Data Mining with Neural Networks

Svein Nordbotten

Svein Nordbotten & Associates

Bergen 2006
Contents

Preface .................................................................................................................................................. 5

Session 1: Introduction ......................................................................................................................... 6
   Introduction ......................................................................................................................................... 6
   Data mining ......................................................................................................................................... 6
   What is a neural network? .................................................................................................................. 7
   Neural networks and Artificial intelligence ...................................................................................... 10
   A brief historic review ...................................................................................................................... 10
   Systems and models .......................................................................................................................... 11
   State transition tables ..................................................................................................................... 13
   State diagrams .................................................................................................................................. 14
   Neurons - the basic building bricks ................................................................................................. 15
   Perceptron ......................................................................................................................................... 18
   Neural network properties ............................................................................................................... 21
   Exercises ........................................................................................................................................... 22

Session 2: Feed-forward networks ..................................................................................................... 23
   Two types of network ....................................................................................................................... 23
   Learning ............................................................................................................................................ 24
   Non-linearly separable classes and multi-layer networks ............................................................... 28
   Multi-layer networks ....................................................................................................................... 29
   Backpropagation learning ............................................................................................................... 30
   Measuring learning ........................................................................................................................... 31
   Generalization ................................................................................................................................. 33
   Classification revisited ..................................................................................................................... 34
   Exercises ........................................................................................................................................... 35

Session 3: BrainMaker software ......................................................................................................... 36
   Software ........................................................................................................................................... 36
<table>
<thead>
<tr>
<th>Section</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>BrainMaker specifications</td>
<td>101</td>
</tr>
<tr>
<td>Training the network</td>
<td>105</td>
</tr>
<tr>
<td>Analysis of training</td>
<td>106</td>
</tr>
<tr>
<td>Running the network in production</td>
<td>109</td>
</tr>
<tr>
<td>Financial application</td>
<td>112</td>
</tr>
<tr>
<td>Exercises</td>
<td>122</td>
</tr>
<tr>
<td>Session 8: Imputation</td>
<td>124</td>
</tr>
<tr>
<td>Small area statistics</td>
<td>124</td>
</tr>
<tr>
<td>Data available</td>
<td>124</td>
</tr>
<tr>
<td>Sizes of census tracts</td>
<td>125</td>
</tr>
<tr>
<td>Variables, imputations and mse</td>
<td>125</td>
</tr>
<tr>
<td>Imputation estimates for Municipality I</td>
<td>128</td>
</tr>
<tr>
<td>Imputation estimates for Municipality II</td>
<td>129</td>
</tr>
<tr>
<td>Extreme individual errors</td>
<td>131</td>
</tr>
<tr>
<td>Four statements needing further research</td>
<td>131</td>
</tr>
<tr>
<td>Exercises</td>
<td>131</td>
</tr>
<tr>
<td>Session 9: Optimization</td>
<td>133</td>
</tr>
<tr>
<td>Additional software from CSS</td>
<td>133</td>
</tr>
<tr>
<td>The Genetic Training Option</td>
<td>133</td>
</tr>
<tr>
<td>Optimization of networks</td>
<td>133</td>
</tr>
<tr>
<td>Genetic training</td>
<td>137</td>
</tr>
<tr>
<td>Exercises</td>
<td>142</td>
</tr>
<tr>
<td>Session 10: Other neural networks</td>
<td>143</td>
</tr>
<tr>
<td>Different types of neural networks</td>
<td>143</td>
</tr>
<tr>
<td>Simple linear networks</td>
<td>144</td>
</tr>
<tr>
<td>Incompletely connected feed-forward nets</td>
<td>145</td>
</tr>
<tr>
<td>Multi-layer feed-forward networks with by-pass connections</td>
<td>146</td>
</tr>
<tr>
<td>Associative memories</td>
<td>146</td>
</tr>
<tr>
<td>Self-organizing maps</td>
<td>148</td>
</tr>
<tr>
<td>Adaptive Resonance Theory</td>
<td>149</td>
</tr>
<tr>
<td>Exercises</td>
<td>149</td>
</tr>
<tr>
<td>A bibliography for further studies</td>
<td>150</td>
</tr>
</tbody>
</table>
Preface

This is an on-line course about Data Mining by Artificial Neural Networks (NN) and based on the BrainMaker software developed and distributed by California Scientific Software. CSS also provided their software at special student conditions. The course was initially given as a face-to-face course at the University of Bergen and later at the University of Hawaii in 2000. Later it was revised and developed as an online course for these universities and other institutions.

The present edition is an extract of the text and illustrations from the course for those students who wanted a reference to the course content. It is hoped that also other readers may find the presentation interesting and useful.

Bergen, July 2006

Svein Nordbotten
Session 1: Introduction

Introduction

This course has previously been given as face-to-face lectures and as net-based ALN sessions (Figure 1). The illustrations are therefore being modified, dated and numbered according to the time and they were prepared for the course. The text contains a number of hyperlinks to related topics. The links are never pointing forward, only to topics in the current and previous sessions. If you wish, you are free to print out text as well as figures by clicking the 'Print' icon in your Windows' tool bar. You can always get back to the text by clicking the 'Back' icon in your browser window after watching a figure or a linked text.

Data mining

Back in the stone age of the 1960's, people had visions about saving all recorded data in data archives to be ready for future structuring, extraction, analysis and use [Nordbotten 1967]. Even though the amount of data recorded was insignificant compared with what is recorded today, the technology was not yet developed for this task. Only in the last decade, the IT technology permitted that the visions could start to be realized in the form of data warehouses. Still, the warehouses are mainly implemented in large corporations and organizations wanting to preserve their data for possible future use.

When stored, data in a warehouse were usually structured to suit the application generating the data. Other applications may require re-structuring of the data. To accomplish a rational re-structuring, it is useful to know about the relations embedded in the data. The purpose of data
mining is to explore, frequently hidden and unknown, relationships to restructure data for analysis and new uses.

Common for all data mining tasks is the existence of a collection of data records. Each record represents characteristics of some object, and contains measurements, observations and/or registrations of the values of these characteristics or variables.

Data mining tasks can be grouped according to the assumptions of the degree of specification of the problems made prior to the work. We can for instance distinguish between tasks which are:

1. **Well specified**: This is the case when a theory or model exists and it is required empirically to test and measure the relationships. The models of the econometricians, biometricians, etc. are well known of this type of tasks.
2. **Semi-specified**: Explanations of a subset of dependent variables are wanted, but no explicit theory exists. The task is to investigate if the remaining variables can explain the variations in the first subset of variables. Social research frequently approach problems in this way.
3. **Unspecified**: A collection of records with a number of variables is available. Are there any relations among the variables which can contribute to an understanding of their variation?

In the present course, we shall concentrate on the semi-specified type of tasks.

Parallel with the techniques for efficient storage of data in warehouses, identification and development of methods for data mining has taken place. In contrast to warehousing, data exploration has long traditions within several disciplines as for instance statistics. In this course, we shall not discuss the complete box of data mining tools, but focus on one set of tools, the feed-forward Neural Networks, which has become a central and useful component.

**What is a neural network?**

Neural networks is one name for a set of methods which have varying names in different research groups. Figure 2 shows some of the most frequently used names. We note the
**Figure 2: Terms used for referring to the topic**

different names used, but do not spend time discussing which is the best or most correct. In this course, we simply refer to this type of methods as **Neural Networks** or **NN** for short.

**Figure 3** shows varying definitions of **Neural Networks**. The different definitions reflect the

---

**Definitions of neural networks**

A neural network is a **parallel distributed processor** that has a propensity for storing experiential knowledge and making it available for users (Rumelhart et al).

Neural computing is the study of networks of **adaptable nodes** which, through a process of learning from task examples, store experiential knowledge and make it available for use (Alexander).

A neural network is a **finite-state machine** made up of elementary units called neurons (Minsky).

---

**Figure 3: NN definitions**

professional interest of the group to which the author belongs. The first definition of the figure indicates that **Rumelhart** and his colleagues are particularly interested in the functioning of neural networks and pointed out that NN can be considered as a large collection of simple, distributed processing units working in parallel to represent and making knowledge available to users. The second author, **Alexander**, emphasizes the learning process as represented by nodes
adapting to task examples. Minsky's definition states that formally a neural network can be considered as a finite-state machine. The definitions are supplementing each other in characterizing a neural network system.

The formal definition of is probably best formulated by Hecht-Nielsen:

"A neural network is a parallel, distributed information processing structure consisting of processing elements (which can possess a local memory and can carry out localized information processing operations) interconnected via unidirectional signal channels called connections. Each processing element has a single output connection that branches ("fans out") into as many collateral connections as desired; each carries the same signal - the processing element output signal. The processing element output signal can be of any mathematical type desired. The information processing that goes on within each processing element can be defined arbitrarily with the restriction that it must be completely local; that is, it must depend only on the current values of the input signals arriving at the processing element via impinging connections and on the values stored in the processing element's local memory."

Neural networks models were initially created as description and explanation of the biological neural network of the human brain. Because of the size and the efficiency of the biological neural network, an artificial computer-based NN can reflect only a small fraction of the complexity and efficiency of a human neural network (Figure 4).

**Characteristics of the human neural network**

The brain is estimated to have more than $10^{12}$ neurons in the cortex, and $10^{22}$ synapses connecting the neurons to each other.

Energetic efficiency of the brain is $10^{10}$ Joules per operation/sec., energetic efficiency of the computer is $10^6$ Joules per operation/sec (Joule: work done by 1 newton through a distance of 1 m).

**Figure 4: Characteristics of the human brain**

What can NN be used for? It can be used to model special human brain functions, to investigate if a modeled hypothesis of a certain brain function behaves in correspondence with what can be observed of the real brain [Lawrence]. NN can also be considered as a logical machine and as a universal function approximation. NN are frequently used for classifying multi-dimensional data or patterns into categories, or to make conditional predictions very similar to what multivariate
statistical data analysis do [Bishop]. The domains of applications are many and we shall discuss some examples during the course.

**Neural networks and Artificial intelligence**

Artificial intelligence is branch of information and computer science working with computers to simulate human thinking. The topic can be divided into

- the **logical/symbolic** approach to which for instance the expert systems belong. The term 'logical' reflects that according to this approach, the purpose is to explain by logical rules how a human arrives to the solution of a problem.
- the **subsymbolic** approach on the other side, tries to explain a solution to a problem by the processes below the logical rules. The neural networks are typical representatives for the subsymbolic approach [Sowa].

Since the 1950's, a competition has existed between the members of the two approaches. More recently, similarities and relations have been identified [Gallant, Nordbotten 1992], and the possibilities of taking advantage of both by constructing hybrid solutions.

**A brief historic review**

In Figure 5, a few of the main events in the history of NN are listed. The history of Neural Networks started as a paper by McCulloch and Pitts in 1943 presenting a formal mathematical model describing the working of a human brain.

<table>
<thead>
<tr>
<th>Year</th>
<th>Event</th>
</tr>
</thead>
<tbody>
<tr>
<td>1943</td>
<td>McCulloch and Pitts: <em>A Logical Calculus</em>...</td>
</tr>
<tr>
<td>1948</td>
<td>Wiener: <em>Cybernetics</em></td>
</tr>
<tr>
<td>1949</td>
<td>Hebb: <em>The Organization of Behaviour</em></td>
</tr>
<tr>
<td>1952</td>
<td>Ashby: <em>Design for a Brain</em></td>
</tr>
<tr>
<td>1958</td>
<td>Rosenblatt: <em>Perceptron</em></td>
</tr>
<tr>
<td>1960</td>
<td>Widrow and Hoff: <em>Adaline</em></td>
</tr>
<tr>
<td>1963</td>
<td>Nilson: <em>Learning Machines</em></td>
</tr>
<tr>
<td>1969</td>
<td>Minsky and Papert: <em>Perceptron</em></td>
</tr>
<tr>
<td>1974</td>
<td>Werbos: <em>Generalized Multiple Regression</em></td>
</tr>
<tr>
<td>1986</td>
<td>Rumelhart and McClelland: <em>Parallel Distributed Proc.</em></td>
</tr>
</tbody>
</table>

**Figure 5: Milestones in the history of NN**

Just after the end of the World War II, Wiener introduced the concept Cybernetics, the study of the processing of information by machines. He did not know that Ampère had been thinking along the same lines and coined the word 100 years earlier [Dyson 1997]. Ashby 1971 contributed much to the cybernetic by modeling dynamic systems by means of the abstract
machines. In psychology, Hebb wrote a paper in 1949 about learning principles which became one of the cornerstones for the development of training algorithms for NN.

Rosenblatt was one of the early pioneers in applying the theory of NN in the 1950's. He designed the NN model known as the Perceptron, and proved that it could learn from examples. Widrow and Hoff worked at the same time as Rosenblatt and developed the ADELIE model with the delta algorithm for adaptive learning. In the 1960's, strong optimism characterized the NN camp which had great expectations for their approach. In 1969, Minsky and Papert published a book in which they proved that the power of the single-layer Neural Networks was limited, and that multi-layer networks were required for solving more complex problems. However, without learning algorithms for multi-layer networks, little progress could be made.

A learning algorithm for multi-layer networks was in fact invented by Werbos and used in his Ph.d. dissertation already in 1973. His work remained unknown for most researchers until the algorithm was re-invented independently by Le Cun 1985 and Parker 1985, and known as the Backpropagation algorithm in the early 1980's. Rumelhart, McClelland and others made the backpropagation algorithm worldwide known in a series of publications in the middle 1980's.

During the last two decades, a number of new methods have been developed and NN has been accepted as a well based methodology. Of particular interest is the interpretation of NN based on statistical theory. One of the main contributors is Bishop.

**Systems and models**

A system is a collection of interrelated objects or events which we want to study. A formal, theoretical basis for system thinking was established by Bertalanffy. A system can for instance be cells of a human being, components of a learning process, transactions of an enterprise, parts of a car, inhabitants of a city, etc. It is convenient to assume the existence of another system surrounding the considered system. For practical reasons, we name the surrounding system the environment system. In many situations, research is focused on how the two systems interact. The interaction between the systems is symbolized by two arrows in Figure 6.
Assume that the system considered is a human brain, and that we want to study how it is organized. In the lower part of Figure 7, we recognize the interaction with the environment from the previous picture, but in addition, the brain has been detailed with components assigned to different tasks. One component of receptor cells is receiving input stimuli from sensors outside the brain, and another component is sending output signals to the muscles in the environment system.

Nobody would believe that this is a precise description of the human brain; it is only a simple description. It is essential to distinguish between the system to be described, and the description of this system (Figure 8). When this distinction is used, we refer to the description of the system...
as a model of the system. We consider NN as a model of the human brain, or perhaps more correctly, as a model of a small part of the brain. A model is always a simplified or idealized version of a system in one or more ways. The purpose of a model is to provide a description of the system which focuses on the main aspect of interest and is convenient as a tool for exploring, analyzing and simulating the system. If it was an exact replica, we would have two identical systems. A model will usually focus on system aspects considered important for the model maker’s purpose ignoring aspects not significant for this purpose. Note that a model is also a system itself.

![System and model](image)

**Figure 8: NN as a model of the brain**

Figure 8 showed a graphical model. There are many types of models. In Figure 9, an algebraic model is displayed. It is a finite-state machine as used by Minsky and models a dynamic stimuli-response system. It assumes that time is indexed by points to which the system state characteristics can be associated. The state of the system at time t is represented by \( Q(t) \) and the stimuli received from the environment at the same time by \( S(t) \). The behavior of the system is represented in the model by two equations; the first explains how the state of the system changes from time \( t \) to time \( t+1 \). The second equation explains the response from the system to the environment at time \( t+1 \).

**State transition tables**

In Figure 9, the basic functions of a finite-state machine were presented. The finite-state machine can alternatively be modeled as a transition table frequently used in cybernetics, or as a state diagrams. In Figure 10, the NN with 2 neurons just discussed can be represented by 2 transition tables describing how the state and the response of the NN change from time \( t \) to time \( t+1 \). In the upper table of Figure 10 representing the control neuron, \( c_0, c_1 \) and \( c_{-1} \) represent the 3 input alternative values to the neuron while \( q_0 \) and \( q_1 \) indicate the alternative states of the neuron at time \( t-1 \). The cells of the table represent the new output from the neuron at time \( t \). The second
table represents the controlled neuron. Here \( q_0 \) and \( q_1 \) are the two alternative inputs at time \( t \) from the control neuron, \( s_0 \) and \( s_1 \) are the 2 alternative input values to the primary neuron at time \( t \) and the cells are the alternative values of the output at time \( t+1 \) of the primary neuron. Note that the value of the control input values at time \( t-1 \) influences the output value of the primary neuron at time \( t+1 \).

**State diagrams**

A system is also often described by a state diagram as indicated at the right side of Figure 10. The hexagons represent states of system components, while the arrows represent alternative transitions from one state to another. Note that some of the hexagons represent outputs (responses) and not states in the meaning of Figure 9. The symbols at the tail of an arrow are the alternative inputs.

---

### Model of finite-state machines

**State equation:**

\[
Q(t+1) = G(Q(t), S(t))
\]

**Response equation:**

\[
R(t+1) = F(Q(t), S(t))
\]

---

**Figure 9: Finite state machines**

---

### Transition tables and diagrams

<table>
<thead>
<tr>
<th>Transition tables:</th>
<th>State diagram:</th>
</tr>
</thead>
<tbody>
<tr>
<td>( G )</td>
<td>( q_0 )</td>
</tr>
<tr>
<td>( c_0 )</td>
<td>0</td>
</tr>
<tr>
<td>( c_1 )</td>
<td>1</td>
</tr>
<tr>
<td>( c_2 )</td>
<td>0</td>
</tr>
<tr>
<td>( F )</td>
<td>( q_0 )</td>
</tr>
<tr>
<td>( s_0 )</td>
<td>0</td>
</tr>
<tr>
<td>( s_1 )</td>
<td>0</td>
</tr>
</tbody>
</table>
Figure 10: Transition tables

Consider the hexagon \( q_0 \). It represents the \( q_0 \), the closed state of the control neuron, and has 3 arrows out. The one directed up represent the transition of the primary neurons. This neuron will either get a 0, or a 1 as input values, but will always be in state \( r_0 \) when the control neuron is in closed state. The state \( q_0 \) will be unchanged if the input values are either -1 or 0, but if the input value is 1, the control neuron will change state to \( q_1 \). It will stay in this state if the control input values are either 0 or 1, but return to state \( q_0 \) if the input value to the control neuron is -1. If the control neuron is in state \( q_1 \), and the primary input value is 0, the state of the primary neuron will be \( r_0 \), while an input value 1 will give the primary neuron the state \( r_1 \).

A more complex finite-state machine can add binary numbers. This transition diagram in Figure 11 represent a machine which can add 2 bits numbers in which the least significant bit is the left

Figure 11: Serial adder represented by a state diagram

The red numbers in the middle of an arrow represents the output of the transition. For example, the decimal number 3 is 11 a binary number and the decimal number 1 is represented as 10. The sum of these to addends is 4 or 0011 as a binary number. Starting with the left bits, the first pair will be 1+1. The initial state is 'No carry' and the input 11 is at the tail of an arrow to the 'Carry' state with 0 as output. The next pair of bits is 01 and the arrow from 'Carry' with this input gives again an output 0. The last pair of input values is 00 which is represented with an arrow back to 'No carry' and an output 1. The final output will therefore be 001, which is the correct result.

Neurons - the basic building bricks

Transition tables and state diagrams are useful when we understand the behavior of a system completely as observed from outside. If not, we need to study the internal parts and their
interactions which we will do by means of neurons and their interconnections. An interesting fact is that finite-state machines and NN are two different aspects of the same type of systems.

Let us return to the human brain system. We have assumed that the brain is composed of a large number of brain cells called neurons. Figure 12 illustrates how the biological neuron is

![Human neuron](image)

**Figure 12: The basic parts of a human neuron**

often depicted in introductory texts. This graphical model of the neurons indicates that it has several different components. For our purpose, we identify 4 main components: the cell's synapses which are receiving stimuli from other neurons, the cell body processing the stimuli, the dendrites which are extensions of the cell body, and the axon sending the neuron response to other neurons. Note that there is only one axon from each cell, which, however, may branch out to many other cells.

Working with artificial neurons, Figure 13 indicates how we can simplify the model even more.

![Model of a neuron](image)
Figure 13: The NN model of a neuron

We denote the axons from other neurons by connection variables $x$, the synapses by the weights $w$, and the axon by the output variable $y$. The cell body itself is considered to have two functions. The first function is integration of all weighted stimuli symbolized by the summation sign. The second function is the activation which transforms the sum of weighted stimuli to an output value which is sent out through connection $y$. In the neural network models considered in this course, the time spent on transforming the incoming stimuli to a response value is assumed to be one time unit while the propagation of the stimuli from one neuron to the next is momentary. In the feed-forward NN, the time dimension is not important.

Figure 14 shows several activation functions frequently used in modeling neural networks.

Figure 14: Three activation functions

Types of activation functions $F$

threshold  linear  sigmoid

Usually the neurons transform the sum of weighted values received as an argument to an output value in the range $-1$ to $+1$, or, alternatively, $0$ to $+1$. The step function is the simplest. An argument, the sum of the weighted input variables, is represented along the $x$-axis. The function will either result in an output value $-1$ if the argument is less than zero (or some other predetermined value), or a value $+1$ if the argument is $0$ or positive (on or to the predetermined value). The linear activation function value is $0$ if the argument is less than a lower boundary, increasing linearly from $0$ to $+1$ for arguments equal or larger than the lower boundary and less than an upper boundary, and $+1$ for all arguments equal or greater than a given upper boundary. An important activation function is the sigmoid which is illustrated to the right in Figure 14. The sigmoid function is non-linear, but continuous, and has a function value range between $0$ and $+1$. As we shall see later, it has the properties which make it very convenient to work with.
Perceptron

Neurons are used as building bricks for modeling a number of different neural networks. The NN can be classified in two main groups according to the way they learn (Figure 15). One group contains the networks which can learn by supervision, i.e. they can be trained on a set of example problems with associated target solutions. During the training, the examples are repetitively exposed for the NN which are adjusting to the examples. As part of the training, the NN can be continuously tested for their ability to reproduce the correct solutions to the examples. The second main group is consists of the networks which learn unsupervised. These networks learn by identifying special features in the problems they are exposed to. They are also called self-organizing networks or maps. Kohonen is one of the pioneers in this field of networks.

