KAPITAŁ LUDZKI
NARODOWA STRATEGIA SPÓJNOŚCI

Politechnika Wrocławska

UNIA EUROPEJSKA
EUROPEJSKI
FUNDUSZ SPOŁECZNY

ROZWÓJ POTENCJAŁU I OFERTY DYDAKTYCZNEJ POLITECHNIKI WROCŁAWSKIEJ

Wrocław University of Technology

Internet Engineering

Jacek Mazurkiewicz

SOFTCOMPUTING

Wrocław 2011

Projekt współfinansowany ze środków Unii Europejskiej w ramach
Europejskiego Funduszu Społecznego

Wrocław University of Technology

Internet Engineering

Jacek Mazurkiewicz

SOFTCOMPUTING

Wrocław 2011

Copyright © by Wrocław University of Technology
Wrocław 2011

Reviewer: Tomasz Walkowiak

ISBN 978-83-62098-23-1

Published by PRINTPAP Łódź, www.printpap.pl

Table of contents

Part 1. Multilayer Perceptron 6
1.1. Theoretical Background 7
1.1.1. Introduction 7
1.1.2. Learning Modelling 8
1.1.3. Activation Function 10
1.1.4. Learning Rule 10
1.1.5. MLP Learning Algorithm 12
1.2. List of Problems 15
1.3. Phases of Laboratory Exercises 15
1.4. Hints for the List of Problems 16
Part 2. Kohonen Neural Network 20
2.1. Theoretical Background 21
2.1.1. Introduction 21
2.1.2. Retrieving Phase of Kohonen Neural Network Algorithm 21
2.1.3. Classic Learning Algorithm for Kohonen Neural Network 22
2.2. List of Problems 24
2.3. Phases of Laboratory Exercises 24
2.4. Hints for the List of Problems 24
Part 3. Hopfield Neural Network 28
3.1. Theoretical Background 29
3.1.1. Hopfield Neural Network 29
3.1.2. Retrieving Phase 30
3.1.3. Hebbian Learning Algorithm 31

3.1.4. Delta-Rule Learning Algorithm 31
3.1.5. Pseudoinverse Learning Algorithm 32
3.2. List of Problems 32
3.3. Phases of Laboratory Exercises 32
3.4. Hints for the List of Problems 32
Part 4. Genetic Algorithms 36
4.1. Theoretical Background 37
4.1.1. Introduction 37
4.1.2. Encodings and Optimisation Problems 38
4.1.3. Genetic Algorithm – Selection, Mutation, Recombination 39
4.1.4. Crossover Operation Details 43
4.2. List of Problems 44
4.3. Phases of Laboratory Exercises 44
4.4. Hints for the List of Problems 44
Part 5. Expert Systems 47
5.1. Theoretical Background 48
5.1.1. Knowledge and its Representation 48
5.1.2. Members of Expert System Development Team 50
5.1.3. Structure of the Rule-Based Expert System 52
5.1.4. Expert System Characteristic 52
5.1.5. Forward Chaining and Backward Chaining 56
5.1.6. Conflict Resolution 59
5.1.7. Metaknowledge 60
5.1.8. Advantages of Expert System 61
5.2. List of Problems 61
5.3. Phases of Laboratory Exercises 62

5.4. Hints for the List of Problems 63
Part 6. Fuzzy Logic 64
6.1. Theoretical Background 65
6.1.1 Fuzzy Set Theory 65
6.1.2. Fuzzy Set Operations 69
6.1.3. Fuzzy Logic 70
6.1.4. Fuzzy Knowledge – Based Systems 72
6.1.5. Development of Fuzzy Expert Systems 80
6.1.6. Tuning Fuzzy Systems 80
6.2. List of Problems 81
6.3. Phases of Laboratory Exercises 81
6.4. Hints for the List of Problems 81
References 83

Part 1. Multilayer Perceptron

1.1. Theoretical Background

1.1.1. Introduction

The Multilayer Perceptron is an example of an artificial neural network that is used extensively for the solution of a number of different problems, including pattern recognition and interpolation. It is a development of the Perceptron neural network model, that was originally developed in the early 1960s but found to have serious limitations.

