5: Governmental and Municipal Information Systems, March 3--5, 1993, Lüneburg, Heft 1, 1993 (überarbeitet und erschienen bei North-Holland, IFIP Transactions A-36, ISSN 0926-5473).
2. Antje Binder, Ralf Linhart, Jürgen Schultz, Frank Sperschneider, Thomas True, Bernd Willenbockel; COTEXT --- ein Prototyp für die kooperative Produktion von Dokumenten, 19. März 1993, Heft 2, 1993.
3. Gareth Harries; An Introduction to Artificial Intelligence, April 1993, Heft 3, 1993.
4. Jens Benecke, Jürgen Grothmann, Mark Hilmer, Manfred Hölzen, Heiko Köster, Peter Mattfeld, Andre Peters, Harald Weiss; ConFusion --- Das Produkt des AWÖ-Projektes 1992/93, 1. August 1993, Heft 4, 1993.
5. Hinrich E. G. Bonin; The Joy of Computer Science --- Skript zur Vorlesung EDV ---, September 1993, Heft 5, 1993 (4. ergänzte Auflage März 1995).
6. Hans-Joachim Blanke; UNIX to UNIX Copy --- Interactive application for installation and configuration of UUCP ---, Oktober 1993, Heft 6, 1993.

4. Jahrgang 1994:

1. Andre Peters, Harald Weiss; COMO 1.0 --- Programmierumgebung für die Sprache COBOL --- Benutzerhandbuch, Februar 1994, Heft 1, 1994.
2. Manfred Hölzen; UNIX-Mail --- Schnelleinstieg und Handbuch ---, März 1994, Heft 2, 1994.
3. Norbert Kröger, Roland Seen; EBrain --- Documentation of the 1994 AWÖ-Project Prototype ---, June 11, 1994, Heft 3, 1994.
4. Dirk Mayer, Rainer Saalfeld; ADLATUS --- Documentation of the 1994 AWÖ-Project Prototype -- -, July 26, 1994, Heft 4, 1994.
5. Ulrich Hoffmann; Datenverarbeitungssystem 1, September 1994, Heft 5, 1994. (2. überarbeitete Auflage Dezember 1994)
6. Karl Goede; EDV-gestützte Kommunikation und Hochschulorganisation, Oktober 1994, Heft 6 (Teil 1), 1994.

7. Ulrich Hoffmann; Zur Situation der Informatik, Oktober 1994, Heft 6 (Teil 2), 1994.

5. Jahrgang 1995:

1. Horst Meyer-Wachsmuth; Systemprogrammierung 1, Januar 1995, Heft 1, 1995.
2. Ulrich Hoffmann; Datenverarbeitungssystem 2, Februar 1995, Heft 2, 1995.
3. Michael Guder / Kersten Kalischefski / Jörg Meier / Ralf Stöver / Cheikh Zeine; OFFICE-LINK --- Das Produkt des AWÖ-Projektes 1994/95, März 1995, Heft 3, 1995.
4. Dieter Riebesehl; Lineare Optimierung und Operations Research, März 1995, Heft 4, 1995.
5. Jürgen Mattern / Mark Hilmer; Sicherheitsrahmen einer UTM-Anwendung, April 1995, Heft 5, 1995.
6. Hinrich E. G. Bonin; Publizieren im World-Wide Web --- HyperText Markup Language und die Kunst der Programmierung ---, Mai 1995, Heft 6, 1995
7. Dieter Riebesehl; Einführung in Grundlagen der theoretischen Informatik, Juli 1995, Heft 7, 1995
8. Jürgen Jacobs; Anwendungsprogrammierung mit Embedded-SQL, August 1995, Heft 8, 1995
9. Ulrich Hoffmann; Systemnahe Programmierung, September 1995, Heft 9, 1995 (ersetzt durch Heft 1, 1999)
10. Klaus Lindner; Neuere statistische Ergebnisse, Dezember 1995, Heft 10, 1995

6. Jahrgang 1996:

1. Jürgen Jacobs / Dieter Riebesehl; Computergestütztes Repetitorium der Elementarmathematik, Februar 1996, Heft 1, 1996
2. Hinrich E. G. Bonin; "Schlanker Staat" & Informatik, März 1996, Heft 2, 1996
3. Jürgen Jacobs; Datenmodellierung mit dem Entity-Relationship-Ansatz, Mai 1996, Heft 3, 1996
4. Ulrich Hoffmann; Systemnahe Programmierung, (2. überarbeitete Auflage von Heft 9, 1995), September 1996, Heft 4, 1996 (ersetzt durch Heft 1, 1999).
5. Dieter Riebesehl; Prolog und relationale Datenbanken als Grundlagen zur Implementierung einer NF2-Datenbank (Sommer 1995), November 1996, Heft 5, 1996

7. Jahrgang 1997:

1. Jan Binge, Hinrich E. G. Bonin, Volker Neumann, Ingo Stadtsholte, Jürgen Utz; Intranet-/Internet- Technologie für die Öffentliche Verwaltung --- Das AWÖ-Projekt im WS96/97 --- (Anwendungen in der Öffentlichen Verwaltung), Februar 1997, Heft 1, 1997
2. Hinrich E. G. Bonin; Auswirkungen des Java-Konzeptes für Verwaltungen, FTVI'97, Oktober 1997, Heft 2, 1997