In this course, we concentrate our attention on the networks which can be trained by supervised learning. The first type of networks we introduce in Figure 16 is the single-layer network. It is
Figure 16: Single-layer NN

called a single-layer network because it has only one layer of neurons between the input sources and the output. The perceptron introduced by Rosenblatt and much discussed in the 1960's, was a single-layer network. Note that some authors also count the input sources as a layer and denoted the perceptron as a two-layer network.

A simple perceptron consists of one neuron with 2 input variables, \( x_1 \) and \( x_2 \). It has a step activation function which produces a binary output value. Assume that the step function responds with \(-1\) if the sums of the input values are negative and with \(+1\) if the sum is zero or positive. If we investigate this NN further, it is able to classify all possible pairs of input values in 2 categories. These 2 categories can be separated by a line as illustrated in Figure 17. The line...
dividing the $x_1, x_2$ space is determined by the weights $w_1$ and $w_2$. Only problems corresponding to classifying inputs into linear separable categories can be solved by the single-layer networks. This was one of the limitations pointed out by Minsky and Papert in their discussion of NN in the late 1960s.

A network with more than a one output neuron, as shown in Figure 16, can classify the input values in more than two categories. The condition for successful classification is still that the input points are linearly separable.

In some systems, it is necessary to control the functioning of a neuron subject to some other input. Consider a neuron with single primary binary input connection, a step activity function with threshold value 2 generating output 0 if the input sum is less than 2 and 1 if it is 2 or greater (Figure 18). Let the neuron have a secondary, control input with values 0 or 1. The neuron will reproduce all values from the primary input source as long as the secondary control input is 1. When the control input value is changed to 0, the reproduction of values from the primary input connection will be stopped. In this way, the processing of the stream of input through the primary input connection can be controlled from the secondary input source.

![Control of flow through a neuron](image)

**Figure 18: Controlling a neuron**

It may, however, be inconvenient to generate a continuous sequence of control 1 values to keep copying of the primary input stream open. If we extend the network with a second, control neuron, we can create an on/off switch. Let the control neuron have 2 input connections, a step activity function with threshold value 1 and binary output as illustrated in Figure 19. The first of
**Figure 19: A simple net with memory**

The inputs is the on/off signals which in this case have the values on=1, no change=0 and off=-1. The second input is a feedback loop from the control neuron's output value. Inspection of the system shows that the sequence of primary inputs to the first neuron will pass through this neuron, if a control value 1 has switched the control neuron on. Reproduction of the primary input stream will be broken, if a control input -1 is received by the control neuron.

**Neural network properties**

Some of the characteristic properties of a neural network are summarized in **Figure 20**. Because of the non-linear activation functions used to model the neurons, networks can contain a complex non-linearity which contribute to the generality of NN. A neural network can be considered as a general mapping from a point in its input space to a point in its output space, i.e. as a very
general multidimensional function. So far, we have only mentioned the adaptability neural networks. This property allows us to consider learning as a particular property of the network. Since the network represents a complex, but well defined mapping from input to output, the response is determined completely by the network structure and the input. Experience indicates that the network is robust against noise in the input, i.e. even if there are errors in some of the input elements, the network may produce the correct response. Because of the parallel, distributed architecture, large network models can be implemented in large computer environments including parallel computers. Even though the human neuron cells are much more complex than the simple models used for constructing artificial neural networks, the study of the behavior of computerized neural networks can extend our understanding about the functioning of human neural networks.

**Exercises**

a. In the section about single-layer networks and linear separability, a network was described with 2 real value variables, a threshold function which gave an output value 0 if the sum of the input functions was negative and 1 if the sum was non-negative. Draw an input variable diagram similar to Figure 15 with a boundary line dividing the input variable space in 2 areas corresponding to the two classes.

b. Construct a neural network corresponding to the binary adding machine in Figure 19.

c. **Black box** is an object the behavior of which can only be observed and analyzed by means of its input and output values. Neural networks are frequently characterized as black boxes although they are constructed from very simple neurons. Discuss the justification of this characteristic of NN.

d. Read Chapter 1: Computer Intelligence, in Lawrence.

e. Read Chapter 6: Neural Network Theory, in Lawrence.

f. Read Chapter 9: Brains, Learning and Thought, in Lawrence.
Session 2: Feed-forward networks

Two types of network

We start this session by introducing two fundamentally different kinds of network (Lippman 1987):

- Feed-forward networks
- Recurrent networks

In feed-forward networks (Figure 1), the stimuli move only in one direction, from the input sources through the network to the output neurons. No neuron is affected directly or indirectly by its own output. This is the type of network we shall study in this course. If all input sources are connected to all output neurons, the network is called a fully connected (Reed and Marks). A feed-forward network becomes inactive when the effects of the inputs have been processed by the output neurons.

In recursive network (Figure 2), neurons may feed their output back to themselves directly or through other neurons. We have already seen one example of this type of network in the previous session. Recursive networks can be very usefully in special applications. Because of the feedback structure in recursive networks, the network can be active after the first effects of the inputs have been processed by the output neurons.
Figure 2: Recursive NN

Learning

In the previous session, we learned that networks may classify input patterns correctly if their weights are adequately specified. How can we determine the values of the weights? One of the most important properties associated with neural networks is their ability to learn from or adept to examples. The concept of learning is closely related to the concept of memory (state of the system). Without memory, we have no place to preserve what we have learned, and without the ability to learn, we have little use of memory.

We start by a few considerations about memory and learning (Figure 3). In feed-forward neural networks, the function $y(t+1) = F(x_1(t), x_2(t), x_3(t-1), x_4(t-1), ...)$ describes the output at time $t+1$ as a function of the inputs at time $t$ and the previous time steps. A recursive net has feedback and will continue to feed itself after an initial stimulus.

Figure 3: An important difference between the human brain and NN

Memory and learning

Little known about how information is represented in human memory. Short-term memory is assumed to be dynamic, correspond to the feedback chains in neural networks. Long-term memory is assumed to be static and corresponds to the weights of the connections between neurons.

Even less is known about how these representations are created, i.e. the biophysics of learning. Experts seem to agree that the brain cannot reverse the direction of the pulses through the network. As we shall soon see, this is an important difference between the natural and some of the artificial networks.
networks, the weights represent the memory. NN learn by adjusting the weights of the connections between their neurons. The learning can either be supervised or unsupervised (Figure 4). We shall mainly concentrate on supervised learning. For supervised learning,

**Types of learning**

Supervised learning is the most simple model of the learning process. In supervised learning, the network is required to compute a response (“guess”) to each input, and compare it with the target answer. If the computed answer differs significantly from the target, the weights of the network are adapted according to a learning rule.

Unsupervised and enforced learning are other types of learning used by neural networks.

At this stage we shall concentrate on the supervised learning and return to other types of learning later in the course.

**Figure 4: Types of learning algorithms**

Examples of problems and their associated solutions are used. The weights of the network are initially assigned small, random values. When the problem of the first training example is used as an input, the network will use the random weights to produce a predicted solution. This predicted solution is compared with the target solution of the example and the difference is used to make adjustments of the weights according to a training/learning rule. This process is repeated for all available examples in the training set. Then all examples of the training set are repeatedly fed to the network and the adjustment repeated. If the learning process is successful, the network predicts solutions to the example problems within a preset accuracy tolerance for solutions.

**Learning rule and algorithm**

A learning rule prescribes how the weights of a neural network are adapted by stimulations from the network environment:

\[ w_i(t+1) = w_i(t) + \Delta w_i(t) \]

where \( t \) indicates the current weights, and \( t+1 \) the adapted weights.

The way the change \( \Delta w_i(t) \) is computed is described by the learning algorithm.

**NB:** In this course the first subscript denotes the source and the second the destination.

**Figure 5: Learning model**
Adjusting the weights is done according to a learning rule (Figure 5). The learning rule specifies how the weights of the network should be adjusted based on the deviations between predicted and target solutions for the training examples. The formula shows how the weight from unit $i$ to unit $j$ is updated as a function of delta $w$. Delta $w$ is computed according to the learning algorithm used. The first learning algorithm we shall study is the Perceptron learning algorithm Rosenblatt used (Figure 6). His learning algorithm learns from training examples with continuous or binary input variables and a binary output variable. If we study the formula carefully, we see a constant, $\eta$, which is the learning rate. The learning rate determines how big changes should be done in adjusting the weights. Experience has indicated that a learning rate $<1$ is usually a good choice.

The learning algorithm of Rosenblatt assumes a threshold activation function. The first task is to classify a set of inputs into 2 categories. The border between the 2 categories must be linearly separable, which means that it is possible to draw a linear line or plane separating the 2 categories of input points. If we, as Rosenblatt, (Figure 6), for example have 2 input sources or variables, the 2 categories of input points can be separated by a straight line. It is possible to prove that by adjusting the weights by repeated readings of the training examples, the border line can be positioned correctly (Figure 7).
Converging Perceptron learning

Rosenblatt proved:

If the points in the input space are linearly separable in 2 classes, the Perceptron learning rule converges the position of the decision hyperplane between the points of the 2 classes.

Figure 7: Converging condition for Perceptron

At the time Rosenblatt designed his Perceptron, Widrow and Hoff created another learning algorithm. They called it the Delta Algorithm for the Adaptive Linear Element, ADALINE (Figure 8). In contrast to Perceptron, ADALINE used a linear or sigmoid activation function, and the output was a continuous variable. It can be proved that the ADELrNE algorithm minimizes the mean square difference between predicted and target outputs. The ADALINE training is closely related to estimating the coefficients of a linear regression equation.

Delta algorithm

Widrow-Hoff created the Adaptive Linear Element (ADALINE) which has a neuron with a linear or sigmoid transfer function.

Delta rule, also called the Widrow-Hoff or the Least Mean Squared (LMS) rule, is similar to the Perception rule but assumes continuous valued output variables and linear or sigmoid transfer functions.

The Delta rule minimizes the mean square error (MSE) between the computed and the target output values.

Figure 8: The Delta algorithm
Non-linearly separable classes and multi-layer networks

We learned above that single-layer networks can classify correctly linearly separated categories of input patterns. However, the category boundaries are frequently much more complex. Let us consider the same input variables, $x_1$ and $x_2$, assume that the input space is divided into two categories by a non-linear curve as illustrated in Figure 9. It is not possible to construct a single-

![Non-linearly separable regions](image)

**Figure 9: Non-linear regions**

layer network which classify all possible input points correctly into category $A$ or $B$. A well known problem which cannot be solved by single-layer networks is the Exclusive Or XOR problem. It has only 2 input variables, $x_1$ and $x_2$, both binary. The complete input space consists of 4 input points, $(0,0)$, $(0,1)$, $(1,0)$ and $(1,1)$. Define category $A$ as composed of the inputs with an uneven number of 1's, i.e. $(0,1)$ and $(1,0)$, and category $B$ of the inputs with an even number of 1's, i.e. $(0,0)$ and $(1,1)$ (Figure 10). In the XOR problem, one of the categories consists of two

![The XOR problem](image)

XOR has a Truth table:

<table>
<thead>
<tr>
<th>$x_1$</th>
<th>0</th>
<th>0</th>
<th>1</th>
<th>1</th>
</tr>
</thead>
<tbody>
<tr>
<td>$x_2$</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>$y$</td>
<td>B</td>
<td>A</td>
<td>A</td>
<td>B</td>
</tr>
</tbody>
</table>

The XOR is the classical example of a problem with decision regions which cannot be separated by a line.
Figure 10: The XOR problem

separated areas around the 2 members of the set of input points, while the other category consists of the remaining input space. Problems which cannot be considered as linearly separable classification problems were discussed extensively by Minsky and Papert in their famous book in 1969.

Multi-layer networks

XOR and similar problems can be solved by means of multi-layer networks with 2 layers of neurons (Figure 11). If the network is considered from outside, only the input points sent to the

![Multi-layer networks](image)

Figure 11: Multi-layer networks

network and the output values received from the output neurons can be observed. The layers of neurons between inputs and outputs is therefore called the hidden layers of neurons (Figure 12).

Hidden layers

The hidden layers are called ‘hidden’ because they are composed by neurons which cannot be observed outside the net. The outputs of the hidden neurons are always input to other neurons.

There is one or more hidden layers in a multi-layer feed-forward network.

The hidden neurons make a network able to classify data in non-linearly separable regions.
Multi-layer networks, MLN, also often referred to as the Multi-layer Perceptrons, MLP, have 1 or more hidden layers. Each layer can have a different number of neurons. A feed-forward MLN, in which each neuron in a previous layer is connected to all neurons in the next layer, is a fully connected network. Network will have different properties depending on the number of layers and their number of neurons.

**Backpropagation learning**

It is possible by trial and error to construct a multi-layer network which can solve the for example the XOR problem. To be a useful tool, however, a multi-layer network must have an associated training algorithm which can train the network to solve problems which are not linearly separable. Such an algorithm was outlined in the early 1970's in a Ph.D. thesis by Werbos. The implications of his ideas were not recognized before the algorithm was re-invented about 10 years later and named the backpropagation algorithm. It was made famous from the books by Rumelhart, McClelland and the PDP Research Group. (Figure 13).

---

**Figure 12: Hidden layers in multi-layer networks**

---

**Figure 13: Werbos and his proposal**

The backpropagation algorithm can be regarded as a generalization of the Delta Rule for single-layer networks. It can be summarized in 3 steps as indicated in Figure 14. The algorithm should be carefully studied with particular focus on the subscripts! If you do not manage to get the full and complete understanding, don't get frustrated: the training programs will do the job. The original algorithm has been modified and elaborated in a number of versions, but the basic principle behind the algorithms is the same.
The BP algorithm consists of a 1) forward computation of the output of all neurons and the output differences of target and computed output y followed by a 2) backwards computation of the derivatives:

a. if \( j \) is an output neuron:
\[
\delta_j = (1 - y_j)^2(y_j - \hat{y}_j)
\]
b. or if \( j \) is a hidden neuron:
\[
\delta_j = x'_j(1 - x'_j)\sum_k \delta_k w_{jk}(n)
\]
and finally, 3) computation of the weight adjustment:
\[
w_{ij}(n+1) = w_{ij}(n) + \eta \delta_i x_j
\]

**Figure 14: The backpropagation algorithm**

It is important to note that the neural network type we discuss is feed-forward networks, while a backwards propagation or errors is used for training the network.

**Measuring learning**

Given a training set of examples with tasks and corresponding target solutions, we need to know how well a network can learn to reproduce the training set. There are many ways to measure the success of learning. We adopt the principle to indicate learning success as a function of how well the network after training is able to reproduce the target solutions of the training examples given the tasks as inputs. We use the metric Mean square error, MSE, or the Root mean square error, RMSE, to express how well the trained network can reproduce the target solutions. Because the differences between target values and output values are squared, positive and negative errors cannot eliminate each other. In **Figure 15**, the MSE is defined for a single output variable. MSE for several output variables can be computed as the average of the MSE's for the individual output variables.
Mean square error

How well a network is trained to recognize patterns can be expressed by the mean square error:

$$\text{MSE} = \frac{1}{N} \sum (y - d)^2$$

where $y$ denotes the computed value and $d$ the target value for each of the $N$ output neurons.

Figure 15: The MSE metric

Training a network is an iterative process. The training set of examples is run through the network repetitively and for each run a new MSE measurement is made. We can compute an MSE error curve as a function of the number of training runs, and we want this curve to be falling as fast as possible to a minimum. We obviously want a training algorithm which adapts the weights in such a way that the value of the MSE is decreasing to a minimum (Figure 16).

Figure 16: The error surface and error minima

Unfortunately as indicated in the figure, when moving around in the space of weights, there may be a number of local minima for the error function. Training methods, which follow the steepest decent on the error surface down to the minimum, are called steepest gradient decent methods. Backpropagation is a steepest gradient decent method (Figure 17). When the adjustment has
Steepest gradient decent

If a downhill skier is surprised by fog and frost, he may want to get down to the bottom of the valley as fast as possible, and try to go where the hill seem to be steepest.

The method of the steepest gradient decent uses the partial derivatives of the MSE function to approach its minimum in the shortest way. To be able to use the derivatives, the transfer functions must have derivatives.

The steepest gradient decent principle is not the most efficient in every case, but is used in many learning algorithms because its simplicity. It is used in the standard backpropagation algorithm.

**Figure 17: The principle of the steepest gradient decent**

lead to a point in the weight space which is a local minimum, other methods must be applied to see if this is a local minimum or a global minimum.

**Generalization**

General experience indicates that a network, which has learned the training examples effectively (found a minimum on the error surface), is not always a network which is able to solve other problems from the same population or domain as well. They may not be capable to generalize from the training examples to problems they have not been trained on. There can be several reasons for inability to generalize. For example, the tasks in the domain can be very heterogeneous and too few examples are available for training, the examples used as training set are unrepresentative, etc. The situation may be improved by drawing a more representative and bigger sample of examples. Since both the tasks and the target solutions are required, this can be expensive.

Another reason can be over fitting. Over fitting occurs when a network is trained too much and has learned to reproduce the solutions of the examples perfectly, but are unable to generalize, i.e. the training examples have been memorized too well. Intensive training can reduce MSE to a minimum at the same time as the network's ability to generalize decreases. Methods to stop training at an optimal point are required.

One simple approach is to divide the set of available examples with problems and target solutions randomly into 2 sets, one training set and one test set. The examples of the training set are used only for training. The test set can be used for continuous testing of the network during training. Another MSE curve is computed based on the application of the network on the test examples. When the MSE curve for the test set is at its minimum, the best point to stop training is identified even if the MSE curve for the training set continues to fall. If the training and test sets are representative samples of problems from the application universe, this procedure gives the approximately best point to stop training network even though the MSE for the training
examples is still decreasing. More sophisticated approaches based on jack-knife methods, can be used when the number of available examples is small.

Classification revisited

We have seen that the XOR problem cannot be solved by a single-layer network. Figure 18 indicates that a two-layer network can solve classification problems for which the category boundaries in the input variable space are disconnected. Three-layer networks can classify input patterns in arbitrary specified regions in the input variable space. These networks can also be trained by the backpropagation algorithm.

The XOR problem can be illustrated in relation to networks with different number of layers (Figure 19). The figure demonstrates that at least a two-layer network (\(l\) hidden layer) is needed for solving the XOR problem. We shall design and train such a network later in the course.

Most of the problems we encounter can be solved by single-, two- or three-layer networks. In very special cases they may be handled better with networks with more hidden layers.

<table>
<thead>
<tr>
<th>Types of decision regions</th>
</tr>
</thead>
<tbody>
<tr>
<td>Networks with no hidden layer can form linear decision regions in the space spanned by the input variables.</td>
</tr>
<tr>
<td>Networks with one hidden layer can form convex decision regions in the space spanned by the input variables.</td>
</tr>
<tr>
<td>Networks with two hidden layers can form arbitrary decision regions in the space spanned by the input variables.</td>
</tr>
<tr>
<td>Networks with three or more layers are not frequently applied.</td>
</tr>
</tbody>
</table>

Figure 18: Decision regions
Exercises

a. Consider a set of married couples. Their marriage histories have been recorded, each individual has either been previously been married or not. A social researcher want to investigate if 'equal' background is an advantage and wants to classify the couples into two groups: 1) the couples who have an equal experience, i.e. both were previously unmarried or both had a previous marriage experience, 2) the couples with unequal experience. Is it possible to train a single layer neural network (without hidden layers) to classify couples into these groups?

b. The Mean Square Error (MSE) is used as a metric to express the performance of a network. Alternatively, the sum of the absolute errors can also be used. What do you feel is the advantage/disadvantage of MSE?

c. Read Chapter 2: Computing Methods for Simulating Intelligence, in Lawrence.

d. Read Chapter 8: Popular Feed Forward Models, in Lawrence.
Session 3: BrainMaker software

Software

In the last decade many implementations of the backpropagation algorithms have been introduced. There exist stand-alone programs as well as programs included as a part of larger program packages (SPSS, SAS, etc). There are commercial programs which can be purchased and freeware programs which can be downloaded from program providers on the net.

In this course, we use software from California Scientific Software, CSS (Figure 1). Information

Brainmaker software

The two main components in the Brainmaker software from California Scientific Software are:

• NetMaker, which is a pre-processing tool, and
• BrainMaker, which is the tool for training, testing and executing neural networks. This program also performs certain post-processing tasks.

Both programs are easily installed and have a number of useful features. As a start only a limited number of functions need to be used. We shall use the simple XOR problem we have discussed previously as an introductory example.

Figure 1: Software

about the CSS is included in the section Software. The software package consists of several independent programs. We use 2 of the programs,

• NetMaker
• BrainMaker

Note that the Student version of BrainMaker has limitations as to the size of the network which can be handled, and functional capabilities compared with the Standard and Professional versions. If larger networks should be processed, the Standard or the Professional version of BrainMaker is recommended.

The software for Windows 95, Windows 98, Windows NT 4.0 and Windows 2000, is compact and distributed on a single floppy diskette. A set of application examples are also included on the distribution diskette. A user should have few, if any, problems installing and using the software. A manual for the programs comes with the software. In the manual, 3 of the applications on the distribution diskette are discussed in detail. These applications can serve as models for specification of network training. Finally, the software package includes an introductory text book, which gives a wider perspective on neural networks.
NetMaker is a preprocessing program which processes ASCII data files to the form required by BrainMaker. BrainMaker is a flexible neural network program which trains, tests and runs data files and also includes some useful analytical features.

You can install the software where you prefer. To make things as simple as possible, we assume that the files are installed as recommended in a folder named c:\BrainMaker. During the course, and particularly when you study this session, you should have the BrainMaker software open running in the background. You can then switch from the session to the programs to look into the different features and back again to this text.

NetMaker

You will find details about NetMaker in the manual, Chapters 3 and 9. Note that NetMaker is not a tool for preparing data files, but for adjusting already prepared data files. Preparation of data files can be done by a number of text programs, as for example NotePad, or by some simple spreadsheet programs such as EXCEL 3.0. Note that the more advanced spreadsheet programs as EXCEL 2000 etc. producing application books and folders are not suited for the preparation of data files for NetMaker. EXCEL 2000 can, however, Save As an EXCEL 3.0 page with the extension .xls which is acceptable for NetMaker.

Double clicking the NetMaker program icon or name will display the main menu with:

- Read in Data File
- Manipulate Data
- Create BrainMaker File
- Go to BrainMaker
- Save NetMaker File
- Exit NetMaker

Selecting Read in Data File is the obvious start. NetMaker can read data files with .dat and .txt, extension, Binary, BrainMaker and Statistics files. As already mentioned the options also include EXCEL files with certain limitation.

Note that some of the files you will want to work with are .txt files, but has other extensions. Example are the statistics files from training and testing which have the extensions .sts and .sta. NetMaker is sometimes unable to recognize these as text files, and you must specify the option Text in the menu Type of file before you open these files.

The data file read is displayed with one column for each variable and one row for each example. The main toolbar contains:

- File
- Column
- Row
The next 2 rows in the table heading refer to the type of variable and to its name in the respective columns. Note that by first clicking on the column name in the second row, we can go to the Label in the main toolbar and mark the variable type, for example Input, Pattern or Not Used, and to rename the variable if you so wish.

Save NetMaker File converts a usual .txt file to a NetMaker .dat file. We shall return later to the other alternatives.

The XOR problem will be used as an example of how to use the programs. We start preparing the problem examples. Type the 4 possible XOR training input points by means of Notepad, EXCEL or any ASCII text processing program as indicated in Figure 2. The result should be like shown in Figure 3. When you have typed in this, save it as a text file and call the file myXOR.txt to distinguish it from the illustration XOR files in the section Datafiles.
Figure 4: XOR as a Notepad file

This text file can be read by NetMaker from the File menu and will be displayed as in Figure 4.

Figure 4: Netmaker’s presentation of the XOR file

Now we can manipulate the data by the options offered by the NetMaker program. If you have not done so, the most important specification is to assign the variables to input or pattern (remember that pattern means output in BrainMaker terminology). There are many options in the toolbar menus as we see in Figure 5 and Figure 6. You will also find the files by clicking Datatypes in the window to the left. The list contains all the files we discuss.
Advanced NetMaker Features

Read the BrainMaker manual, Chapter 3 and 9, about the more Advanced features of NetMaker. Experiment with transferring data from the XOR.txt or use one of the data files in the BrainMaker folder.

Most important is NetMaker’s ability to create correct BrainMaker files including the network definition file and a random subsamples for independent testing of the trained network.

Another feature useful in case of a time series application is the possibility to move a column one or more periods (rows) up or down creating ”lag” or ”lead” variables.

Figure 5: More NetMaker features

Figure 6: NetMaker’s feature for exploring correlations

NetMaker permits exploration of correlation between pairs of variables. By copying the target column of the XOR.dat, we get the following graph indicating perfect correlation.

You can download the files to your computer by

- Open a File/New File in Notepad
- Edit/Copy the wanted file in Datafiles to your Clipboard
- Edit/Paste the file into the opened file
- Save the file with a name by File/Save As
The trained networks may be slightly different from those displayed in the figures because they are based on another initial set of weights and with a few variations to demonstrate the some additional possibilities.

Usually it will be required to divide the data file into training and testing files. NetMaker has the option File/Preferences by which you can specify how you want the data file randomly divided between the two files. In the case of the XOR problem, training and test files are identical and no division is needed. The mark in File/Preferences/Create Test File must therefore be deleted.

In File/Preferences there are several other options. The last row is Network Display with 2 options, Numbers or Thermometers. During training, the first gives a continuous display of the calculated variable values in digital form while the second in a graphical form. With less powerful computers, it was interesting to follow the development. However, with high speed computers, the figures change too fast to give any information. Default is Thermometers. I suggest that you try to use Numbers which is a less disturbing alternative. It is also possible to turn the display off in BrainMaker.

When data and specifications are ready, the material must be converted to the format required by the BrainMaker program. The conversion option is found in NetMaker’s File/Create BrainMaker Files. Since we usually specify the variable types for File/Read Data, we can usually select options Write Files Now. Your XOR problem is converted to a definition file, myXOR.def and a training file, myXOR.fct (Figure 7). In most application, there will also be a test file. The test file has the extension .tst. All files can have different names. The default is to give the BrainMaker files the same name as the NetMaker .dat file. Use this convention in this course.

![BrainMaker XOR.def file](image)

This is not the complete .def produced by NetMaker. It has been edited to contain the minimum definitions needed for BM to train for solving the XOR problem and to test the trained network.

The definition file may be edited either by means of Notepad or the menus of BrainMaker.

**Figure 7: BrainMaker’s definition file for the XOR problem**

In the main toolbar, there are many possibilities for manipulating the data files. Row/Shuffle Rows is important. In many NetMaker data files there may be embedded trends, small units may be in the beginning of the file, large at the end, and so on. To obtain good training conditions, the
data should be well shuffled. Just before creation of BrainMaker files, it can be a good idea to shuffle the data rows several times. Note that in a few applications, it is important to maintain the initial order.

Another important preparation is the option Symbol/Split Column into Symbols. The term Symbols is equivalent to Binary variable names. If you have a categorical (coded) variable, say a disease diagnosis with 10 alternative codes, the codes in the column must be converted to 10 separate, named binary variables. Mark the column and click on this option. The option requires that you specify how many categories exist and their names (NetMaker will give them default names in case you do not specify your own). The expansion to binary variable is handled by NetMaker when the training and testing files are created for BrainMaker.

The last NetMaker option we consider is Operate/Graph Column. This option offers a convenient way to visualize the content of a column. BrainMaker will produce statistics for instance after each training iteration. It is frequently required to study the progress of the results to identify the best point to stop the learning. Inspection of a graph can indicate the point we are looking for.

**BrainMaker**

You will find the details of the BrainMaker program in Chapters 3, 10, 11 and 12 of the manual. When opened, BrainMaker displays a rather empty interface with only one option, File, in the toolbar. In this, we find File/Read Network File. This option presents the .def and .net files of the folder c:\BrainMaker\. You will look for a file of the first kind when you start a training task. Training generates one or several .net files which you can use to continue training, to test or run a trained network. BrainMaker accepts only these 2 types of files as specification for training, testing and operation.

The definition file is a text file which can be opened by any text program as NotePad etc. It starts by specifying the layout of the problem example. A definition file for the XOR problem is displayed in Figure 7. The first line specifies that for each problem in the training file, input is on 1 line and consists of 2 elements while target output is on a separate line and consists of 1 single element. The last line in the layout specifies one hidden layer by the number of neurons. If more hidden layers, each is specified by the number of neurons it contains. In our case, there is 1 hidden layer with 2 neurons.

The definition file for the XOR problem as produced by NetMaker is more extensive than the one in Figure 7. The definition file illustrated in the figure has been edited to show a simpler version. The definition file can be read and edited by Notepad according to your needs and the rules given in the manual. Take a look at the XOR.def in Datafiles which contains a third version of the definition file for the XOR - problem.

From Figure 7 you can see that there are 3 initial specifications required:
input must be followed by the type of input used, i.e. if the input is picture, number or symbol. In the XOR application, we use number. Then the number of lines and elements per line follow. For each example, we have 1 line with 2 elements (the x and y variables). The specification of output is similar. In our XOR illustration, 1 line with 1 number output is specified.

Each hidden layer is specified by the number of neurons contained in the layer. If not specified, a default specification is used.

The files used for training and eventually testing must be specified, filename trainfacts and filename testfacts are the keywords required. Then the definitions of several parameters follow, the most important are:

- learnrate
- traintol
- testtol

The parameters are set to default values if not specified.

The scale minimum and scale maximum for input and output are identified by NetMaker. They inform BrainMaker about the minimum and maximum values for the individual variables. They are used for normalizing all facts to internal values to between 0 and 1 for computations in BrainMaker. This eliminates dominance of variables with large variation ranges.

The specifications can also be changed and modified by the BrainMaker menus, but these changes may not be saved. BrainMaker has a main toolbar with the options:

- File
- Edit
- Operate
- Parameters
- Connections
- Display
- Analyze

These give a high degree of flexibility for use of the program. The most important options are discussed below, but you are encouraged to experiment and get your own experience.

The File in the toolbar includes:

- Read Network
- Save network
- Select Fact Files
The 2 first are obvious and need no comments. *File/Select Fact Files* permits file specifications and can override the specifications written by NetMaker in the definition file (Figure 8).

---

During training after each run (iteration), BrainMaker can generate statistics such as number of good predictions, average error, root mean square error, correlation between predicted and target values etc. If *File/Training Statistics* is selected, the statistics are computed and saved in a file with a `.sts` extension. When a test run is specified, similar statistics can be produced and saved in another file with extension `.sta`. The default names for the statistics files are the same as the fact file name, and they are distinguished by the extension.

The option *File/Write Facts to File* offers a possibility for each example record to write the input variable values, the target variable value(s) and the predicted output variable value(s) to a file with extension `.out`. This file is required when network generalization should be evaluated.

We can postpone the main toolbar option *Edit* to some later time and continue with the *Parameters*. The following options are used frequently:

- *Learning Setup*
- *Training Control Flow*
- *New Neuron Functions*

The possibilities in *Parameters/Learning setup* are many (Figure 9). From the previous session we remember that the aim of learning is to identify the weight point associated with the minimums of the error curve or surface. If changes in weights are too large, there is a risk that the
File 9: Learning setup

minimum may be passed undetected. It is a general experience that a learning rate which changes according to the learning progress is a better choice than a constant learning rate. **Linear learning rate tuning** is often very effective. This tuning is based on an initial learning rate, for example 0.5, used in the first stage of learning. As the network becomes more trained, the learning rate is proportionally reduced to a specified minimum rate. **Automatic Heuristic Learning Rate** is another interesting and useful algorithm according to which BrainMaker will automatically reduce the learning rate if the learning progress becomes unstable. Use the default constant learning rate set to 1 in the XOR application.

The next selection is the Parameters/Training Control Flow (Figure 10). This menu gives
another set of specification possibilities. The specification of *Tolerances* gives the option to decide how accurate the network computations must be to be considered 'correct'. A tolerance set to 0.1 means that the absolute difference between the computer output and the target value for any variable must be equal or less than 10% of the target value to be considered correct. Since we are considering output values either 0 or 1 in the XOR case, the training tolerance can be increased to 0.4. In applications with continuous output variables, it may often be necessary to reduce default test tolerance from 0.4 to 0.1.

The *Parameters/Training Control Flow* also offers the user control to stop the training process subject to different conditions. Default is that training should continue until the network is able to reproduce all outputs within the tolerances specified. Make you acquainted with the other stopping options. For the XOR application accept the default condition, *All Training Facts are Good*.

The last training control flow option in this menu is *Testing While Training*. This is a very powerful strategy which we have already discussed in the previous session. It permits us to localize the best point to stop the training to avoid over fitting. By turning this and the *File/Testing Statistics* options on, the network applied on the test file can be saved after each iteration. If the option *Save after every run* has also been turned on, we can return to the network version just before the best stop point, and train this network the necessary number of iterations to the best stop point. After a sufficient number of training runs, the training is stopped and the test RMSE inspected. Usually the number of training iterations needed to obtain the best network can be identified. For the XOR problem, we do not need the testing since the training and testing sets are the same, and we leave the marking squares blank.

In *Figure 11*, *Parameters/New Neuron Functions* to determine the activation functions to be used is shown. The sigmoid function is default, but it is easy to change to another activation function. For the XOR task, we choose the **sigmoid** activation function for the neurons in the hidden layer. This activation function could also have been used for the output neuron, the computed value of which then could have been interpreted as an estimate of the conditional probability for an input to belong to the category with unequal input values. A low probability therefore would indicate equal (0, 0 or 1, 1) input variable values. To demonstrate the possibility of using mixed activity functions, we choose a **step** function for the output neuron. This function will output either 0 or 1 representing the 2 categories of inputs.
Figure 11: Changing number of layers

The next toolbar option is Connections/Change Network Size which permits us to change the number of layers and neurons in each layer (Figure 12). Check that the menu display 2 inputs, 1 hidden layer with 2 neurons and 1 output neuron. The menu also summarizes the number of connection (weights) in the network. You may notice that there in addition to the 4 possible connections between the 2 input sources and the 2 hidden neurons are 2 more. These are connections to each of the 2 hidden neurons from a threshold input source which always emits 1's. For the same reason there are 3 connections to the output neuron, 2 from the respective neurons in the hidden layer and 1 from a threshold input source. More about the threshold input sources important for effective learning will be discussed in the next sessions.

Figure 12: Changing the size of the network
The option *Display* in the toolbar permits us to follow the training as it progresses. In the menu we check that *Enable Display*, *Network Progress Display*, *Display Parameters* and *Display Statistics* are all marked. *Network Progress Display* will give as continuous picture of the training progress expressed in a RMSE graph, while the 2 other displays give numeric information about the network parameters and continuously updated statistics for the training process.

The last item in the toolbar is *Analyze* which gives options for analyzing a trained network.

**Training and testing**

You are now prepared to start the training of a network based on your *myXOR* files. Go to toolbar option *Operate* and select *Training*. The training starts with an *initial set* of small random weights. Because they are random, the training can develop different each time the program is started. This is important to note. You will not always get as good results as your fellow students (but sometimes better!).

The training progress can be observed on the computer display (*Figure 13*). BrainMaker was set to stop training when it had learned to predict the output values. The training was in the run illustrated stopped after 72 iterations when all predictions were within the set learning tolerance. NetMaker can display the graph in a nicer form as can be seen in *Figure 14*. Remember to save the trained network. It will receive the network name with the extension ".net".

*Figure 13: Training for the XOR*

You can see the specifications by reading XOR.net in Datafiles by NotePad. Reading this file into BrainMaker will show that the training required 88 runs this time.
Figure 14: RMS graphs

You should now have a network trained to solve the XOR problem. Since the training set covers all possible XOR problems, it is unnecessary to test the trained network. Formally, we can, however, test the trained network by copying the training file and give the copy the extension .tst. Click on File/Select Files and mark Read Testing File From and type in the name of the test file, i.e. myXOR.tst (Figure 15). Specify also File/Write Facts to File and you will get the input

Figure 15: Testing the trained network

If you have an independent test file, the generalisation ability of the trained network can be tested. In the XOR case all possible input combinations have been used. We therefore use a copy XOR.tst of the training file as test file.

To be able to view each individual output we specify the option to Write facts to the output file named XOR.out. No changes in default settings are needed.

Figure 15: Testing the trained network

and the predicted output for each of the records in the test file. Testing is done by selecting Operate/Test.
Evaluation

Results from the XOR exercise can be studied by means of NetMaker (Figure 16). From the figure we can see that the network solve the 4 possible input problems perfectly.

**Figure 16: The output file**

BrainMaker permits us also to study the weight matrices as shown in Figure 17. When a network diagram is prepared with weights assigned to the connections, it is possible to visually study how the network handles the XOR problems (Figure 18). The 2 threshold neurons can be identified.
Figure 18: The XOR solution

One is at the top of the figure emitting 1’s to the 2 hidden neurons. The second is at the bottom of the network and emits 1’s to the output neuron. The trained weights associated with these neurons correspond to the thresholds of the receiving neurons. It can also be shown that these thresholds play the same role as the constant term in a regression function.

There are many more features of BrainMaker not included in our discussion of the XOR example. BrainMaker uses for example extreme values to transform the actual values with wide ranges to internal values between -1 and 1 during processing. After training, testing or operational runs are finished, the resulting output values are transformed back to their actual

Figure 19: Scaling

NetMaker identifies the minimum and maximum of each input and output, and specifies these in the definition file. The minimum and maximum series can also be derived by BrainMaker if the fact and definition files are constructed manually.

The min-max specifications are used to scale the facts in such a way that the variables are all varying within the same range. By scaling the facts, the range of the weights will also be normalized.

If target variables are scaled, the output variables must be re-scaled before use. This is done automatically by BrainMaker.
variable range (**Figure 19**).

A summary of additional features of BrainMaker are listed in (**Figure 20**). Some of these features will be discussed and used during the course.

---

**Advanced BrainMaker feature**

Here are a few important features not used in the XOR-example:

1. Adjusting learning rate.
2. Adjusting smoothing factors.
3. Reduced adjustment of heavy weights.
4. Modify size of network while training.
5. Prune a neuron from a hidden layer and continue training.
6. Testing when training.
7. Setting of stop rule.
8. Prune small connections.

---

**Figure 20: More BrainMaker features**

**Exercises**

a. Before you start on the next session, **install** the software and make yourself acquainted with at least the 2 programs we use in this course. Since you will be using them frequently, it may be a good idea to create **Shortcuts** for the programs handy on the desktop. Do not be afraid of experimenting with the programs and the data files. You can always return to the original files by downloading the files from the distribution diskette. You can also click on Datafiles in the window to the left, select and click the files you want to see from the list.

b. **Start up** NetMaker and click on **File/Read in Data File**. You will get the content of the BrainMaker folder in response. Select the data file **Widgets.dat**. You will get a NetMaker table with 30 data rows and 12 columns in return. The data have been prepared for conversion to Brainmaker format. The last column is obviously the target/output/pattern variable. Inspect the **different options** in the toolbar, but do not make any changes. Finally, you can click **File/Create Brainmaker File** and answer **Write File Now**. You will get a warning that the files exist. You may cancel the process, but **no damage** is done if you respond **Overwrite**.

c. NetMaker produces 3 files, **Widget.def**, **Widget.fct** and **Widget.tst**, which are the **definition file**, the **training file** and the **test file**, respectively. Start **NotePad** or another ASCII reader, and **open** c:\BrainMaker\ **Widget.def**. It shows the form of the network definitions. Make a copy and save it under another name, for example **W.def**. Try to make different **changes** in **Widget.def** in Notepad. You may delete all dictionaries, names etc. until you are left with input output, hidden
layer, filenames, and minimum and maximum specifications. Save the file after your modifications.

d. Start Brainmaker and click File/Read Network to load your modified Widget.def file (or W.def). Click Operate/Train Network. Did it train? How many training runs were required to learn the examples of the training file? I got 172. Compare with your colleagues' results. If you load the definition file once more without saving the trained network, and then train again you may get another number. I got 117. As already explained, the weight matrices are initiated with small random number which give the network a new starting point each time it starts.

e. Study Chapter 1, 2 and 3 in the BrainMaker User's Guide and Reference Manual.

f. Read Chapter 10: Neural Network Design Process, in Lawrence.

g. Read Chapter 11: Data Preparation, in Lawrence.
Session 4: Survey of applications

Classification and regression problems

The application domain for neural networks is extensive. Grouping similar applications in types helps to profit from previous experience when you are required to design new applications. It is usual to distinguish between 2 main types:

- **classification problems**
- **regression problems**

Figure 1 lists examples from the two application types.

A data classification task is characterized by a set of records which should be assigned to one of a set of predefined categories based on the content of the records. The content is a set of variable values. In some applications, there are only few categories, minimum 2, to which a large number of inputs should be assigned. We have already met and solved one classification task, the XOR problem. From the list in Figure 1, we recognize other similar classification problems. Examples

![Applications Table](image)

**Figure 1: Examples of NN applications**

frequently used are classification of a set of medical records with symptoms of illness in categories as records for serious and less serious cases, classification of a set of digitized voices representations into a category for male and an another for female voice representations.

A typical application is **quality control** in mass industrial production. The input is a set of recorded characteristics reflecting product quality for each produced item. Each item record must be assigned either to a bin of acceptable items or to a bin of rejected items because of bad quality. In real applications, the classifier, in our case the neural network, can be build into a real time system of 3 parts which the items moves through in sequence [Ashby 1971]. The first part is
the sensory component observing the items when they pass, the second part is the control component deciding which of 2 bins each item should be directed to, and the last part is the physical opening of the door to the decided bin when the item arrives to this component.

In other applications, the number of categories may be large. The extreme case is one class for each input, i.e. we require unique identification of each input. The identification of individuals by their fingerprints serves as an example for this kind of problem. Different kinds of problems are discussed in the statistical theory of classification [Duda and Hart 1973].

Formally, the classification problem can in our context be stated as indicated in Figure 2. In

![Classification problems](image)

\[
\text{Let:}
\begin{align*}
\mathbf{x} & \text{ be a vector of } m \text{ variables which may be integer or continuous valued, and} \\
\mathbf{y} & \text{ a vector of } n \text{ variables with values } 0 \text{ or } 1 \text{ constrained by condition } \sum y_i = 1. \\
\mathbf{w} & \text{ is a set of weights (parameters).}
\end{align*}
\]

Classification is application of a function (a set of rules):
\[
y = F(x; w)
\]

The curse of dimensionality: Possible points in the \(n\) dimensional space of \(x\) increase exponentially with number of variables. To design complete mapping rules or specify \(F\) will be prohibitive even for a moderate number of variables.

**Figure 2: Classification**

theory, the set of categories is represented by a binary, category vector with one element for each feasible category. The sum of the vector elements should therefore be 1. Each permissible vector has one and only one element with value 1 indicating the class to which the input should be assigned. All other elements have values 0. Each item to be classified has properties represented by another vector (corresponding to the inputs) which can comprise discrete as well as continuous variable. The rules of classification can be imagined as a mapping from each possible vector point to one and only one class vector.