8. Jahrgang 1998:

1. Hinrich E. G. Bonin; Der Java-Coach, Oktober 1998, Heft 1, 1998 (CD-ROM, PDF-Format; aktuelle Fassung)

2. Hinrich E. G. Bonin (Hrsg.); Anwendungsentwicklung WS 1997/98 --- Programmierbeispiele in COBOL & Java mit Oracle, Dokumentation in HTML und tcl/tk, September 1998, Heft 2, 1998 (CD-ROM)

3. Hinrich E. G. Bonin (Hrsg); Anwendungsentwicklung SS 1998 --- Innovator, SNiFF+, Java, Tools, Oktober 1998, Heft 3, 1998 (CD-ROM)

4. Hinrich E. G. Bonin (Hrsg); Anwendungsentwicklung WS 1998 --- Innovator, SNiFF+, Java, Mail und andere Tools, November 1998, Heft 4, 1998 (CD-ROM)

5. Hinrich E. G. Bonin; Persistente Objekte --- Der Elchtest für ein Java-Programm, Dezember 1998, Heft 5, 1998 (CD-ROM)

9. Jahrgang 1999:

1. Ulrich Hoffmann; Systemnahe Programmierung (3. überarbeitete Auflage von Heft 9, 1995), Juli 1999, Heft 1, 1999 (CD-ROM und Papierform), Postscript-Format, zip-Postscript-Format, PDF-Format und zip-PDF-Format.

10. Jahrgang 2000:

1. Hinrich E. G. Bonin; Citizen Relationship Management, September 2000, Heft 1, 2000 (CD-ROM und Papierform), PDF-Format --- Password: `arbeiten`

2. Hinrich E. G. Bonin; WI>DATA --- Eine Einführung in die Wirtschaftsinformatik auf der Basis der Web_Technologie, September 2000, Heft 2, 2000 (CD-ROM und Papierform), PDF-Format --- Password: `arbeiten`

3. Ulrich Hoffmann; Angewandte Komplexitätstheorie, November 2000, Heft 3, 2000 (CD-ROM und Papierform), PDF-Format

4. Hinrich E. G. Bonin; Der kleine XMLer, Dezember 2000, Heft 4, 2000 (CD-ROM und Papierform), PDF-Format, aktuelle Fassung --- Password: `arbeiten`

11. Jahrgang 2001:

1. Hinrich E. G. Bonin (Hrsg.): 4. SAP-Anwenderforum der FHNON, März 2001, (CD-ROM und Papierform), Downloads & Videos.

2. J. Jacobs / G. Weinrich; Bonitätsklassifikation kleiner Unternehmen mit multivariater linear Diskriminanzanalyse und Neuronalen Netzen; Mai 2001, Heft 2, 2001, (CD-ROM und Papierform), PDF-Format und MS Word DOC-Format --- Password: `arbeiten`

3. K. Lindner; Simultanttestprozedur für globale Nullhypothesen bei beliebiger Abhängigkeitsstruktur der Einzeltests, September 2001, Heft 3, 2001 (CD-ROM und Papierform).

12. Jahrgang 2002:

1. Hinrich E. G. Bonin: Aspect-Oriented Software Development. März 2002, Heft 1, 2002 (CD-ROM und Papierform), PDF-Format --- Password: `arbeiten`.

2. Hinrich E. G. Bonin: WAP & WML --- Das Projekt Jagdzeit ---. April 2002, Heft 2, 2002 (CD-ROM und Papierform), PDF-Format --- Password: `arbeiten`.

3. Ulrich Hoffmann: Ausgewählte Kapitel der Theoretischen Informatik (CD-ROM und Papierform), PDF-Format.

4. Jürgen Jacobs / Dieter Riebesehl; Computergestütztes Repetitorium der Elementarmathematik, September 2002, Heft 4, 2002 (CD-ROM und Papierform), PDF-Format.

5. Verschiedene Referenten; 3. Praxisforum "Systemintegration", 18.10.2002, Oktober 2002, Heft 5, 2002 (CD-ROM und Papierform), Praxisforum.html (Web-Site).

13. Jahrgang 2003:

1. Ulrich Hoffmann; Ausgewählte Kapitel der Theoretischen Informatik; Heft 1, 2003, (CD-ROM und Papierform) PDF-Format.

2. Dieter Riebesehl; Mathematik 1, Heft 2, 2003, (CD-ROM und Papierform) PDF-Format.

3. Ulrich Hoffmann; Mathematik 1, Heft 3, 2003, (CD-ROM und Papierform) PDF-Format und Übungen

4. Verschiedene Autoren; Zukunft von Verwaltung und Informatik, Festschrift für Heinrich Reinermann, Heft 4, 2003, CD-ROM und Papierform) PDF-Format

Herausgeber:
Prof. Dr. Dipl.-Ing. Dipl.-Wirtsch.-Ing. Hinrich E. G. Bonin
Fachhochschule Nordostniedersachsen (FH NON)
Volgershall 1
D-21339 Lüneburg

email: bonin@fhnon.de

Verlag:
Eigenverlag (Fotographische Vervielfältigung), FH NON

Erscheinungsweise:
ca. 4 Hefte pro Jahr.
Für unverlangt eingesendete Manuskripte wird nicht gehaftet. Sie sind aber willkommen.