Another formulation, which is more effective when using NN is that we search for an output vector with continuous variables values in the range 0 to 1. The variables are defined as the conditional probabilities that the associated categories \(i\) are the correct assignments given the input vector point [Bishop 1995].

In the previous session, the XOR problem was solved by training multi-layer neural networks. The XOR problem is a special case of a general problem, referred to as the parity problem (Figure 3). Increase the number of elements in the binary input vector from 2 to an arbitrary
number. The problem is to assign all input vectors containing an even number of 1's to class A, and all vectors containing an odd number of 1's to class B. It can be proved that a multi-layer networks can be trained to solve any parity problem.

Logical classification

Minsky and Papert proved that the parity problem:

\[
b + b + \ldots = \begin{cases} 
1 & \text{if number of 1's is odd} \\
0 & \text{if number of 1's is even}
\end{cases}
\]

Where b's are binary variables, known as XOR problem in the 2+2 problem, cannot be solved by a Perceptron.

Even though it is easy to design a simple counting algorithm which solves the problem, it requires a complex network with a layer of hidden neurons to train and solve the problem.

Figure: Regression

There are other more effective ways to solve the parity problem. The point is here to prove that multi-layer networks can be trained to solve complex classification problems.

Pattern recognition

The most known application domain for neural networks is probably pattern recognition (Figure 4). The pattern recognition applications vary from training a neural network to uniquely identify each individual in a set of photographic images, to training a net to classify individuals in a population by gender based on pictures. Humans have a fantastic ability to perform pattern recognition without being able to give a comprehensive explanation for the 'rules' they use. We have usually no problem to distinguish between pictures of a 'cat and a dog. But try to set down the rules you use for a rule-based computer system.
Pattern recognition

Pattern classification is a 'classical' NN application. The inputs are images and the purpose of the recognition is to identify patterns of important characteristics in the images. The first step will be to lay a grid over each image assuming that the content of each rectangle is homogenous as to color. A high resolution (number of rows, columns of the grid) will give a visually more attractive image to the human eye than a low resolution.

However, a lower resolution maintaining the characteristic pattern of the image may be more efficient for recognition purposes than a higher resolution which disturbs the process by unimportant details. The optimal resolution will depend both on the images as well as on the purpose of the recognition.

Figure 4: Pattern recognition

Another frequently investigated application is character recognition. Also in this field, the tasks vary from the very simple to the complex. A simple application is the recognition of decimal digits in a standard form, while the most challenging is the recognition of letters in handwritten messages. The approach to solving these tasks is to create an image for each character, divide each image into components by a grid. Each grid cell corresponds to a pixel of the image and is represented by an input variable. In the case of a black and white image, each pixel can be represented by a binary variable with only 2 values, 'white' or 'black'. If the image has colors, a categorical variable will be required for each pixel with as many codes as there are different colors. The whole area represented by the pixel is considered to have the same color. The resolution, the amount of detail or number of the pixels used in the application to represent the input character, is an important factor. High resolution means that details are preserved in the image, but it also means that the number of input variables is large and resource consuming, while a low resolution does not preserve as much information but is cheaper to process. It is important to find a good balance between the requirements of details and resources.

Much development is being done to communicate vocally with computer-based systems. To be able to do so, a component, which can convert analogues voice signals to a digital representation is needed (Figure 5). Simplified, each word has its own sound pattern. Neural networks have been trained to recognize a limited number of different words and used in different voice applications.
Figure 5: Voice recognition

A sound can be described by its frequency, amplitude, and duration. A human sound in speech (phoneme) can be described by a pattern of sound frequencies, amplitudes, and lengths. Each word has its general pattern even though it may vary according to how it is pronounced and by whom.

In a NN for recognizing phonemes, each phoneme pattern may be represented by a multiplex of the 3 sound variables and unique pattern identifier on the output side. An analog pattern of speech as illustrated, can easily be digitalized.

Figure 6: Music recognition

Related to the voice recognition is music recognition. Digitized music has become usual and conversion from analogue to digital form is unnecessary. Neural networks can be trained to recognize different features in music by certain composers, from different time periods and regions, and from different categories of music (Figure 6). These networks can for example be used to help identifying unknown pieces of music.

Figure 6: Music recognition

In its simplest form, a melody can also be represented by a sequence of notes, each represented by a few variables as its pitch, duration, etc. A sample of melodies can be represented as input vectors with the specific variable values for each melody. A NN can be trained to recognize the features of each melody or melodies of different composers. Each composer can be assigned separate target variable.

The NN may be trained to classify each melody by composer, and if it has been successfully trained, it may be used to assign unidentified melodies to a composer assuming he/she was represented in the training sample.

We have disregarded problems as varying length of melodies, etc.
Diagnostic tasks

Producing a diagnosis based on a set of symptoms is similar to a classification problem. Many medical applications of NN are associated with diagnostic applications (Figure 7). A generic

Medical diagnosis

The objective of the medical application in the tutorial of BrainMaker is to predict the length of stay.

Another application is diagnoses of diseases based on symptoms. The problem is also a popular application of expert systems. A list of symptoms, fever, coughing, headache, stomach pain and so on, is represented by variables in the input vector. These variables may include continuous variables as degrees of fever, and categorical variables with categories 'no cough', 'moderate coughing', 'heavy coughing'. A list of diseases is represented by an output vector of binary variables with '0' meaning that the disease variable is not diagnosed and '1' that disease is diagnosed.

Figure 7: Diagnostics

diagnostic model can, however, be relevant in a number of applications such as finding the causes of a car which has stopped, a computer which is malfunctioning, etc. in . Using a trained neural network model, the output variable values can be interpreted as probabilities for the different 'diseases' given an input pattern of symptoms/observed abnormalities (Figure 8).
Generic diagnostics

The medical diagnoses NN can be considered as one application of a generic diagnosis network. The generic diagnosis network is characterized by an input vector of continuous and/or discrete variables, neurons organized in several layers, and an output layer with variables representing each of the feasible diagnoses. The output variables in the generic network can be continuous variables bounded by 0 and 1.

It can be proven that if sigmoid transfer functions are used, and the backpropagation algorithm (minimizing a sum-of-squares error function) is applied for training, a network trained with correct output values, i.e. 0 and 1, will produce estimates of posterior probabilities in the Bayesian sense.

Figure 8: Generic diagnostic networks

Quality control

We have already mentioned above that quality control is another facet of classification. A non-industrial quality control application is screening and detecting errors in data records. To maintain a high level of information quality, data collecting/recording organizations spend huge amounts of resources to detect and correct errors in recorded data Figure 9. Neural networks can be trained to screen the data, classify each record as acceptable or suspicious, and correct rejected values with more probable.

Data editing

A very important application of NN classification is editing of large masses of observations which may include errors. The purpose is to classify each observation as either "acceptable" or "suspicious".

The approach is to subject a small random sample to an ideal method of observation which gives accurate observations, but is expensive, as well as to the ordinary method which may give erroneous observations. A network is trained on the sample observations to classify observations as "acceptable" if both methods gave the same results and as "suspicious" if results were different.

The trained network is then used to classify the remaining observations. The class "suspicious" can later be subjected re-observation.

Figure 9: Data editing
An even more intriguing task is to detect grammatical errors in texts. It is usually implemented by means of rule-based systems, but humans do it usually on an intuitive basis. Can NN learn to do it in the same way?

In quality control applications, there are risks for making 2 types of errors,

- **Type 1 error**: processing a *good* item as being *bad*,
- **Type 2 error**: accepting a *bad* item as being *good*,

as indicated in **Figure 10**.

![Quality control table](image)

**Figure 10**: *Quality control*

Within a constrained budget, we can reduce the risk for Type 1 errors only by increasing the risk for Type 2 errors, and vice versa. It is important to consider which is the more important and adjust the classification to the specific application.

**Regression problems**

The *regression* applications are different from the classification tasks. In regression applications, the objective is to find the most likely value within a continuous range of values given a set of input values. In most applications, only one output variable is relevant. A typical application is assessment of the sales *value* of a property given the property size, location, etc. as input variables.

In other applications, the output set can consist of several variables. In addition to expected sales value, the number of interested buyers, the expected time before a contract is signed, etc. are
other relevant output variables to predict. Imagine an application in which the height and weight of a missing person is requested by the police, but for whom only a photographic image is available. Is it be possible to train a neural network to estimate the two values from the image?

The regression problem is expressed formally in Figure 11. The notation $E_y$ is borrowed from estimation theory. It symbolizes the average value of $y$ given the set of input values if we could make a large number of replicated observations of $y$ and its associated input values $x$. Our regression problem consists in training a neural network in such a way that the network generates the best predictions of $E_y$ given $x$. The similarity to the statistical regression is obvious.

---

Regression problems

The second type of NN applications is characterized by:

- $x$ a vector of continuous/discrete input variables,
- $y$ a vector of continuous target variables,
- $w$ a matrix of weights.

Regression is application of a function:

$$E_y = F(x;w)$$

The purpose of the regression function is to give the expected values for $y$ given the values of $x$. The challenge is to find $w$ which gives a satisfactory approximation to the expected values of $y$.

---

Figure 11: Regression application

Data mining

Data mining has been used as a term describing explorative analysis of large data sets (frequently stored in data warehouses) with the objective to identify hidden relationships among the variables in the set.

NN is one of many tools for data mining. Imagine a set of data records each with values for variables in a vector $z$. Initially select the first variable as a target variable and train NN with the remaining as input variables. Rotate the variables as target variable and train. Study the results to see if there is any indication of relationships.

---

Figure 12: Data mining
Regression equations are one type of relationships data mining tries to identify, and one of the most important tool for the search of such relationships in data sets is neural networks (Figure 12).

Data controlled was discussed above as a procedure to classify data records into accepted and suspicious data records. What can be done with the suspicious data records? If both observed as well as target values exist for a sample of observed objects, we can try to train a neural network to predict less suspicious values from the accepted data records (Figure 13). Success depends on the existence of a relationship between good target values and the associated input values. If the relationships can be identified and estimated, computation of improved output values may be possible.

Data imputation

What further processing can be done with the suspicious observations rejected in a data editing application? If we are in a position to identify a subset, x, of the variables z in the data edited, and the values of the x-variables can always be considered correct, then there is a possibility for imputing new data values for other variables denoted \( y \)-variables.

One approach is to use the set of accepted observations to train a NN to predict the values for \( y \)-variables using the \( x \)-variables as input variables. The trained NN can be used to compute imputed values for all \( y \)-variables in the set of observation rejected by the data editing application. The observed values of the \( y \)-variables are finally substituted by the imputed values if the differences exceed preset thresholds.

Figure 13: Data imputation

Neural networks applied on time series

Time series is another interesting application field for neural networks. One reason is that very often there are strong but hidden relations among different time series which can be used, for example for prediction of future development of a series. Analysis of a time series, for instance a monthly series of consumer prices, frequently assumes that the series is composed by several components (Figure 14). A neural network can be trained on historic time series with decomposed components and later used for predicting the decomposition of future time series values. This is important when it is necessary to decide if a change in the time series is caused by the season, or by a real change in the development. In this kind of application, the time series and its components are target variables while year, quarter, month and day are input variables.
In timeseries, there are frequently a seasonal component, i.e. in measurements of temperature and prices of vegetables. Special algorithms have been designed for decomposing the series into a seasonal component, a trend component and an erratic component:

\[ y(t) = a'.x+z(t)+e(t) \]

where \( a \) is a vector of 12 unknown constants, \( x \) is a vector of 12 binary variables \((2^{12})\), \( t \) is a time variable and \( e \) is an erratic variable with a 0 expectation. A NN network may be trained to predict the value of \( y(t) \) with \( x \) and \( t \) as input.

The trained network can be applied to compute the twelve seasonal constants clamping in turn one of the elements of \( x \) to 1, the remaining equal and \( t \) equal to 0.

**Figure 14: Seasonal decomposition**

An alternative approach is to train a network to recognize auto-correlation in a time series. Auto-correlation implies that each term in the series is related to previous terms, i.e. that the series is generating itself (**Figure 15**)

In some time series, the next time value can be assumed to be a function of current and/or previous time values:

\[ y(t+1) = F(y(t), y(t-1),...) \]

(a finite-state machine)

NetMakers data correlation identifies the correlations between \( y(t+1) \) and \( y(t+1) \) as a first step. When the important lags are determined, a NN can be trained to approximate complex autocorrelation functions.

**Figure 15: Autocorrelated series**

The most promising approach is time series which are assumed to be partly determined by autocorrelation and partly by the development of other time series. In complex systems there can be several target variables which are determined by a set of input variables. The network can then be specified as a simultaneous prediction model (**Figure 16**).
Simultaneous predictions

In many research areas, the hypothesis to be investigated must be represented by several simultaneous relations because of mutual interactions among the variables:

\[ f(y_1, ..., y_n, x_1, ..., x_m; a) = 0 \quad (i=1, ..., n) \]

If these can be expressed in a form with all endogenous variables at the left side as functions of the exogenous variables:

\[ y_i = g(x_1, ..., x_m; w) \quad (i=1, ..., n) \]

it is obvious that this is a candidate for application of NN. There are 2 problems. First, as usual it may be impossible to train the net to a satisfactory degree. Second, if estimates of the structural parameter vector \( a \) is required, the transformation of the weights \( w \) to \( a \) may be impossible or difficult.

Figure 16: Simultaneous predictions

Other applications

Financial applications are popular, if not always successful tasks (Figure 17). Applications range from predicting the success of companies based on their past history to predicting the development in the financial stock market. Also government authorities have considered neural networks as an interesting approach to solve some of their tasks. Assessing property values for taxation purposes is one such application (Figure 18). Training neural networks to evaluate how much the individual tax declaration values can change from one year to the next without being suspicious is another application.

Figure 17: Financial predictions
The meteorologists have shown interest in neural networks. Their task is typically a simultaneous prediction of several weather variables based on historic time series for the same and possible additional variables (Figure 19). One student made an interesting study based on historic meteorological measurements for the Pacific.

![Value assessment](image)

**Value assessment**

Tax authorities have studied the possibilities to apply NN networks to assess the value of properties.

The problem is very similar to the sample case in Ch.6 of the Brainmaker manual, except for the fact that the tax authorities do not need to worry about making incorrect assessment. They can simply say that the computed values is the values to be used for taxation purposes.

![Figure 18: Value assessment](image)

Another student was interested in robotics. He imagined that a robot car driver, and assumed that the robot must be able to recognize traffic signs. He trained a network to recognize such signs based on pictures of the signs.

![Weather forecasting](image)

**Weather forecasting**

NN may be an interesting tool for an amateur meteorologist.

Let $y_1(t+1)$ and $y_2(t+1)$ be the temperature and precipitation tomorrow while $y_1(t)$ and $y_2(t)$ are the values today, and $x(t)$ are a vector of today's values of other relevant factors such as wind, clouds, etc. The model is:

$$y_1(t+1) = f_1(y_1(t), y_2(t), x(t))$$
$$y_2(t+1) = f_2(y_1(t), y_2(t), x(t))$$

Meteorological data are easily available, and alternative neural networks for predicting $y_1(t+1)$ and $y_2(t+1)$ may be trained and tested.

![Figure 19: Weather forecasting](image)
Steps in developing a neural network application

In Figure 20, the main steps required for solving a problem by means of neural networks are listed. As we shall learn in the next sessions, each of these steps can be subdivided into a number of details which have to be considered.

Summary

All the previous examples of NN applications require these steps:

1. Specify the purpose of the application precisely.
2. Identify the target variables and check that data for the variables are available.
3. Identify the input variables and check that available data are compatible with the target data.
4. Select the number of layers, the numbers of neurons, the transfer functions and the rate of learning for each layer, tolerances, stop criteria for the training, and other training parameters.
5. Divide randomly available data in training and test sets.
6. Train and test the network(s).

Figure 20: Summary

Exercises

a. Go to the Section on literature and see if any of the application oriented titles are available in your library. Select one you find interesting, and try to make a design for an experiment including collection of data, recording data in a form which can be read by NetMaker, specification of the network you think will be suitable including the setting of a definition file. If no literature is available, you have the introduction to Neural Networks by J.Lawrence which was part of the software package

b. Read Chapter 7 in the BrainMaker User's Guide and Reference Manual. Select the Tic-Tac-Toe example and study it carefully. Note that the network is not trained to play, but to evaluate moves. To which extent did it follow the pre described rules for specification in Figure 20?
c. Activate your NotePad and read the *TicTac.def* file from the BrainMaker folder or click on the Datafiles in the window at your left hand. Check all the specifications. Note that the facts are included in the definition file and not a separate file as in the XOR example you studied in the last session. When the fact file is small, it can be better to have everything in one file.

d. Load BrainMaker and read the *TicTac.def* file. Train the specified network and study the results. It is not the game itself which is interesting, but how the network is trained to learn the examples in the implicit fact file.

e. Study the *Optical Character Recognition* example in Chapter 7 of the BrainMaker manual. Consider the differences in the specification of the XOR problem and the OCR problem.

f. Read Chapter 11: Data Preparation, in Lawrence.

g. Read Chapter 12: Advanced Design Topics, in Lawrence.
Session 5: Formal description

Top-down description

In this section, we summarize the feed-forward neural network in a formal description. We use a top-down approach, which means that we take the neural network discussions from the previous sessions as a starting point and proceed to the details (Figure 1). As in object-based system theory, the description can be done by classes of objects. For a C++ object-oriented discussion, see Rogers.

Figure 1: Top down approach

We start by describing data structures of objects. There are 3 network types of objects which we will distinguish in our description (Figure 2).

Figure 2: Object types
Sets of data

It is convenient to distinguish between the 5 sets of data as indicated in Figure 3 even though they are overlapping. The input data can be considered in a wide sense as the problem we aim to solve. The 'problem' may be a picture we want to identify or classify, or a set of numerical variables, for instance the measurements of a property, which has a mapping to a value set from which we seek the correct value. The input set is subdivided into records. Each record is associated with some problem object. Usually an input record comprises several variables (Figure 4).

Figure 3: Data sets

Figure 4: Input data
The input set is denoted by

\[ X = \{ x^{(k)} \} \]

where the \( i \) refer to the input variable and \( k \) to the record number.

The target data record is the solution, identification, classification category or property value we search. If we want a mapping to a continuous variable as a property value, the target record can consist of only one single variable. If we want to classify, a neural network works with categorical target variables. Each category has a unique name, and each name is transformed to a binary variable with 0 or 1 as the only 2 permitted values. A categorical variable therefore are transformed to as many binary target variables as there are existing categories.

In some applications, we may have several continuous and/or category target variables. The net is then performing simultaneous mapping Figure 5. The target set of data is referred to by:

\[ Y = \{ y^{(k)} \} \]

where \( j \) is an index for the output variable. If there is only a single output variable, the subscript is dropped.

![Target data records](image)

**Figure 5: Target data**

The third data set consists of computed output data records (Figure 6). These are in format like the target data records, but contain the result of the mapping done by the network. Ideally, we would like to have networks which produce output records identical to the target data records. As we shall see in later sessions, we have to be satisfied with output data records which deviate from the target data records within pre-set tolerance limits. The output set of variable values is denoted

\[ Y' = \{ y'^{(k)} \} \].
Training and testing are important processes in the development of networks. These functions are carried out on example data records (Figure 7). An example data record is a pair of input data and target data records associated to the same real life object. A collection of example data records, frequently compiled for a random sample of real life units, is used as a training data set, while another, usually independent, collection is compiled as a set of testing data records. We denote an example data set as:

\[ F=\{X,Y\} \]
Evaluation data records are a last type of data sets (Figure 8). An evaluation data record is a pair of target data and output data records, which permits to compute the deviations of the values between target and output variables to evaluate the performance of a neural network. The evaluation set is denoted:

\[ E = \{Y', Y\} \]

**Figure 8: Evaluation data**

**Network topology**

The topology describes how the network is designed (Figure 9). The design comprises how

**Topology**

- Layers in feed-forward networks
- Connections in feed-forward networks

**Figure 9: Network topology**
many layers of neurons the network has and their size in number of neurons. The inputs are by some authors counted as a separate layer while others do not consider the inputs as a layer. In this course, the input data are introduced through the input sources and the layer of input sources is not counted as a layer because it does not contain any neurons (Figure 10). There are 2 kinds of input sources, the ordinary sources through which the input data are introduced and the threshold sources which generate monotonously inputs with value 1 for each record processed.

**Figure 10: NN layers**

The minimum number of layers of neurons in a network is therefore 1, the output layer of neurons. A network with only an output layer is called a single layer network. Layers between the input sources and the output layer are named hidden layers. Networks with 1 or more hidden layers are called multi-layer networks. Multi-layer networks can adjust to more complex mapping relationships than single layer networks.

A network with hidden layers larger than the input source layer and the output layer of neurons is said to have a convex topology, while a network with hidden layers smaller than both the input and the output layers has a concave topology. Networks being neither convex nor concave are said to possess an ordinary topology. We shall limit our discussions to ordinary topologies even though experimentation with extra ordinary topologies is encouraged.

The type of neurons is also characterizing the network topology. The characteristics of a neuron were discussed in the previous sessions. A neuron receives input values from sources or other neurons and transforms the input to an output value. Many possible transformations exist. We limit our concentration to normalization of inputs by summation of all input values and transformation by one of the three most frequently used activation functions (Figure 11).
Figure 11: Neuron properties

The step function is defined by Figure 12

\[ y' = 0 \text{ if } S \ x < A \]

or

\[ y' = 1 \text{ if } S \ x \geq A \]

where \( x \) represent the inputs to and \( y' \) is a computed value by the neuron. \( A \) is the step point frequently set equal to 0. The function can respond to any input with one of two values. Note that in many implementations, the lower value of the function may be defined as -1 instead of 0.
The linear function is slightly more complex (Figure 13):

\[ y' = 0 \text{ if } S \times < A \]

or

\[ y' = \frac{S \times (B-A)}{B-A} \text{ if } S \times = > A \text{ AND } S \times < B \]

or

\[ y' = 1 \text{ if } S \times = > B \]

where \( A \) and \( B \) are to preset points such that \( B > A \). As you can see, the function is not strictly linear but composed by 3 linear fragments.
The sigmoid function (Figure 14) is probably the most frequently used activation function in connection with feed-forward networks based on the backpropagation learning algorithm:

\[ y' = \frac{1}{1 + e^{-Sx}} \]

The explanation for its popularity is that it is differentiable, which is a requirement by the backpropagation training algorithm.

The connections between neurons are the second part of describing the topology (Figure 15). A

**Figure 15: Connections**

- Non-recursive networks
- Fully connected networks
- Relations characterized by weights

**Figure 16: Weights**

- Continuous weight values
- Determined by iterative training
- Represent the permanent network memory
number of different topologies can be designed by variation of the connections. The feed-forward networks are characterized by directed connections starting from neurons in one layer (or the input sources) and ending in neurons of the next layer. The networks we work with are mainly fully connected, i.e. each neuron in one layer is connected to all neurons in the next layer. Each connection is characterized with a single number, the weight (Figure 16). The set of weights in a feed-forward network is symbolized by

\[ W^{(m,n)} = \{w_{ij}^{(m,n)}\} \]

where \(i\) and \(j\) indicate transmitting and receiving neuron respectively, and \(l\) is the layer of the receiving neuron, while \(m\) refers to the weight set after the \(m^{th}\) record has been processed in the \(n^{th}\) repetitive through training examples.

**Relations**

Assuming a feed-forward topology and sigmoid activity functions, we can now write out and inspect the complete relation between an output variable and the input variables. For a single layer network it is

\[ y' = \frac{1}{1 + \exp(-w_0x_0 + s_i^Bw_ix_i)} \]

where \(x_0\) is an threshold variable always transmitting value 1 and \(B\) is the number of regular input variables. By specifying \(B=1\) is easy by means of a calculator to compute a set of output values and verify the sigmoid form of the output curve.

The formula becomes more complicated when there are several output variables and hidden layers. For \(j=1..A\) output variables, \(C\) variables in a hidden layer, and \(B\) input variables the formula looks like this:

\[ y'_j = \frac{1}{1 + \exp(-w_{0j}z_0 + s_k^Cw_{kj} \cdot (1 + \exp(-w_{0k}x_0 + s_i^Bw_{ik}x_i)))} \]

It takes some time to inspect this formula.

**Procedures**

Three procedures are needed for the description of the neural networks (Figure 17). The core of the learning procedure is the Backpropagation (BP) algorithm (Figure 18). The algorithm was described in Session 2.

Each time a new record \(m\) in the \(n^{th}\) run is processed by BP, an updating of the weight set

\[ W^{(m,n)} \rightarrow W^{(m,n+1)} \]

can be done. This is called record (pattern) mode of learning and is the usual Backpropagation
Figure 17: Procedures

Procedures

• Learning
• Testing
• Running

Figure 18: Learning algorithm

Learning algorithm

• Iterative use of the Backpropagation algorithm
• Subject to training parameters and stop criteria
• Training by record
• Epoch training

mode. An alternative is to save the computed changes for each weight, and update the weight set by the average change at the end of each iterative run

\[ W^{(n)} \rightarrow W^{(n+1)} \]

This mode is called epoch training. Epoch training is sometimes faster than the record training. Usually satisfactory results are obtained by record training.

The algorithm for the updating/training can be explained in more detail for a two layer network with 2 input sources, 2 hidden neurons and 1 output neuron (in addition there are 2 threshold input sources), and sigmoid activation functions [Lippmann 1987].
The updating, or the training, is aimed at adjusting the weights to decrease the deviation between target and computed output values for the current example. The process starts with adjusting the weights of the connections to the output neuron, and continues with adjusting the weights of the connections to the hidden neurons.

The adjustment of the weights of the connections to the output neuron can be expressed by:

\[ w_{j1}(n+1) = w_{j1}(n) + a \cdot x_{j1} \cdot y(x_{j1}) \cdot (1-y(x_{j1})) \cdot (t-y) \]

where \( a \) is the learning rate, \( x_{j1} \) is the value of the input from the hidden neuron \( j \), and \( y(x_{j1}) \cdot (1-y(x_{j1})) \) is the derivative of the sigmoid function \( y \) in the point \( x_{j1} \). The product of these factors multiplied by the deviation gives the wanted adjustment in weight \( w_j \). Index \( j = 0, 1, 2 \) refers to the threshold input source, and the 2 outputs from the hidden neurons.

The updating of the weights for the hidden neurons are more complicated because no explicit deviations from targets exist for the output of these neurons. Instead computed deviations are used. The expression \( y(x_{j1}) \cdot (1-y(x_{j1})) \cdot (t-y) \) use above multiplied by the weight, \( w_{j1}(n) \) for the connection to the output neuron, is used as a computed deviation. The second step of adjusting the connection weights to the hidden neurons from the input variables \( z_i \) can then be written:

\[ w_{ij}(n+1) = w_{ij}(n) + a \cdot z_i \cdot y(z_i)(1-y(z_i)) \cdot [ w_{j1}(n) \cdot y(x_j)(1-y(x_j)) \cdot (t-y) ] \]

\( z_i \) denotes the input variables and the expression enclosed in square brackets, [..], is the computed deviations for the outputs from the hidden neurons' computed 'targets'.

The formulas become more complex when there are 2 or more output neuron.

Testing is a very important procedure for development and implementation of neural networks (Figure 19). Since training is repeated until the training requirements are satisfied, a user can easily believe that the results must be satisfactory. To check that the network really is able to generalize, it should always be run on an independent set of test example after training.
Experience shows that the network can be trained too much, and become useless when confronted with a test set. To avoid this situation of overfitting, a good strategy is to test the network regularly on independent test examples during the training. Testing while training can be carried out after each iterative training iteration, or alternatively, after a specified number of iterations.

A trained and satisfactory tested NN can be used for operative tasks in 2 ways (Figure 20). The simplest is to run the work within the network program. BrainMaker has a mode dedicated for operative running of input data sets with the results recorded in an output set. An alternative approach is to use the weight set W obtained after training in an application program which are tailor-made for the considered application. To embed W in an application program is a minor task.

**Parameters**

A NN development can be controlled by parameters and required conditions (Figure 21). In many implemented systems for development of NN, only a few may be options for the user, the remaining are set in the programs.

The mode, the choice between learning, testing and operative running, is always determined by the user. Initial weights on the other hand are not always subject to the control of the user. They may be randomly set by the NN system according to a specified probability distribution, or the selection of the initial weight set can be left to the user. Taken into account the importance the initial starting point for the path to the wanted final weight set, a developer may want to control the selection process.

Learning rate and momentum determine the relative adjustment of the weight set after processing a record. Usually most NN systems permit the user to set at least the learning rate. However,
Running

A trained network can be used for operative tasks

- Running input records by the network program
- Embedding weight matrix in an application system

Figure 20: Running the network

Parameters

- Mode
- Initial weights
- Learning rate
- Momentum
- Epoch training
- Tolerance
- Testing frequency
- Saving frequency
- Stop criteria
- Input/output format

Figure 21: Parameters

More advanced forms of learning rates, for instance dynamic learning rates which change after each run by the percentage of 'good' outputs, are not always available.

The choice between record and epoch training may not be important since in most cases the former will be superior. However, in a few situations with very large networks, record training may be too time consuming and the possibility to select epoch training will be wanted.
To set the training *tolerance* will always be an option for the user. The possibility to specify different tolerances for learning and testing is not usually offered the user, but may be important for the researcher. Specifying tolerances as functions of training/testing status are not usual.

If *testing while training* is available, the question how frequent the testing should be *performed* and how frequent should the developing network be *saved* must be decided. If the network is not to large and the platform fast, testing can be performed after each iteration of training, while the network can be saved for instance after each 20\textsuperscript{th} iteration. It must be possible to stop the network when the testing indicates that the training has passed the 'optimal' point, open the last saved version of the network before this point and run the network for the few iterations needed for getting to the optimal network (in average 10 iterations if the network was saved for every 20\textsuperscript{th} iteration.

There must be at least one possibility to set one condition to stop the training after a specified number of runs. Stop criteria depending on the training development are desired.

Last, but not least, a network should offer *format flexibility* for reading input and example sets and writing output sets.

**Exercises**

a. Read Appendix B: Linear Algebra, in Lawrence.

b. Read Appendix C: Back Propagation Mathematics, in Lawrence.

b. Read Appendix E: Neuron Transfer Functions, in Lawrence.
Session 6: Classification

An image recognition problem

Image recognition is an old challenge to the computer scientist. It became also early one of the popular application problems in the Neural Network area. The application we discuss in this session is outlined in Figure 1. It is based on a paper [Nordbotten 97] on research in the potential of knowledge based and neural network models combined in a hybrid system. You can find a full text copy of the paper in pdf format in hybrid.pdf in the course section Articles in the left window.

Image recognition

This is a simplified version of an application used in a research paper, A HYBRID SYSTEM FOR AUTOMATIC CODING OF HANDPRINTED RESPONSES IN STATISTICAL SURVEYS. Interested can find a link to this paper at the web course pages (Literature page).

The limited problem we are discussing in this lecture is how to train a network to distinguish each of 12 capital letters (A-L) from each other. For training, we use a standard image for each letter and 2 distorted versions of each letter.

To test the networks capabilities, we have constructed 2 other distorted sets of the 12 letters, but have made different variations in their forms.

25/5/00 Sven Nordetten

Figure 1: I(mage recognition)

Imagine an organization collecting data by means of a form on which you are requested to print the name of your place of birth. This form must be automatically processed, and you are asked to paint the name in capital letters in preprinted boxes, one box for each letter. The task of the hybrid system is to recognize these letters and compile them into a sequence which represents the name of your birth place. Is it possible to find the relations between the form of the letters and the letters? Reading this piece of text you do perform this process without much effort, but can we get a computer to do the same?

To focus on the character recognition and simplify the discussion, we consider only first 12 capital letters and disregard any spelling control. We approach the representation of letter images by drawing a grid over the letter images (Figure 2). Each grid cell is assigned to one of 2 possible categories depending on whether the cell encloses part of the letter or not. Each grid cell is assigned the value 1 if part of the letter is within the cell (we call it a black cell), else the value 0 (called a white cell). By increasing the number of lines in the grid, we can increase the resolution of the details of the image representation. In general, we expect that increased resolution implies increased recognition. However, the number of cells will increase fast and so will the processing cost. This increase is usually referred to as the curse of the dimensionality

84
because when we increase the number of lines in the grid, the number of cells (input variables) increases exponentially.

The 12 letters we use in this session are the capital letters A to L. Each letter picture is represented in a grid with 8 rows of 8 cells each. The size of the letters is normalized in such a way that the respondents are requested to 'fill' the boxes with the letters printed. The letters shown in Figure 3 are called the standard letters. We must expect, however, that the letters printed by the individual survey respondents will vary significantly from the standard forms.

The original application referred to was more complicated (Figure 4). The letters used were the 26 capital letters of the English alphabet. The task was to recognize 368 American city names spelled out in hand-printed letters. In addition to the respondents' painted letters which usually deviated from the standard letters, misspellings of the names also appeared. The system used to
Hybrid problem

"The classification list used was the names of cities of about 50,000 or more residents in the USA. A total of 548 different city names with names up to 47 characters were recorded. Each city name was considered a keyword or name for a category in a geographical classification by which each respondent should be classified."

Figure 4: Hybrid problem

solve the problem was a hybrid combination of a neural net trained to recognize each image separately, and a knowledge based system for checking correctness of the sequence of recognized letters. If some of the letter in a city name were incorrectly recognized, the knowledge base system, which comprised a list of all possible city names, would use a set of rules to transform the incorrectly spelled names to the most likely correct names. The use of hybrid systems permits more realistic solutions to many problems.

Setting up training and test files

To train a neural network to solve the problem outlined, BrainMaker requires a training file with examples of letter pictures and the corresponding letter symbols starting with the word ‘facts’. Note that BrainMaker uses the terminology picture for an image input and symbol for a categorical output.

Each pair of picture-symbol must be represented as follows in the file:

1. The picture is represented in 8 rows with 8 positions each corresponding to an 8x8 grid of cells.
2. Our simple application works with binary cell values either 'X' or 'blank'.
3. A cell with value 'X' represents a part of the letter form, while value 'blank' means that the grid cell is untouched by the letter.
4. After the 8th row follows a row with the target output symbol.

In Figure 3, we saw how the pictures for the 12 standard letters looked. To train the network to recognize different variants of the standard letters, we need examples of different printed versions for each letter. In a real investigation, we would collect and use painted examples for each letter from a random sample of respondents. Another approach can be to generate synthetic examples of the letters by producing artificial distortions of the standard letter fonts (Figure 5).
Figure 5: Training file

The distortions can be created by adding and deleting X's in the letter grid cells. This can be done randomly by a computer program. In the application of this session, we use the standard letter pictures and 2 distorted picture versions of each letter, in total 36 examples. In the original application referred to above, the training was done on more complex sets of letters (Figure 6).

Figure 6: Hybrid training data

"The network weights were computed by a Backpropagation training algorithm. The training was carried out based on 520 patterns representing the 20 slightly distorted versions of the standard character alphabet. These versions were generated from the standard alphabet fonts, by a distortion similar, but independent of the distortion used for the descriptor files. The distortion used for the 20 versions of the alphabet were based on noise probabilities 0.05 for both white and black noises. On the basis of previous experience, this specification gave a variation of character fonts suitable for training the ANN to recognize distorted character patterns."

Figure 7: Test set

A test set can be constructed in the same set. We used two distorted picture versions of each letter as a test set (Figure 7). These sets were of course different from those used for training. Including the standard set, would be meaningless since it was used for training. Compared with the full-blown experiment (Figure 8), the test set we use is rather simple.
**Test file**

The test images are distorted standard images not used in the training. The more the test images are distorted, the harder the test will be.

The test file consists of 2 distorted versions of the 12 letters, in total 24 images.

It is possible to introduce a gray scale by using pixels with values between 0 and 1. The images might be distorted in scale by adding or subtracting randomly determined values to the scale value.

Colors might be represented by codes. In the current example, each image would then be composed of 64 symbols, each with as many alternative codes as there were colors. For example, with 16 colors, there would be 1,024 binary input variables. These can be distorted by changes in the code of pixels.

**Figure 7: Test file**

---

**Hybrid test data**

"The distortion in the last step was performed sequentially for each character pattern of the descriptor string as a random process by which each black pixel of the standard character font pattern was exposed to a pre-set risk for being erased by a white pixel, while each white pixel in the standard pattern was exposed to be transformed to a black pixel."

"The probabilities for white and black noises used were specified to 0.125 and 0.075, respectively."

**Figure 8: Hybrid test data**

---

**Definition file**

As the fact files, the definition file has also been made without use of NetMaker.

It specifies the type of input, "picture", and the size, 8x8 pixels, followed by number of hidden neurons, 15, in 1 layer. The names of the training and testing files must be specified.

In the BrainMaker manual, there is an example of how training facts can be included in the definition file. In our application, they are kept as separate files which gives a higher freedom to make changes. Finally, the output type, symbol, is specified, and the names of the 12 output variables are listed as dictionary output.

The network will have 175 and 172 weights in its two matrices.

**Figure 9: Definition file**
Tuning the Learning Rate

In our discussion so far, we have only mentioned that fast learning may result in passing the minimum point in the RMS—weight space. We can reduce the learning rate proportionally as we approach the situation when the training set is learned by selecting the option Linear Learning Rate.

In the current application, we start with a learning rate equal to 0.5 and reduce it linearly to 0.1 as we approach 100% good recognition of the training set. We could alternatively trust BrainMaker and use Automatic Heuristic Learn Rate.

Figure 10: Tuning the learning rate

Tuning the Training Tolerance

The training tolerance defines how close the network output must be to the target value before it is considered good. It is a good strategy to start with a wide tolerance and make it more narrow when the training output becomes better. BrainMaker has a menu which easily permits automatic tuning. We start with both training and testing tolerances 0.1. Turn on tuning and set Minimum Tolerance = 0.05.

When we have a test sample, it is always good to Test While Training and save network after every 10 Runs.

Figure 11: Tuning the training tolerance
Figure 12: Hybrid specification

Training the network for letter recognition

The training and test files used are named alfa.fct and alfa.tst. The definition file used is alfa.def. As Figure 9 shows, the network selected is a 2 layer network (1 hidden and 1 output layer). The number of hidden neurons is set to 15 after a few experiments. Note that the output is declared as symbol and that there are in total 12 output variables, one for each symbol. The term symbol is another name for categorical or nominal variables. We aim at training the network to predict the conditional probabilities that the different symbols correspond to the picture input. This specification result in 975 and 192 weights in the 2 weight matrices, respectively.

Another specification we can set by means of the BrainMaker toolbar is to require a linearly decreasing learning rate (Figure 10). There is no reliable rule for specifying the learn rate, but experience indicate that starting with a rate about 1.0, which is decreased as the learning improves, is a good strategy. Another important parameter in the training model is the tolerance setting. This is set by selecting Parameters/Training Control Flow and mark Tolerance Tuning. From the original experiment, we obtained a some experience we shall use (Figure 11) As we already has recommended, testing while training is a useful feature. In this application, we decide to test the network after each run and to save after each 10th run. Also other options can be considered (Figure 12)

During the training, we turn on a new display, Display/Show Histogram, to monitor the training (Figure 13). Interpretation of the 2 histograms, one for the weight distribution in the matrix before the hidden layer of neurons, and a second histogram for the matrix before the output layer of neurons, is discussed in the BrainMaker manual, Chapter 8. The two histograms indicate a healthy network (Figure 14).
Training and testing

After checking the file selection and activating Test Statistics, the training can start. We also turned on Display Show Histograms which will give information as to the distributions of weights. If the network starts following a bad path, it may end up brain-dead.

The network trained for 849 runs before it computed all 36 training images correctly subject to the Maximum Learning Tolerance 0.05.

At this point the Learning Rate was tuned down from 1.00 to 0.15.

Figure 13: Training and testing

Figure 14: Distribution of weights

The histograms show the distribution of weights in the two matrices. Note the difference in the y-axis which reflects the difference in the size of the matrices.

Distributions which are skewed at one side of the diagram indicate a network which is useless and cannot be trained.

The distributions in the present network seem healthy.

We made a separate run through the test set to obtain individual predictions (Figure 16). It is important to understand that the output value associated with a symbol is an estimate of the conditional probability for the respective symbol given the input picture. For this reason, the sum of two probabilities can quite well be greater than 1.0. The most likely symbol to predict is the one with the highest probability. Using this rule when inspecting the predicted outputs, 3 of the 24 predictions were incorrect.
Figure 15: Inspection of the test results

The test statistics show that the minimum RMS error = 0.1307 was attained at Run 571 and the corresponding network was renamed alpha not for further work.

Figure 16: Individual recognitions

As we have done in previous applications, a separate test run was set up. In File/Write Facts To File we specify Symbol for both Output and Pattern. It should be remembered that in contrast to previous applications, we have 12 simultaneous outputs.

Since we are working with sigmoid transfer functions, each output will be a value between 0 and 1. The value can be interpreted as the probability for the respective output to be true. We use the output variable with the highest probability for each image as prediction of its pattern. 21 of the 24 images in the test sample were correctly recognised.

Because the probabilities are not representing a probability distribution, their sum need not be equal to 1.

Figure 17 shows the first of the three incorrect predictions. The picture of an 'I' is predicted to be an 'L', even though with a low probability (0.16). The standard picture of 'I' (see Figure 3) is a column of 'X's only. The picture to be interpreted was seriously distorted which contributed to uncertainty (low probability) in prediction.

The second and third failures were two picture of 'J' as seen in Figure 18. The first was predicted to be 'D' and the second to be 'C'. A careful inspection of the pictures, may explain why.

Failing to recognize more than 10% of the test set cannot be considered satisfactory. We ask what can be done to improve the recognition rate. Several possible ways are indicated in Figure 19, and more can be added. Since the number of examples on which we have trained the network
In order to learn about the features of BrainMaker, we investigated another possibility, namely pruning the network. Pruning means to delete small weights based on the assumption that they are uncertain estimates and may disturb the prediction. Figure 20 shows how pruning can be done and what the result was in our application. We can look away from pruning as a means to improve the prediction rate at this stage of our application.

A second possibility is to specify a more complex network structure as indicated in Figure 21. Two hidden layers were specified, as well as an automatic increase in number of neurons when no improvement in learning was detected. The results were disappointing (Figure 22). The network increased to 147 neurons in each of the hidden layers and learned the 36 training
Ways for improvement
The letter recognition application failed in recognising correctly
3 out of 24 images; i.e. The success rate is 0.8 which is not good.
Possible improvements:

- The training sample is too small and should have had more than
  3 versions of each letter. Noise too strong?

- A network with 2 hidden layers may be more adequate for this
  type of problem?

- The distortions in the training sample are not representative for the
  the test sample?

- The resolution (8x8) of the patterns is too low?

Figure 19: Ways of improvement

Pruning the network
To avoid small noisy weights, we prune the network, eliminating
all weights with absolute values less than 0.1 in both matrices are
eliminated.
This exercise was not successful.
The results this time were about
as for the unpruned network:

9. J -> L (0.1742)
10. J -> D (0.6026)
11. J -> C (0.3635)

Less and harder pruning was tried
without any better results.

Figure 20: Pruning the network

examples perfectly. However, the ability to generalize and solve the examples in the test set was
bad.

A third option is to open the weight file and see if we manually can correct the weights for the
implied letters in such a way better results are obtained (Figure 23). This kind of micro-
manipulations may be successful, but require a very detailed understanding of the weight
matrices and their relative effects.

What have we learned in our efforts to improve the letter recognition? Figure 24 indicates a few
possible answers. The approach most likely to give success is to increase the training file.
Advanced networks

To investigate the possibilities of using 2 or more hidden layers with neurons, a very advanced model was specified. It was set up with 2 hidden layers, each with 8 neurons each. If the RMS error did not decrease with 0.05 or more every 20 run, a BrainMaker algorithm added a new neuron where its effect was strongest. Linear Tuning of Training Rate and Tuning of Training Tolerance were also specified.

Figure 21: Advanced networks

Disappointing results

The network learned the training file images about perfect, but at the cost of a large number of hidden neurons. The result, as should be expected, was a network with poor generalization which could not compare with the simpler network already described.

Figure 22: Disappointing results
Figure 23: Microscopic investigation

One way of improving the network is to manually edit connections 1 share with L, and J with C and D by inspecting the weights from the to those firing output neurons. Use Edit/Connection Matrix or an ASCII text editor:

<table>
<thead>
<tr>
<th>A</th>
<th>B</th>
<th>C</th>
<th>D</th>
<th>E</th>
<th>F</th>
<th>G</th>
<th>H</th>
<th>I</th>
<th>J</th>
<th>K</th>
<th>L</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>2</td>
<td>3</td>
<td>4</td>
<td>5</td>
<td>6</td>
<td>7</td>
<td>8</td>
<td>9</td>
<td>10</td>
<td>11</td>
<td>12</td>
</tr>
<tr>
<td>12</td>
<td>13</td>
<td>14</td>
<td>15</td>
<td>16</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
<th>6</th>
<th>7</th>
<th>8</th>
<th>9</th>
<th>10</th>
<th>11</th>
<th>12</th>
</tr>
</thead>
<tbody>
<tr>
<td>13</td>
<td>14</td>
<td>15</td>
<td>16</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
<th>6</th>
<th>7</th>
<th>8</th>
<th>9</th>
<th>10</th>
<th>11</th>
<th>12</th>
</tr>
</thead>
<tbody>
<tr>
<td>13</td>
<td>14</td>
<td>15</td>
<td>16</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
<th>6</th>
<th>7</th>
<th>8</th>
<th>9</th>
<th>10</th>
<th>11</th>
<th>12</th>
</tr>
</thead>
<tbody>
<tr>
<td>13</td>
<td>14</td>
<td>15</td>
<td>16</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
<th>6</th>
<th>7</th>
<th>8</th>
<th>9</th>
<th>10</th>
<th>11</th>
<th>12</th>
</tr>
</thead>
<tbody>
<tr>
<td>13</td>
<td>14</td>
<td>15</td>
<td>16</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
<th>6</th>
<th>7</th>
<th>8</th>
<th>9</th>
<th>10</th>
<th>11</th>
<th>12</th>
</tr>
</thead>
<tbody>
<tr>
<td>13</td>
<td>14</td>
<td>15</td>
<td>16</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
<th>6</th>
<th>7</th>
<th>8</th>
<th>9</th>
<th>10</th>
<th>11</th>
<th>12</th>
</tr>
</thead>
<tbody>
<tr>
<td>13</td>
<td>14</td>
<td>15</td>
<td>16</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
<th>6</th>
<th>7</th>
<th>8</th>
<th>9</th>
<th>10</th>
<th>11</th>
<th>12</th>
</tr>
</thead>
<tbody>
<tr>
<td>13</td>
<td>14</td>
<td>15</td>
<td>16</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
<th>6</th>
<th>7</th>
<th>8</th>
<th>9</th>
<th>10</th>
<th>11</th>
<th>12</th>
</tr>
</thead>
<tbody>
<tr>
<td>13</td>
<td>14</td>
<td>15</td>
<td>16</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
<th>6</th>
<th>7</th>
<th>8</th>
<th>9</th>
<th>10</th>
<th>11</th>
<th>12</th>
</tr>
</thead>
<tbody>
<tr>
<td>13</td>
<td>14</td>
<td>15</td>
<td>16</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
<th>6</th>
<th>7</th>
<th>8</th>
<th>9</th>
<th>10</th>
<th>11</th>
<th>12</th>
</tr>
</thead>
<tbody>
<tr>
<td>13</td>
<td>14</td>
<td>15</td>
<td>16</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
<th>6</th>
<th>7</th>
<th>8</th>
<th>9</th>
<th>10</th>
<th>11</th>
<th>12</th>
</tr>
</thead>
<tbody>
<tr>
<td>13</td>
<td>14</td>
<td>15</td>
<td>16</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
<th>6</th>
<th>7</th>
<th>8</th>
<th>9</th>
<th>10</th>
<th>11</th>
<th>12</th>
</tr>
</thead>
<tbody>
<tr>
<td>13</td>
<td>14</td>
<td>15</td>
<td>16</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
<th>6</th>
<th>7</th>
<th>8</th>
<th>9</th>
<th>10</th>
<th>11</th>
<th>12</th>
</tr>
</thead>
<tbody>
<tr>
<td>13</td>
<td>14</td>
<td>15</td>
<td>16</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
<th>6</th>
<th>7</th>
<th>8</th>
<th>9</th>
<th>10</th>
<th>11</th>
<th>12</th>
</tr>
</thead>
<tbody>
<tr>
<td>13</td>
<td>14</td>
<td>15</td>
<td>16</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
<th>6</th>
<th>7</th>
<th>8</th>
<th>9</th>
<th>10</th>
<th>11</th>
<th>12</th>
</tr>
</thead>
<tbody>
<tr>
<td>13</td>
<td>14</td>
<td>15</td>
<td>16</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
<th>6</th>
<th>7</th>
<th>8</th>
<th>9</th>
<th>10</th>
<th>11</th>
<th>12</th>
</tr>
</thead>
<tbody>
<tr>
<td>13</td>
<td>14</td>
<td>15</td>
<td>16</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
<th>6</th>
<th>7</th>
<th>8</th>
<th>9</th>
<th>10</th>
<th>11</th>
<th>12</th>
</tr>
</thead>
<tbody>
<tr>
<td>13</td>
<td>14</td>
<td>15</td>
<td>16</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
<th>6</th>
<th>7</th>
<th>8</th>
<th>9</th>
<th>10</th>
<th>11</th>
<th>12</th>
</tr>
</thead>
<tbody>
<tr>
<td>13</td>
<td>14</td>
<td>15</td>
<td>16</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
<th>6</th>
<th>7</th>
<th>8</th>
<th>9</th>
<th>10</th>
<th>11</th>
<th>12</th>
</tr>
</thead>
<tbody>
<tr>
<td>13</td>
<td>14</td>
<td>15</td>
<td>16</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
<th>6</th>
<th>7</th>
<th>8</th>
<th>9</th>
<th>10</th>
<th>11</th>
<th>12</th>
</tr>
</thead>
<tbody>
<tr>
<td>13</td>
<td>14</td>
<td>15</td>
<td>16</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
<th>6</th>
<th>7</th>
<th>8</th>
<th>9</th>
<th>10</th>
<th>11</th>
<th>12</th>
</tr>
</thead>
<tbody>
<tr>
<td>13</td>
<td>14</td>
<td>15</td>
<td>16</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Figure 24: How can we learn from the application

How can we learn from the application

We have demonstrated different approaches for improving the recognition of the 12 letters in our application, and they were not successful for all letters.

Taking into account that our training set consisted of 3 variation x 12 letters, we should not expect better results.

The main condition for good recognition is a training on a set which has a satisfactory size and variation.

Study the application “Hybrid...” referred to in the Literature pages.

Exercises

Exercises

a. You will find copies of the alfa.def, alfa.fct and alfa.tst files the Datafiles section to the left. Copy this files into your c:\BrainMaker folder. The steps are:

1. Open the Datafiles folder
2. Mark and Edit/select all the file you want to copy.
3. Load NotePad and Edit/paste.
4. Save the file in the BrainMaker folder by means of File/Save as.

Read the alfa.def file into Brainmaker. Click on Connections and change the structure of the NN to a single-layer network, Try to train the network by means of the fact file alfa.fct and test the
trained NN on the *alfa.tst*. Make a **report** of the number of runs required and the test results obtained. Make a comparison of the results you got with those reported above.

b. **Repeat** the training of your single-layer NN once more. **Compare** the number of runs required in the first and the second training you did. Discuss the differences. As pointed out, the differences are caused by the random initial weights.

c. Why not try some **alternative** specifications? If you are a patient researcher, you may be able to improve the character recognition.
Session 7: Regression

Continuous output variables

The second main type of NN applications is prediction of continuous output variables. The task is to predict the most probable value of a continuous output variable given a set of input values [Bishop 1995]. In the NN literature, this type of applications is considered more difficult than the classification type. We shall use 2 examples from the BrainMaker package.

LOS

We start by studying an application from the BrainMaker manual about the Length of Stay, the LOS application (Figure 1). This application is discussed in Chapter 5: Medical tutorial in the BrainMaker manual. The scenario for this application is the need of a hospital administrator to predict the length of stay for admitted patients expressed by the continuous variable number of days. The assumption is that the medical observations of the patient when admitted can be used as a basis for predicting the length of the stay. From the description in the figure, 64 patient records are available for the analysis. Among these, 58 randomly selected will be used for training and the remaining 6 for testing the trained models. Each record contains values for the LOS target variable and 11 input variables. The inputs are partly variables with continuous values, and partly categorical and binary variables. The data are recorded in LOS files which are accessible at the distribution diskette from CSS. Figure 2 gives a list of the files we use in this application.

Figure 1: LOS application

![Figure 1: LOS application](image)
Files used by the application

The following files are used by this application:

- **LOS.dat**: Original data file.
- **LOS2.dat**: Data file adjusted for BrainMaker use.
- **LOS2.dat**: Training file with 58 records.
- **LOS2.txt**: Test file with 5 records.
- **LOS2.net**: Network trained until all records learned.
- **LOS2stat**: Training statistics file.
- **LOS2.dat**: Testing statistics file.
- **RUN0005.net**: Network trained 35 runs.
- **LOS2k.net**: Network trained for min. testing RMS error.
- **LOS2.dat**: Data file with 5 records.
- **LOS2.dat**: Run file with 5 records (Adjusted data file).
- **LOS2k.dat**: Output file for the run file.

Figure 2: Files used by the application

NetMaker preprocessing

The original file is the data file, *LOS.dat*, which you will find at the distribution diskette, in the `c:\BrainMaker\` folder if you have copied it to your own computer or in the section *Datafiles* to your left. This file can be read by NetMaker *(Figure 3).* The original file contains different types of variables represented in a way which cannot be used by BrainMaker without modifications. Codes of categorical variables must be transformed such that each category is represented as a separate binary variable. The first categorical variable is the primary diagnosis, **PrimDiag**, which has a numeric code for each diagnosis. The diagnosis with the highest code value, 35, is of course not 35 times as serious as the one with the lowest code value 1. The following procedure will initiate NetMaker to convert the coded, categorical variable into as many binary variables as there are different codes for the categorical variable:

Figure 3: NetMaker *LOS.dat*
The steps are:

1. Read LOS.dat into NetMaker.
2. Click Manipulate data in the menu appearing.
3. Mark column PrimDia.
4. Select Symbol in the toolbar.
5. Select Split column into symbols.

NetMaker will name the new variables according to a convention suggested in the Split menu and give each new binary variable a symbolic name, for example PrimD22. Note that because of the restricted width of the NetMaker columns, you will not in this case see the last character of the names. To avoid this, you must choose a shorter name when NetMaker provides its suggestion.

---

**Figure 4: NetMaker LOS2.dat**

The codes of PRIMDiag will be substituted by 15 new variables with symbolic names. The original PrimDia variable can be deleted. There are 2 more variable of the same type, namely Admit# which is a code for how the patient was admitted and Heredity. From the original table, we see that Admit# can have 5 codes which are given different symbolic names (Figure 4). Note that if the categorical variable uses names instead of values. Finally, the variable Heredity with three named categories must be processed likewise. NetMaker is able to take care of all the necessary transformations.

The next preparation is to assign the columns as Input, Output (pattern) and Annotations/Not used. Note that in BrainMaker, output is called pattern. Pattern is, however, by others authors also used about the input, the input pattern, and to avoid confusion we refer to the output pattern as output variables. Assignment is done by marking a column (click on its name) to be assigned and selecting Label in the NetMaker toolbar. Figure 4 demonstrates how each column is specified as an input column, an output (pattern) column or an annotate column. The resulting
table should now be named LOS2.dat and saved. You can check your results with the file Los2.dat in Datafiles.

The last 2 task in NetMaker pre-processing is to produce the default definitions of the neural network and 3 files for BrainMaker. The first task is done by reading the LOS2.dat by clicking File/Preferences (Figure 5). This opens for certain options. At this stage, we select the default specifications divided randomly the file of the 64 available cases into 2 files, one for training with 90% of the cases, and one for testing with 10% of the cases. For the second and final task we select File/Create BrainMaker Files accept default names for the definition, the training and for the test files.

![Figure 5: Specifying BrainMaker files](image)

**Figure 5: Specifying BrainMaker files**

**BrainMaker specifications**

A BrainMaker definition file is produced as the last preprocessing step by NetMaker. It can be read by NotePad, and **stripped**, or **expanded**. We start discussing a stripped version of LOS.def as shown in (Figure 6). The first row indicates that the input is represented with 1 example per line and with 31 input variables. If we compare with the LOS2.dat file in Figure 4, this file had only 10 input columns. The explanation is that the categorical variables are all expanded to have one binary variable for each category, i.e. Heredity is expanded to 3, Gender to 2, Primary to 15 and Admit to 5 which makes 25 plus 6 input variables satisfactory represented. Note that NetMaker takes care of the expansion to the correct number of binary variables. The second line indicates that there is only 1 output variable.

The next line in the definition file specifies that we want a multi-layer network with 31 hidden neurons between the 31 input sources and the 1 output neuron. The following 2 rows give the names of the training and the test files which we specified in NetMaker. Finally, 2 pairs of rows with scaling parameters terminate the specification. The first set with pair of values specifies the **minimum** and the **maximum** values in the training set for each input variable. The first variable, Diags, which is a numeric variable for the number of diagnoses, has for example a minimum
value 1 and a maximum value 7. Age of the patients varies from minimum 1 to maximum 64 years. All binary variables typically have 0 as minimum and 1 as maximum value. The second set with pair of values specifies the minimum and maximum values for the output variable LOS which in our application varies from 7 to 182. These minimum and maximum values are used to scale each variable to values between 0 and 1 internally during processing. This reduces the problem of varying variable ranges during the processing. The output is re-scaled again before displayed. Of course, in a real application, we should at least allow a shorter stay in hospital and correct the minimum from 7 to, say, 1 day.

You can study the complete definition file as generated by NetMaker by reading LOS2.def by means of either Notepad or directly from Datafiles. Note that if some parameters are missing, for example the Learn rate, it means that BrainMaker will use default values if the parameters are not included later.

Let us look what happens when the NetMaker prepared definition file is read by BrainMaker. Figure 7 shows the first part of the BrainMaker display of LOS2.def. The first row contains the name of the training file, LOS2.fct, and two important parameters. One is the Learning rate which determines the size of adjustments used by the training algorithm. The default value is 1.00. The learning rate can be changed by clicking Parameters/Learning Setup in the BrainMaker toolbar. The option Constant Learn Rate is marked and specified with value 1.00. At this point we only note there is a possibility of making a change.

A high Learning rate will usually result in a faster training, but there is a risk that the weights may be adjusted too much and the best combination of weight values is passed. With a small Learning rate, the weights will slowly be adjusted to the training examples, and the possibility to find a good set of weight values is improved.

The last parameter on the first row we observe the Training tolerance with default value 0.1. This means that if the predicted output value deviates with a fraction with less than +/- 0.1 from
the corresponding target output value, the predicted output value is considered correct. Tolerance can be reset by selecting the toolbar Parameters/Training control flow. Obviously, a wide tolerance will terminate training faster than a narrow tolerance, but the trained network may not be as useful as if the training tolerance was set at a more narrow tolerance. In classification problems, particularly those with only 2 output categories, the tolerance can be set wider than in regression problems in which the aim is to predict a value as near the target as possible.

The remaining of the BrainMaker display is a list of the names of all input variables and the output variable. As for the output variable we have to distinguish between the predicted output Out and the target output Ptn.

With only 58 patients and 31 input variables, 31 hidden neurons can certainly result in overfitting. The manual indicates how a guess for a more reasonable number can be made:

\[ f(31+0.6 + (31^2)/80)^{1/2} \approx 11 \text{ neurons} \]

By selecting Connections/Change Network Size, the menu shown is made available for changing the number of hidden neurons to 11.

Click OK.
If we click the toolbar at Connections/Change Network Size, a specification form for the NN topology appears [Figure 8]. The form allows changing the number of hidden layers and the number of neurons in each layer. In the LOS application, 1 hidden layer with 31 neurons is specified. If required, we could change both in this form. Looking carefully at the form, we can also see that the numbers of connections are listed. The connection from input to the hidden layer is 992 (31*31+31) and the number of connections from the 31 hidden neurons to the single output neuron is 32 (31+1). The 31 and 1 additional connections originate from the threshold neurons emitting always 1's to each of the 31 hidden neurons and to the 1 output neuron.

We want to test the network during the training to avoid that we 'overfit' the network to the training set. Several decisions must be made. First, when testing while training, it is reasonable to use the same tolerance for testing as for training and to change testing tolerance from default 0.4 to 0.1 [Figure 9]. In Parameters/Training Control Flow, the part on testing, we mark that we want to test the network on the test file after each training iteration. That means that after each iteration, the RMS etc. can be computed both for the training set and for the independent testing set. Since there is no known way to 'sense' a global minimum in the performance as exhibited by the RMS curve for the test set, it is a good strategy to save the trained network periodically. It would require a lot of capacity to save after each run. We decide that saving after every 5th run can be satisfactory. It means that we may have to make up to 4 additional training interactions from the saved network up to the one which according to the recorded test RMS seems to be the best. The right hand side of the Parameters/Training Control Flow form concerns the criteria to stop the training. Default is when all training records in pass the tolerance requirement We decide to use that. As seen, other options are available.

![Set training control](image)

**Figure 9: Set training control**

Before we can start the training, we must specify how and where the training progress information should be saved. In toolbar option File/Select fact files, check that the form contains LOS2.fct and LOS2.tst which were the names given to the 2 files in NetMaker, and that the training box is marked. Under File/Training statistics File/Test statistics, we mark each of them
in turn and accept the proposed names with the extensions, .tst and .sta, respectively. In these 2 files, we get data logged which can later be analyzed in order to find the best final network.

**Training the network**

We are now ready to start training. Click *Operate/Train Network* on the toolbar of BrainMaker. The network will start training and you will see the display as in Figure 10. There will be 2 windows open, the main window informing about the current run and a graphic display of the training progress. The last overlap the first and can be moved freely around.

![Ready for training](image)

**Figure 10: Ready for training**

The 2 first rows of the first window give continuous information during training about the last of the 58 facts read from in the training file, the total of facts read, the number of bad facts read so far in this run and the number of bad facts in the last run, the number of good facts read so far of the current run, the number of good facts in the previous run and the number of iterations or runs through the training set.

The second window contains 2 graphs presenting the progress of the network training. The upper histogram indicates the absolute value of the deviation between predicted and target (ptn) output value divided by target value. The staples represent the number of cases in the training deviate there are at the different relative deviations. During a successful training, the cases will be moved to the left. With a Training tolerance 0.1 the training would be considered completed when all cases are at the left side of the vertical 0.1 line.

The lower graph measure the RMS for the deviations of all cases of the training set in the last completed run. In a successful training the curve will be falling down toward the right, and approach zero as the predictions become identical with the target values. The displayed window in Figure 11 indicates that the required training tolerance was obtained after 93 runs. The RMS was about 0.4 at the start of the training and less than 0.1 when training stopped.
Figure 11: Training progress

Analysis of training

After saving the trained network, *LOS2.net*, we quit BrainMaker and return to NetMaker. Read the training statistics file, *LOS2.sta*, click *Operate/Select Columns to Graph*. You get a form up in which you click *Choose x-axis* and then mark the column *Run* in the table. *Column 1* and the name *Run* appear in the form. Then mark column *RMS error* and this will appear as *Track 1* on the form. If you have trouble, cancel the form and start up the selection again. When satisfied, click *Make graph* and you will get a second form in which you may make further specifications. At this point only click *Make plot* and you will get the display of (Figure 12). This plot is the same as the progress graph in BrainMaker, but gives a more detailed view of the RMS curve.

Figure 12: Detailed training view
We have pointed out that training until all examples are perfectly solved within the training tolerance specified, frequently lead to bad predictions because the network has learned the details of the training set without being able to generalize. For this reason, we run the network on the test set after each run while training. The test statistics were saved in the file LOS2.tst. By repeating the plotting procedure described above, but now after reading the LOS2.tst file, we obtain the graph displayed in Figure 13. We recall that the examples in the test set have not been used for training and that the 8 examples are new to the network. The displayed plot can therefore be considered as a measure of how well the network predicts LOS for future patients. The most interesting information from the plot is that the network which was trained for 96 iterations was far from the best. Inspection of the plot indicates the network gave the best generalization, i.e. minimum RMS for the test set, after 36 runs of training.

![Continuous testing](image)

**Figure 13: Continuous testing**

We might now have decided to start the training again and let it run for 36 iterations. However, when training is initiated, it starts with a new random set of weights. Empirical experience shows that the learning progress depends on the initial set of random weights. In a second training, the best results might have been obtained after a different run than the 36th. For this reason we have specified BrainMaker to save the network after every 5th run during the training process. The trained network after the 35th run is thus saved and available as Run00035.net. We load this network and run another training iteration to obtain the 'optimally' trained network. This network we name LOS2b.net, which is used for the remaining part of our discussion.

The final procedure is:

1. read file Runxxxx, where xxx is the last run below the test RMS curve minimum,
2. set Parameters/Flowcontrol form to stop after Run y, where y is the run at which the test RMS curve has its minimum, and
3. start training by Operate/Train network.
From the histogram in Figure 14, we can see that the selected network LOS2b.net was able to predict 4 of the examples in the test set within the test tolerance 0.1 in our example. All except one of the cases in our test file within a tolerance of 0.2. The test set contains, however, only 8 examples and the present exposition must therefore be considered as an illustration.

Figure 14: 36th run network

By selecting Edit in the BrainMaker toolbar and clicking on Connection matrices, the individual weights can be studied. The window is, however, small and a better solution is to open LOS2b.net in NotePad/WordPad (Figure 15). The 2 weight matrices in our application are displayed with a header row. In this you will find the number of layers (including the input sources!), number of threshold neurons per layer, number of input sources in layer 1, number of neurons in the 1 hidden layer, and the number of neurons in the output layer. The header layer is followed with the weight matrices. The next row (can be wrapped by NotePad into several lines) is the weights from each of the 31+1 input sources (the last is the weight from the threshold neuron) to the first hidden neuron. There are 11 such rows, one associated with each hidden neuron. This first matrix is followed by the second composed by the weights from the hidden neurons, 11+1, to the output neuron. The weight from the threshold neuron is always the last of the row.
Running the network in production

After the network is trained and testing indicates that it predicts independent records satisfactory, it is ready for use in production runs. We reserved 5 records of patients in file LOS4.dat. These have neither been used during training nor in testing the network. These were pre-processed in the same way as the training and testing data with the exception that we did not include the length of stay variable as output target (pattern) because this is the variable we want to predict (Figure 16). When the preprocessing is finished, we produce a BrainMaker run file from NetMaker by selecting File/Create a Running Fact file. The BrainMaker run file is named LOS4.in (Figure 17).
To prepare BrainMaker for the production run (Figure 18):

1. Load network LOS4b.net.
2. Specify Running file as LOS4.in on the form obtained from File/Select fact files.
3. Click Select File/Write Facts To file and name the output file LOS2b.out. The production run is now specified.
4. Select Operate/Run Trained Network.

The run will be finished very rapidly.

You can study the output predictions by starting NotePad and read the file LOS2b.out (Figure 19). Five blocks of data is displayed. In the header of each block is the Patient number. In the
following lines are the input values displayed. The predicted output value is printed on a separate row.

Figure 19: LOS predictions

A trained network is frequently required as a component of an information system. The weight matrices which we studied in the previous section can easily be copied and embedded as a module of a larger system. Prediction can easily be implemented with a few simple matrix routines.

The BrainMaker manual contains many good advises for improving the networks, and it is well worth to study the examples given in the manual (Figure 20).

Figure 20: Improvements

The manual proposes several ways to improve the application:
• Design other networks and compare the performance.
• Use alternative parameters and data flow controls.
• Get more facts

For further details, read Chapter 5 and experiment with the LOS data files available in the BrainMaker directory and at the Floppy disk.

Figure 20: Improvements
Financial application

Our second regression application example is based on the tutorial in Chapter 4 of the BrainMaker manual. We also use other data files which will be developed during our discussion. You should find them all in Datafiles. The purpose of these exercises is to design and train a neural network for predicting the price changes of a stock. Predictions like these are popular applications, and many networks have been trained for this and similar purpose.

We start by discussing the data file, Price1.dat, which you can find on your BrainMaker distribution disks or in Datafiles. This file has 10 columns. The first column contains a day index indicating time. Time is an important element of this application. The second column contains the values of the price variable, BD100, of the stock BD. It is the changes in this price we want to predict. The remaining 8 columns represent input variables by means of which we will try to predict the changes when the network has been trained.

By means of NetMaker, the file PRICE1.dat must be preprocessed to a form which we can use. The preprocessing operations are:

- differences in the column BD100 between day t and day t-1, have been computed and shifted down 4 rows from day t to t+4. These are the values in the column BD+4 to be predicted,
- variable names ending with a D indicates that differences have been computed between day t and day t+1,
- variable names ending with -4 are variable columns of which have been shifted up 4 rows, i.e. from day t to t-4, compared with the original
- BDAvg2 is a variable computed as the BD average between day t and day t-1
- the first columns labeled annotate, the BD+4 column is labeled pattern (target), and the remaining input

The result is the file Price4.dat (Figure 21). Detailed discussion of analysis and data
Prerequisites

It should be noted that the Price4.dat file was produced by means of NetMaker from the original file Price1.dat using preprocessing functions:

- Operate/Make Graph
- Operate/Difference Column
- Operate/Cyclic Analysis
- Column/Shift Columns Up
- Column/Shift Columns Down
- Operate/Repeat
- Row/Delete
- Column/Delete
- Operate/Moving Average
- Label...
- Operate/Data Correlator

Figure 22: Prerequisites

Preference options

Select File>Preferences. We accept the default to save 10%, i.e. 18 observations, for testing. We still have 158 rows for training.

Figure 23: Preference options
Specifying BrainMaker files

We name the training file Price4.fct and the test file Price4.tst. All unmarked fields (the last 10) is marked input.

The rows are Row/Row/Rows, and File/Create BrainMaker Files.

Figure 24: Specifying BrainMaker files

Cleaning up the Price4.def

The Price4.def has by default a number of data which are only used to display current processing. With a reasonably fast processor, following the displays is impossible. For more efficient processing, the definition file can be cleaned up as shown.

Figure 25: Cleaning up the Price4.def

preprocessing can be found in the BrainMaker manual Chapter 4. The Price4.dat file is almost ready for being transformed to BrainMaker files (Figure 22). The last operation is to partition the initial data into training and test files (Figure 23). This is specified by clicking File/Network Creation Preferences before finishing by clicking File/Create BrainMaker Files. The training file is marked Price4.fct and the test file Price4.tst (Figure 24). Make certain that the created files are stored in c:\BrainMaker or some other place you can retrieve it from. Please also note that we use different files than the BrainMaker tutorial.

The NetMaker generation of BrainMaker files results in 3 files, training, testing and definition files. The definition file, Price4.def, starts by specifying the number of input variables, 21, and then lists a dictionary with name labels for all input variables. A name label is also given to the
output variable. One hidden layer with 21 neurons is specified, followed by display specifications, etc.

The definition file can be simplified by removing unwanted specifications as shown in Figure 25. You may want to change the default 21 neurons in the hidden layer. We used 16 hidden neurons, and selected Connections/Change Network Size from the menu (Figure 26).

![Figure 26: The neuron topology](image)

In the example, a test of the trained network was specified to be performed after each run, that every 5th network should be saved, and default file names of the statistics files were accepted (Figure 28). The training was set to be stopped after run 1000 if the training was not completed at an earlier run.
In our application, the training of the network went on until the run 1000 stop criterion was met. Figure 29 indicates that the training expressed by the RMS has flattened out and there seems to be no reason for continuing. By means of NetMaker, the plot of Price4.sts can be studied in detail and RMS can be seen to be about 0.045.

However, more interesting is the study of the test statistics file, Price4.sta, which indicates the performance of the trained network applied to the independent test data. Figure 30 shows the plot of these statistics. The plot indicates that the best RMS is about 0.089 which appears about run number 488. After this run, the RMS for testing is increasing, a symptom of over fitting.

We saved each 5th version of the trained network and were able to return to version 485 (Figure 31). This version of the network was read back into BrainMaker and trained for 3 more runs. The
Figure 30: Testing RMS

The RMS error went down to 0.0093 in the 488th run.

Figure 31: Versions of trained networks

We use network version Run00488.net to compute Run00488.net which we rename Price4a.net.

Figure 30: Testing RMS

Figure 31: Versions of trained networks

trained version of the network after run 488 seemed to perform best according to the test file. This version was finally saved as Price4a.net and used in the following analysis.

The test file was also run separately with recording of each individual prediction (Figure 32). The options from File/Write Facts to File of the BrainMaker toolbar are many, and some will be investigated more in detail.

Different versions of the BrainMaker software have different capabilities. The Professional version offers an interesting possibility to study the sensitivity or influence of the different input variables on the target variable. Figure 33 shows average negative/positive effects of input variables on the output variable by bars to the left/right of the vertical line. The effects are measured by automatic variation of each input variable within pre-specified limits and observing
Figure 32: Prepare individual predictions

We want to study the individual predictions. Read in Price4a.net, and prepare for a separate test run with recording of the predictions.

Figure 33: Sensitivity diagram

BrainMaker Pro has a feature which permits a systematic analysis of the sensitivity of each of the input variables. Bars to the left/right indicate negative/positive sensitivity. The sensitivity is proportional to the length of the bar.

The bars can be considered as graphic ‘regression’ coefficients.

Figure 34: Sensitivity diagram

The change in the target variable. The relative size of the effects is expressed by the length of the bar. In the prediction model we trained, the input variables BDavg2 and price 1D had the strongest positive effects, while BD100D and Ind1-1 had the strongest negative effects. More precise charts can be obtained from BrainMaker (Professional) toolbar Analysis/Sensitivity

Analysis shown in Figure 34. The sensitivity measurement is still primitive since no account is made for simultaneous effects.

Based on the sensitivity analysis, we may ask if there are variables with a disturbing effects included among the 21 input variables, in other words, can we obtain better results if we
eliminate any of the input variables? We selected the most influential input variables from Figure 35 for training a smaller network with only 8 input variables. The network was defined by

\[\text{Figure 34: Sensitivity figures}\]

\[\text{Revised model}\]

\[\text{Figure 35: Revised model}\]

\[\text{Price4r.def}\] and the trained net were saved as \text{Price4r.net}. The number of hidden neurons was reduced to 10. This is a high number compared with the 8 input variables, but a few experimental runs indicated that it was a good specification.

A repeated sensitivity analysis is presented in Figure 36 and Figure 37. In general, the effects of the remaining input variables seemed to be strengthened, and one, Price2D, had changed sign.
We can easily compare the predictive power of the two prediction models for example by computing the simple linear regression between predictive and target values for the 18 cases in the test file. Figure 38 shows that the smaller model is the superior for predicting the changes in the stock price. The explanation is most likely that some of the variables in the larger model have an disturbing effect on the weights of the trained model resulting in inferior predictions.

What about increasing the number of hidden layers? We tried a network with 2 hidden layers with 6 and 4 neurons (3 layers with 6, 4 and 1 neurons), respectively. The result was further improvement as shown in Figure 39. The structure may be improved by further experimentation with the number of neurons in the hidden layers, etc.
Figure 38: Comparing performances

The extreme point deserves attention.

Figure 39: Multi-layer network

We discussed the extreme result at the SouthEast quadrant during the Lecture. Here is another network, 2 hidden layers with 8 and 4 neurons. It solves the extreme problem, and gives a correlation $R = 0.848584$.

Figure 40 shows the results. The predictions using this regression equation on the test file data compared with the multi-layer network discussed in the preceding paragraph is displayed in Figure 41. As we see, the simple linear regression predicts better results than the complex non-linear neural network. Why?
Figure 40: Linear regression

Regression/network

As shown, the multiple linear regression gives an even higher correlation than the multi-layer non-linear network. One possible explanation may be that the artificial data were generated by means of a linear model.

Figure 41: Regression network

Exercises

a. In Figure 8 about the LOS network connections, 11 hidden neurons in 1 layer are used. Try to specify 2 hidden layers with 10 neurons in each, train the network, test the trained network and compared the results with those reported in this session. Discuss the factors which may cause the differences.

b. It is frequently argued that NN do better classification than regression. Look up the original LOS.dat file and investigate the distribution of the continuous target variable, LOS. You will find that the minimum value is 7 and the maximum 182. Try to divide the range into 10 categories. Redefine LOS as a categorical variable with categories referred to by the symbols 0 to 9 (or the letters A to J). Apply the knowledge you acquired from Session 5 to train and run a Neural
Network which predicts the probabilities that each example belongs to the different categories. Do you get any interesting results? What kind of precision do you obtain?
Session 8: Imputation

Small area statistics

Our societies are rapidly becoming more dependent on detailed information about the socioeconomic state and development for small areas and/or groups. On the other hand, demands for more detailed statistics can often not be served by traditional data collection and processing because of the associated high costs. Many requests for the 2000 Population Census preparations indicate needs, which the national statistical offices will not be able to serve.

Censuses are frequently supplemented with sample surveys, to obtain statistics too expensive to be collected on a complete basis in the census itself. Unfortunately, traditional estimation methods will not always provide reliable results for areas or groups if the areas are small and/or samples from these are below a certain size. In this session, we illustrate how useful small areas/groups statistics may be provided by methods studied in previous sessions. For a more comprehensive discussion of the experiments, see Nordbotten 1996, Nordbotten 1999 and .pdf file.

Data available

Two experiments were performed on Norwegian population data from the 1990 Population Census. Data for two municipalities, Municipality I with 17,326 individuals distributed to 56 census tracts, and Municipality II with 10,102 individuals distributed to 44 census tracts, were used in for the experiments. We shall focus our attention on imputation estimates for these 90 small areas.

The 2 selected municipalities differ in several respects. Municipality I is located in the middle part of the country near a city and with a mix of farming, manufacturing and transport as its main industries. Municipality II is located in the northern part of the country. Fisheries and fish processing are its main industries. The average census tract size of the Municipality I was 310 inhabitants while the average size of the tracts in Municipality II was 230 inhabitants in 1990.

For most municipalities in Norway, survey observations were collected from samples of the inhabitants in addition to census data available for each individual. The two municipalities used in the study required, however, statistics based on complete counts also for the survey variables and paid Statistics Norway for the additional observations themselves.

In the experiments, we simulated that a simple random sample survey of 2,007 individuals was taken also in Municipality I and that no survey observations were made at all in Municipality II (Figure 1). Neural networks were trained on the data from the sample of Municipality I to impute survey variable values for individuals in Municipality I not included in the sample, and for all individuals in Municipality II. Because complete survey data existed for both populations, an excellent basis existed for testing small area imputation estimates and their predicted accuracy.
Sizes of census tracts

The majority of census tracts in Municipality I have from 100 to 300 inhabitants with an average of 310, a few tracts have more than 1000 inhabitants. Because of the skew distribution, many of the tracts would be represented with 10-20 individuals in a simple sample survey of 2000 individuals. For these tracts, traditional estimators could not be expected to provide useful statistics.

The tracts in Municipality II have even smaller populations than tracts in the first municipality. The average tract has 230 inhabitants. Out of the 44 tract, 32 have less than 200 inhabitants and 13 less than 100. In our experiments, no sample survey was assumed at all for this municipality, and traditional estimates could therefore not be computed at all.

Variables, imputations and mse

The census provided individual values for a large number of variables and these were supplemented by a rather extensive survey. In this session, we focus the attention on 2 categorical variables from the survey (Figure 2). These were transformed to 15 binary (symbolic in BrainMaker terminology) variables. From the census, 97 variables were used.
**Figure 2: Category of survey variables**

Two neural networks representing 15 simultaneous imputation functions were used to impute 15 variable values for each inhabitant (**Figure 3**). The first network included a set of 9 imputation functions and provided individual probabilities for each binary variables representing the main Cohabitation for each individual (only the main cohabitation was used), while the second network provided probabilities for a set of 6 binary variables representing the main Means of transportation categories. The variables with the highest probabilities in each set were set equal to 1, while the remaining binary variables were set equal to 0. Both networks used the individual values of the 97 census variables as independent variables. In addition, both networks included 25 latent or hidden variables.

**Figure 3: Network variables**

Two experiments were carried out. In the first, 2 random and mutually exclusive samples were drawn from the population in Municipality I. We assumed that the survey was carried out in both
samples. Sample 1 counted 1,845 individuals. Data from this sample were used to train the two networks with 5,240 weights. Sample 2 comprised only 165 individuals and its data were used to estimate the mse of the imputed variables. These samples had together approximately the same relative sample size used in the 1990 Census for most other municipalities (Figure 4).

**Sample sizes**

<table>
<thead>
<tr>
<th>Total population:</th>
<th>N = 17,326</th>
</tr>
</thead>
<tbody>
<tr>
<td>Sample 1:</td>
<td>n₁ = 1,845</td>
</tr>
<tr>
<td>Sample 2:</td>
<td>n₂ = 162</td>
</tr>
<tr>
<td>Sample 3:</td>
<td>n₃ = 15,319</td>
</tr>
</tbody>
</table>

Figure 4: Sample sizes

<table>
<thead>
<tr>
<th>Cohabitation variable mse</th>
<th>Sample 2</th>
<th>Sample 3</th>
</tr>
</thead>
<tbody>
<tr>
<td>Cohabitation</td>
<td>rmse₂</td>
<td>rmse₃</td>
</tr>
<tr>
<td>Nobody</td>
<td>0.249</td>
<td>0.251</td>
</tr>
<tr>
<td>Spouse</td>
<td>0.163</td>
<td>0.164</td>
</tr>
<tr>
<td>Cohabitant</td>
<td>0.199</td>
<td>0.200</td>
</tr>
<tr>
<td>Children</td>
<td>0.270</td>
<td>0.271</td>
</tr>
<tr>
<td>Parents</td>
<td>0.188</td>
<td>0.189</td>
</tr>
<tr>
<td>Siblings</td>
<td>0.222</td>
<td>0.223</td>
</tr>
<tr>
<td>In-laws</td>
<td>0.054</td>
<td>0.054</td>
</tr>
<tr>
<td>Gr.children/gr.parents</td>
<td>0.061</td>
<td>0.061</td>
</tr>
<tr>
<td>Other</td>
<td>0.174</td>
<td>0.175</td>
</tr>
</tbody>
</table>

Figure 5: Cohabitation variable mse

An imputation estimate of each total was computed as the sum of the observed variable values for each of the 2,007 inhabitants in the two samples and the imputed variable values for the remaining 15,319 individuals in Municipality I.

Means and rmse for the 15 survey variables were computed from Sample 2. The rmse from Sample 3 were used for evaluation. The results of the comparisons of rmse's for the Cohabitation
variables are presented in Figure 5. The figure indicates that the rmse's from Sample 2 are good estimates.

**Imputation estimates for Municipality I**

Both producers and users of statistics wish to identify which estimates belong to the high accuracy estimates. The producers need a tool for providing quality declarations while the users need accuracy declarations for evaluating the usability of the estimates for their particular applications.

Altogether 840 Y'-totals were estimated and corresponding target Y totals computed for the 56 census tract areas in Municipality I. For each imputation estimate, the imputation error RMSE was predicted. The following accuracy policy was assumed:

- estimates errors |Y'-Y| larger than 5 persons are regarded as unacceptable, and
- users are willing to take the risk that 1 out of 4 estimates were incorrectly rejected.

With this risk, the policy implies rejection of an imputation estimate if RMSE*1.15>5, i.e if its imputation error RMSE >4.3. To test the validity of using the RMSE-s for predicting the accuracy of the imputation estimates, the 504 cohabitation estimates were classified as rejectable if RMSE>4.3, if not they were classified as acceptable. Because the real totals were available in the experiment, the imputation estimates were cross tabulated by their real deviation from the targets. The results are given in Figure 6.

![Table: Predicted/observed accuracy. Cohabitation. Municipality I](image)

**Figure 6: Predicted/observed accuracy. Cohabitation. Municipality I**

Two types of errors, well known from statistical theory of testing, can be used for the discussion of accuracy prediction. From assumptions made, we would expect that the number of Type I errors, rejecting incorrectly estimates that satisfy the requirement, would be less than 126 estimates or 25% of the estimates. Figure 6 indicates that 51, or only 10%, were incorrectly classified not to satisfy the condition. Type II errors, accepting incorrectly estimates that do not
satisfy the condition, were 67. Figure 6 shows that 430 Cohabitation imputation estimates for small areas were predicted acceptable. Out of these, 363 were correctly predicted while 57 were incorrectly predicted as not acceptable. If Type II errors are considered relatively serious or expensive, a constant less than 4.3 will reduce the number of these errors, but at the same time also increase the number of Type I errors.

For the Means of transport totals, Figure 7 shows that 283 estimates were predicted to deviate from the targets with 5 or less individuals. The Type II errors committed were 48 estimates incorrectly predicted to meet the condition. The Type I errors made were only 9 or less than 3% of the 336 estimates. These figures indicate that accuracy predictions based on imputation errors can be reliable for applications of the type considered.

<table>
<thead>
<tr>
<th>Predicted/observed accuracy. Means of transportation Municipality I</th>
</tr>
</thead>
<tbody>
<tr>
<td>Observed</td>
</tr>
<tr>
<td>----------</td>
</tr>
<tr>
<td>Predicted</td>
</tr>
<tr>
<td>&lt;=5</td>
</tr>
<tr>
<td>&gt;5</td>
</tr>
<tr>
<td>Sum</td>
</tr>
</tbody>
</table>

Figure 7: Predicted/observed accuracy. Means of transportation. Municipality I

For release of survey statistics, Statistics Norway used a publication rule for the 1990 Census results stating that only estimates with coefficients of variation less than 0.3 should be published in printed tables. In the experiments, a similar requirement was tested and gave results for the imputation estimates similar to those reported above and far better than those obtained in the census publication [Nordbotten 1996].

Imputation estimates for Municipality II

In the second experiment, Municipality II was assumed not to be surveyed at all. The neural networks trained in Municipality I were assumed to have a general validity and used to impute individual survey variable values for all inhabitants in Municipality II. The individual imputations were aggregated to 660 imputation estimates for population totals in the 44 census tracts of the municipality.

Figure 8 shows that 396 Cohabitation imputation estimates were computed. Out of these, 315 estimates were predicted to have an acceptable accuracy out of which 30 were incorrectly accepted. 81 imputation estimates were predicted not to meet the requirement of an error of 5 or
less. The accuracy prediction failed to accept 44 estimates, which were incorrectly rejected. Of the 396 imputation estimates, 83% had the required accuracy.

**Figure 8: Predicted/observed accuracy. Cohabitation. Municipality II**

The accuracy of the estimates for *Means of transportation* totals are reported in Figure 9. 88% of estimates had errors of 5 or less individuals. The accuracy predictions classified 231 of imputation estimates correctly as acceptable or not. The figure indicates that 33 estimates were incorrectly classified as acceptable when they should have been rejected or were rejected when they should have been accepted.

**Figure 9: Predicted/observed accuracy. Means of transportation. Municipality II**

This experiment assumed that no survey data were collected in Municipality II. No neural network could therefore be trained in this municipality. It was assumed that the relationships between survey and census data were shared by the municipalities. Imputations networks derived
from data for Municipality I were used to impute all \( y \)-variable values for each individual in Municipality II. The imputation estimates computed for Municipality II included therefore no observed values. Taking into account that these estimates were completely based on individually imputed values, both the real accuracy and the prediction accuracy are remarkable. It is even more remarkable taking into account that the neural networks and the \( rmse \) used were borrowed from a very different municipality.

**Extreme individual errors**

For *Cohabitation* totals in Municipality I, the largest deviation between an estimate and the corresponding target total was for the number of individuals living alone in a tract with a total population of 1,046 persons. The target total for people living alone was 248, which in the imputation estimate was underestimated with 26 individuals.

For *Means of transportation* totals, larger differences were identified. Estimates for the number of people reported to use bicycle as a means of transportation to work demonstrated great deviations from their targets in some areas. In one tract, the target total of inhabitants who used bicycles as their means of transportation to work was 165. The imputation estimator underestimated this total with 66 inhabitants. The explanation seemed to be that the topology permitted few other means of transportation than bicycle in this tract, which was not reflected well in the imputation network trained at data from another municipality.

**Four statements needing further research**

- Imputation networks can be trained to impute individual survey values from census data. Trained networks can subsequently be used for imputing individual survey variable values for non-sampled individuals using census data as input to the neural networks. The available observations and the imputed values can be added up to imputation estimates for population totals.
- **Imputation errors** can be computed as \( RMSE \) using the \( rmse \) for residuals between individual imputed and target values from an independent sample.
- Reliable statistics based on imputation estimates can be used for areas too small for traditional estimation.
- Imputation networks developed for one municipality can in some applications be applied also in other municipalities.

**Exercises**

_a_. In session 9, we used the file *IncR.dat* as starting point for investigating accuracy and generating BrainMaker files. We developed the network *IncR.net* for imputing individual variable values. In the fact files, a categorical variable named *Region* was used to derive the 4 symbolic variables *reg1* to *reg4*. Assume that these are 4 small areas in an area represented by the 100 records in *IncRB.tst*. They have an average size of 25 individuals. Investigate the real number of inhabitants represented in each small area.

_b_. I have decomposed *IncRB.tst* into 4 subfiles named: *IncRB1.dat, IncRB2.dat, IncRB3.dat* and *IncRB4.dat* which you will find in *Datafiles*. Create BrainMaker test files (0% training files!).

131
Use the network *IncR.net* to test each of the four files. Use the *rmse* in *IncRA.sta* estimated from the validation sample to make accuracy predictions for the imputation estimates of the income totals for the four small areas.

**c.** Run *individual* imputations for each of the small areas, *sum up* imputed and target values separately and *compare* the predicted and real deviations. How successful were your predictions?

**d** How do traditional estimates for the four areas compare with your imputation estimates?
Session 9: Optimization

Additional software from CSS

California Scientific Software offers an additional package called the Genetic Training Option. This additional software contains two programs which can be used in combination or individually to find the best neural networks topology within a specified set or subject to specified rules. The first program permits training of a number of networks with different parameters within specified ranges. Each trained network is tested and ranked by performance.

The second program represents a genetic evolution of networks. Starting with a pair of parents, some of their weights are 'mutated' and by 'cross-over' their weights are copied to a child which is tested. The best trained networks are saved for further evolution, comparison and use.

Since the Student version of BrainMaker does not include these features, we describe in this session the general structures of the programs, demonstrate their use. Interested students are recommended to contact CSS and try out the programs themselves. In the present session, the students are therefore only required to study the data files.

The Genetic Training Option

The GTO can be considered as a package which calls the BrainMaker program as a subroutine to do the necessary training. When opened, the program displays a toolbar with File as the only option. Clicking the File, we get a menu with the following items:

- Select Network File
- New Optimizer
- Open Optimizer File
- (Save Optimizer File)
- New Genetic
- Open Genetic File
- (Save Genetic File)
- About GTO
- Exit GTO

The 2 options in parentheses, appears only when a relevant file has been created or opened.

Use of GTO requires an already trained NN saved as a .net file, e.g. LOS2.net, which has first to be opened by File/Select Network File. The next step will be to specify a new optimizer or genetic file. After specification, the Optimizer/Genetic file can be saved and re-opened later for execution.

Optimization of networks

We will start to discuss the Optimizer program. For illustration, we use the stock price change application discussed in Session 6. (Figure 1). The task is to find the 'optimal' network within the set
Figure 1: Searching for the optimal network

defined by hidden neurons in layer 1 between 3 to 6, and in layer 2 from 6 and 10 neurons. The total number of networks in the set, which must be trained and evaluated, is therefore $4 \times 5 = 20$.

The package GTO uses BrainMaker’s features with no interference required from the user. After starting GTO, we select the option File/New Optimizer File from GTO’s toolbar. Next step is to open a network .def or .net file. Under the option Optimize/File, the network we want to work with is the Price4r.net. The training and test files are Price4r.fct and Price4r.tst. From experience in Session 6, we limit the number of runs through the training examples to 200 runs for each alternative. We decide to test after each run. The program itself proposes .ACC as extension for the output file. (Figure 2)
The next option is *Optimize/Hidden Neurons*. As shown in Figure 3, we decide to limit out investigation to the range from 6 to 10 hidden neurons in the first layer, and 3 to 6 neurons in the second hidden layer. Note that we do not need to investigate all combinations within the specified ranges. If we had selected 2 in the field for *Steps* for the first hidden layer, 6, 8 and 10 neurons would have been chosen. That would have resulted in a set of 12 instead the 20 we specify.

![Specifying alternatives](image1)

**Figure 3: Specifying alternatives**

![Results of the GTO optimization](image2)

**Figure 4: Results of the GTO optimization**

In the lower part of the form, we set the conditions for introducing a new neuron. These conditions are always subject to the previously specified condition that no more than 200 runs should be used.
When we are satisfied with specifications, the training is started. Each alternative network is trained, evaluated and statistics recorded and ranked according to the criterion selected (Figure 4). In the figure, the different networks are ranked according to increasing RMS and we see that the networks with 8 hidden neurons in the first layer and 5 in the second seem to be superior.

We can repeat the same regression analysis of the individual predicted compared with the target values as we did in session 6. From Figure 5, we see that the optimum model performs better than any of the other.

Figure 5: Comparison

The GTO network is only slightly better measured by the correlation coefficient than the 3 layer network with 6 and 4 hidden neurons reported in image 20 in Lecture 5.

Measured by the RMSE the difference is larger as may be seen from the graph.

Other optimization options

The GTO also have options for alternative training rates, smoothing coefficients, tuning strategies, etc. All can be done with testing while training and evaluation of results based on an independent test data set.

From an overall point of view, it seems efficient to start investigating alternatives with GTO. When promising alternatives are identified recording networks after every 8th runs is a good strategy.

Figure 6: Other optimization options

Other options could be used under Optimize. They were left as specified in Price4r.net, but there are possibilities to investigate the effect of setting different tolerance limits, changing learning rate, etc. before the file is opened in GTO (Figure 6).
Genetic training

The second approach to find an 'optimal' network uses ideas from genetic evolution theory (Figure 7). As for biological organisms, networks with good, but different characteristics are used for 'breeding' new networks. If a descendant network inherits superior properties from parent networks, the new network will be used in future 'breeding' of networks. The question is of course how the breeding is carried out.

Introduction to GTO evolution

The GTO extension provides three more options inspired from biology:

1. Mutation
2. Crossover
3. Combined mutation and crossover

Mutation is evolution of a sequence of single networks with changing neurons. In crossover a couple of parent networks produces a child which inherits neurons from both parents and the best of the 3 becomes a new couple of parent networks which produce a new child network, etc. In combined mutation and crossover the child network can in addition get mutated neurons.

Figure 7: Introduction to GTO evolution

Again two concepts from biological evolution theory is used. First, mutation in a sequence of self-reproducing networks is considered. The weights for connections to a neuron can be regarded as corresponding to DNAs in a real life cell (Figure 8). Mutations in NN mean that weights are changed, usually in a random selection. The BrainMaker GTO permits a detailed specification of the mutation process.
The second concept borrowed from biological evolution theory is crossover. Crossover requires that each 'child' network has 2 parent networks (Figure 9). The child network inherits some of its neurons from one of the parents, the remaining from the other.

It is also possible to create more complex, genetic schemes by combining mutation and crossover. As indicated in Figure 10, we can use the GTO to investigate if we can breed better character recognition networks than obtained in Session 5. We loaded the GTO program and selected File/Select Network File and specified the alfa.net developed in Session 5 followed by New Genetic. This last specification opens the toolbar option Genetics. In this we started by specifying the General Setup. We decided that 200 runs should be performed for each network, that 20 generations should be reproduced and that the 5 best should be saved (Figure 11).
Combined mutation and crossover

For our discussion, we chose the most advanced option, combining mutation and crossover, to illustrate genetic evolution.

We make use of the pattern recognition net, alfa.net, developed in Lecture 7 for recognizing the 12 first alphabetic characters, and use the alfa.net and alfa.tet files. In that application, we managed to classify 21 out of 24 images correctly.

Figure 10: Combined mutation and crossover

Image 23 of Lecture 7 suggested that we may solve the problem of classifying 1 and j correctly by changing the individual weights. Genetic development changes the individual weights and saves networks which are best through generations of networks.

As in all experimentation, we must first specify the conditions for the genetic evolution.

March 19, 2006
Soen Sinhbotan
10

next option was specification of mutation and crossover rates. We specified these as 10 and 30, respectively. In the application, we had 15 hidden neurons and 12 output neurons, which meant that a mutation rate of 10 in average changed 2.7 neurons. The crossover rate of 30 meant that the child inherited 70 pct of the neurons from the parent ranked as the best and got 30 pct. crossover from the second parent. We specified that the same neuron can both be a crossover and a mutation.

Specification of crossover and mutation processes was done in the next two forms in Figure 12. In the previous form, it was specified that about 9 neurons should be crossed over from the second parent. Of these 9 we determined that 50 percent, in average 4.5 neurons
Figure 12: Mutation and crossover parameters

would get all their weights from the second parent. The remaining 4-5 neurons would get their weights from the first parent.

The 50% setting of the neurons which received crossover was more complicated since it was an expectation of a random variable. The result of the settings was that in average 50% of the weight difference were passed from each of the parents. However, since the weight difference was a random variable, in some cases it was more than 50%, in other cases less. The bandwidth 0.25 indicates however, that the variation range of the crossover variable was small (Figure 13). The mutation model is about the same.

Figure 13: Measuring genetic training quality

Finally, the GTO provides options to evaluate the produced network. MSE is selected in our example.
We ran the evolution specified for 20 generations. After the specified generations were generated, GTO produced the results in Figure 14. The 5 best networks were saved and listed with their squared correlation coefficients. The best, saved as GTO0001.net, had a squared correlation coefficient equal to 0.9840 corresponding to $R=0.9920$.

For each generation, one new network evolves, i.e., 20 different networks have been evaluated.

Figure 14: 5 best networks

Results

In images 17 and 18 of Lecture 7, we presented the results from applying sfa.net on the test sample and showed that there were 3 misclassifications.

The genetic network, GTO001.net applied on the test sample, gave an improved result with the following 2 classification errors:

Still, improvements can obviously be made for this network as outlined in Lecture 7. However, it is interesting to see that an evolution process can make an improvement which was very difficult to make manually as was shown in Lecture 7.

Figure 15: Results

It was interesting to make individual predictions for the test set. Figure 15 shows that the new neural network genetically developed recognized correctly 22 of the character pictures. This is an improvement compared with the findings in Session 5.
Exercises

a. GTO is not available in the Student version of BrainMaker. Do you see how the Optimizing approach described in this session could be implemented with repeated calls on BrainMaker? Try to make an outline of how such software could be implemented with algorithmic step-by-step instructions to a human computer operator.

b. If you succeeded in exercise a., try to make a similar design for Genetic network evolution.

c. If you have spare time, it is always interesting to investigate the results from the designs. You should by now have required experience to train a network of the type we have been studying.
Session 10: Other neural networks

Different types of neural networks

This course has been focusing on one particular type of neural networks, the fully connected feed-forward networks. In different contexts, other types have been mentioned such as recursive networks. In this last session, other types and their properties will be briefly surveyed. A taxonomy of neural networks due to [Lippmann] is categorizing the nets by their input and learning approach as in Figure 1. In the figure, we recognize some of the NN we have discussed.

**Figure 1: Lippman’s taxonomy**

as a Continuous-Valued Input nets associated with a Supervised Learning algorithm, while the pattern recognition tasks we have studied do not fit well into this scheme.

**Figure 2: Gurney’s taxonomy**
Gurney classifies neural nets by their main structure and their main tasks (Figure 2). His main categories are feed-forward, recurrent and competitive networks. The main tasks for the feed-forward category are classification and function interpolations (which we have called regression). The recurrent networks are well suited for associative memorizing tasks. An example of this type of tasks is training a network to recognize corrupted pictures/patterns. The character recognition application discussed in Session 6 and which we used a feed-forward net to implement, is an example of the type of tasks which could have been carried out by a recurrent network. The third type Gurney lists, is the competitive nets. Their application task is typically to identify clustering properties, i.e. how input patterns create clusters in the output space because of similarities. These nets are typically trained without supervision.

Gurney also points out that the activation functions used by a neural network are important for separating nets in different categories. In this course, we have only considered weighted sums of input,

\[ \text{argument}_j = \sum w_{ij}x_i \]

as activation function arguments. Many other possibilities exists, for example the sum of weighted products.

Lawrence has a third taxonomy in which she has categorized most of the network types we have discussed in this course.

In the following sections, some types of neural networks not discussed in the previous sessions, will be briefly surveyed.

**Simple linear networks**

We have mainly been discussing single and multi-layer feed-forward networks with step or sigmoid activity functions. The simplest activity function is, however,

\[ y_i = \sum w_{ij}x_i \]

It has the advantage of being extremely simple and has been used for certain tasks. An interesting aspect of multi-layer networks with this type of activation function is that they can always be reduced to a single layer network. The disadvantage is that there is strict assumption to the input patterns interrelationship (linearly separable and orthogonal).

Let \( W_1 \) be the weight matrix between the input ant the first layer of neurons, while \( W_2 \) is the weight matrix between the hidden layer and the output layer. \( X_k, Z_m \) and \( Y_n \) are the input, the hidden layer output and the final output vectors with dimensions \( k,m \) and \( n \), respectively. The two-layer network can be expressed by:

\[ Z_m = W * X_k \]
and

\[ Y_n = W^* Z_m \]

It is easy to see that substituting the hidden layer from the second equation using the first, an equivalent single layer network can be expressed as:

\[ Y_n = W_1^* W_2^* X_n \]

Network with this kind of simple linear networks, can be used for example to associate corrupt input patterns with correct output patterns.

**Incompletely connected feed-forward nets**

All nets we have worked with have been fully connected, i.e. there are connections between each neuron in one layer (or input source) to all neurons in the next layer. The category of feed-forward networks also contains incompletely connected net topologies. Figure 3 exemplifies an incompletely connected network.

*Figure 3: Incompletely connected networks*

Incompletely connected feed-forward networks can save training time and storage capacities, but deep knowledge of the application task is required for a successful approach. In input pattern or image recognition tasks in which we know that certain clusters of pixels are correlated to each other, these input sources can effectively be connected to the some of the neurons in the next layer, and not to others.

This type of networks has been applied for simulating visual tasks for which it is realistic to assume that there is focus on certain areas.
Multi-layer feed-forward networks with by-pass connections

A last type is feed-forward networks which contain by-pass connections. Figure 4 illustrates a multi-layer network with by-pass connections. By-pass connections have been used, but it uncertain under which condition they are superior to fully connected networks.

![Networks with by-pass connections](image)

**Figure 4: Networks with by-pass connections**

Application of this type of networks requires also much knowledge about the application task as well as the working of a by-pass net. With some modifications, it is possible to train both incompletely connected and networks with by-pass connection using BP.

**Associative memories**

The associative memory type of neural nets, is explained with the so-called Bi-directional Associative Memory (BAM) network in this session [Kosko 1988]. BAM can be conceived as a simple 2 neuron layer network fully interconnected by connections by which signals can move in both directions (Figure 5). The first layer, called the input layer, has as many neurons as there are input sources. In contrast to the feed-forward networks, each neuron in the input layer receives unweighted input from one and only one input source. The input layer neurons send their output to the neurons of the second layer, called the output layer, which in turn send their input back to the neurons of the first layer. The neurons of the input layer process their new inputs and return their output to the output layer neurons. This process continues until the outputs of the output layer do not change any more. Stable output values are considered the final output of the net. As we see, the BAM network is a recursive network. BAM is assumed to work with bipolar input and output values (-1 or 1).

The BAM network requires a supervised training algorithm. It very simple and required only a single run through the training set:

1. **Initialize** all weights between the 2 layers to 0
2. **Load** one example with input and output values from the training set into neurons of layer 1 and 2.
3. **Multiply** each input neuron value with each output neuron value with each other.
4. **Add** the product to the weight of the respective connection.
5. **Repeat** step 2 to 4 for all input patterns of the training set.

A trained BAM is **robust** in the sense that even if there is noise in the input pattern, the correct target can be successfully recognized. The number of neurons in both the input and the output layers must be at least equal to the number of different categories. The network in **Figure 5** will for example not be able to distinguish input patterns in more than 4 different categories.

**Figure 5: Bi-directional memory**

Consider the **character recognition** problem with 26 letters each represented by $8\times8=64$ pixels as an example. Applying BAM will require 64 neurons in the input layer and 26 neurons in the output layer. The single layer feed-forward networks will as we have seen also require at least the same number of neurons plus a threshold neuron.

Airlines are now experimenting with **identifying** passengers by their eye iris during boarding. One scenario may be that when you are checking in, a camera is taking a **picture** of your right eye. The picture is resolved into a grid of, say, $100\times100$ pixels and associated with the passengers name, etc.. During boarding, you again passes a camera, which takes a picture of your right eye and the network determines if you have checked in, and inform you about your final seat assignment. If the plane has a capacity of 250 passengers, a Bam network with 10,000 input neurons and at least 250 output neurons are trained during check-in. Such a network can easily be implemented.

However, at the moment it is probably unrealistic to believe that a Bam can be developed for a **large population** of several millions of individuals because of the size of the required **weight** matrix. One solution proposed is to develop a **system of smaller** BAMS. When the capacity of the first is exploited during training, a second independent Bam is established, and when it becomes
full, a third Bam is developed, and so on. In this approach, the total number of weights in the weight matrices (one for each BAM) can be limited. The price to pay during operation will be a sequential search through the BAMs which probably will require some preprocessing module.

Other well known associative memory nets are the Hopfield nets [Hopfield 1982]. The Hopfield nets have a single layer of neurons which are receiving input from input sources. The output of these neurons are sent back to the other neurons as new inputs, processed and returned again. When the outputs have finished to change, the values are considered the final outputs of the net. Also the Hopfield net makes use of a supervised learning algorithm. Usually the associative networks are used with binary inputs.

**Self-organizing maps**

Kohonen's Self-Organizing Maps (SOM) are neural networks belonging to the category of competitive networks [Kohonen 1977]. It is also a representative of the type of nets which can be trained unsupervised to discover clusters of similar input patterns.

A SOM has a set of input sources and a two-dimensional layer of neurons, called a Kohonen layer (Figure 6). The input sources transmit inputs to the Kohonen layer. This layer is frequently compared with the cerebral cortex contained in the human cranium [Freeman & Skapura 1991]. The cerebral cortex is characterized by the fact that if one neuron is excited by an input, the neighboring neurons are also excited. The neighborhood is predetermined at the start and is being reduced during training.

![Figure 6: Self-organizing map](image)

The SOM is trained from a set of input examples. Input values and the connection weights must be normalized. The unsupervised training algorithm can be outlined as:

1. Set initial learning rate, neighborhood limits, and small, random weights of the connections,
2. Read a random example from training set,
Determine winning neuron by minimum 'distance' between inputs and weights,
4. Update weights of winning neuron and its neighbors,
5. Adjust training rate and neighborhood limits,
6. Repeat 2-4 until output pattern becomes stable.

If there are clustering characteristics in training set, these will show after a number of iterations up as stable 'clouds' of points in a mapping of the output points (Figure 7), each cloud representing a category of input patterns. The map in (Figure 7) indicates 2 distinct clusters, but there are a few points between which need to be allocated either to one of the two categories or to a third cluster. The map may therefore be further processed using a program which decides the limits for each cluster on the map.

![Final SOM output map](image)

**Figure 7: Final SOM output map**

An illustration of application is the representation of animal categories. With input patterns including properties for, training a SOM resulted in a map in which each animal was represented by a certain location on the map in such a way that related animals were closer to each other than less related animals.

**Adaptive Resonance Theory**

The Adaptive Resonance Theory, ART, is particularly connected to the name of Grossberg. The ART networks try to solve a general set of problems, and the theory is complex. In this course, we note that the ART networks are competitive and are trained unsupervised.

**Exercises**

a) Read Chapter 7: Neural Network Models, in Lawrence.
A bibliography for further studies


Reed, R.D. and Marks, R.J. II (1999): Neural Smithing: Supervised Learning in Feed Forward Artificial Neural Networks. MIT Press.