Artificial Neural Networks (ANNs) attempt to model the functioning of the human brain. The human brain for example consists of billions of individual cells called neurons. It is believed by many (the issue is contentious) that all knowledge and experience is encoded by the connections that exist between neurons. Given that the human brain consists of such a large number of neurons (so many that it is impossible to count them with any certainty), the quantity and nature of the connections between neurons is, at present levels of understanding, almost impossible to assess. The issues as to whether information is actually encoded at neural connections (and not at the quantum level for example, as argued by some authors – see Roger Penrose “The Emperor's New Mind"), is beyond the scope of this course. The assumption that one can encode knowledge neutrally has led to some interesting and challenging algorithms for the solution of AI problems, including the Perceptron and the Multilayer Perceptron (MLP).

Neurons can be modelled as simple input-output devices, linked together in a network. Input is received from neurons found lower down a processing chain, and the output transmitted to neurons higher up the chain. When a neuron fires, it passes information up the processing chain. This inherent simplicity makes neurons fairly straightforward entities to model. It is in modelling the connections that the greatest challenges occur. When real neurons fire, they transmit chemicals (neurotransmitters) to the next group of neurons up the processing chain alluded to in the previous subsection.

These neurotransmitters form the input to the next neuron, and constitute the messages neurons send to each other. These messages can take one of two different forms:

– excitation – excitatory neurotransmitters increase the likelihood of the next neuron in the chain to fire,

– inhibition – inhibitory neurotransmitters decrease the likelihood of the next neuron to fire.

If we can model neurons as simple switches, we model connections between neurons as matrices of numbers (called weights), such that positive weights indicate excitation, negative weights indicate inhibition. How learning is modelled depends on the paradigm used.

1.1.2. Learning Modelling

Using artificial neural networks it is impossible to model the full complexity of the brain of anything other than the most basic living creatures, and generally ANNs will consist of at most a few hundred (or few thousand) neurons, and very limited number of connections between them. Nonetheless quite small neural networks have been used to solve what have been quite difficult computational problems. Generally Artificial Neural Networks are basic input and output devices, with the neurons organized into layers. Simple Perceptrons consist of a layer of input neurons, coupled with a layer of output neurons, and a single layer of weights between them (Fig. 1.1.).

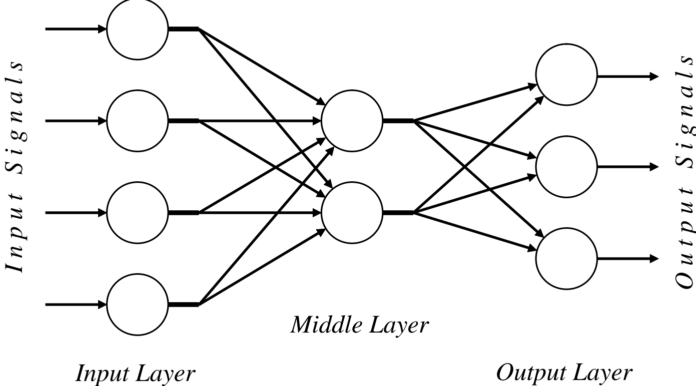


Fig. 1.1. Simple Perceptron Architecture

The learning process consists of finding the correct values for the weights between the input and output layer. The schematic representation given in (Fig. 1.1.) is often how neural nets are depicted in the literature, although mathematically it is useful to think of the input and output layers as vectors of values (I and O respectively), and the weights as a matrix. We define the weight matrix $\boldsymbol{W_{io}}$ as an $i \times o$ matrix, where i is the number of input nodes, and o is the number of output nodes. The network output is calculated as follows:

$$O = f(IW_{io}) \quad (1.1)$$

Generally the data is presented at the input layer, the network then processes the input by multiplying it by the weight layer. The result of this multiplication is processed by the output layer nodes, using a function that determines whether or not the output node fires. The process of finding the correct values for the weights is called the learning rule, and the process involves initialising the weight matrix to a set of random numbers between -1 and +1. Then as the network learns, these values are changed until it has been decided that the network has solved the problem. Finding the correct values for the weights is achieved by using a learning paradigm called supervised learning. Supervised learning is sometimes referred to as training. The some testing data is used to train the network, this constitutes input data for which the correct output is known. Starting with random weights, an input pattern is presented to the network, which makes an initial guess as to what the correct output should be. During the training phase, the difference between the guess made by the network and the correct value for the output is assessed, and the weights are changed in order to minimise the error. The error minimisation technique is based on traditional gradient descent techniques. While this may sound frighteningly mathematical, the actual functions used in neural networks to make the corrections to the weights are chosen because of their simplicity, and the implementation of the algorithm is invariably uncomplicated.

1.1.3. Activation Function

The basic model of a neuron used in Perceptrons and MLPs is the McCulloch-Pitts model, which dates from the late 1940s. This modelled a neuron as a simple threshold function:

$$f(x) = \begin{cases} 1 & x > 0 \\ 0 & x \leq 0 \end{cases} \quad (1.2)$$

This activation function was used in the Perceptron neural network model, and as can be seen this is a relatively straightforward activation function to implement.

1.1.4. Learning Rule

The Perceptron learning rule is comparatively straightforward. Starting with a matrix of random weights, we present a training pattern to the network, and calculate the network output. We determine an error function E:

$$E(O) = (T - O) \quad (1.3)$$

Where in this case T is the target output vector for a training input. In order to determine how the weights should change, this function has to be minimised. What this means is to find the point at which the function reaches its minimum value. The assumption we make about the error function is that if we were to plot all of its potential values into a graph, it would be shaped like a bowl, with sides sloping down to a minimum value at the bottom.

In order to find the minimum values of a function, differentiation is used. Differentiation is used to give the rate at which functions change, and is often defined as the tangent on a curve at a particular point. If our function is perfectly bowl shaped, then there will be only one point at which the minimum value of a function has a tangent of zero (i.e. have a perfectly at tangent), and that is at its minimum point.

In neural network programming the intention is to assess the effect of the weights on the overall error function. We can take (1.3) and combine it with (1.1) to obtain the following:

$$E(O) = (T - O) = T - f(IW_{io}) \tag{1.4}$$

We then differentiate the error function with respect to the weight matrix. The discussion on Multilayer Perceptrons will look at the issues of function minimisation in greater detail. Function minimisation in the Simple Perceptron Algorithm is very straightforward. We consider the error of each individual output node, and add that error to the weights feeding into that node. The Perceptron learning algorithm works as follows:

1. initialise the weights to random values in range [-1, 1],
2. present an input pattern to the network,
3. calculate the network output,
4. for each node n in the output layer...
   (a) calculate the error $E_n = T_n - O_n$,
   (b) add E_n to all of the weights that connect to node n (add E_n to column n of the weight matrix,
5. repeat the process from 2. for the next pattern in the training set.

This is the essence of the Perceptron algorithm. It can be shown that this technique minimizes the error function. In its current form it will work, but the time taken to converge to a solution (i.e. the time taken to find the minimum value) may be unpredictable, because adding the error to the weight matrix is something of a 'blunt instrument' and results in the weights gaining high values if several iterations are required to obtain a solution. This is akin to taking large steps around the bowl in order to find the minimum value. If smaller steps are taken, we are more likely to find the bottom.

In order to control the convergence rate and reduce the size of the steps being taken, a parameter called a *learning rate* is used. This parameter is set to a value that is less than one, and means that the weights are updated in smaller steps (using a fraction of the error). The weight update rule becomes the following:

$$W_{io}(t + 1) = W_{io}(t) + \varepsilon E_n \tag{1.5}$$

Which means that the weight value at iteration $t + 1$ of the algorithm, is equivalent to a fraction of the error εE_n added to the weight value at iteration t.

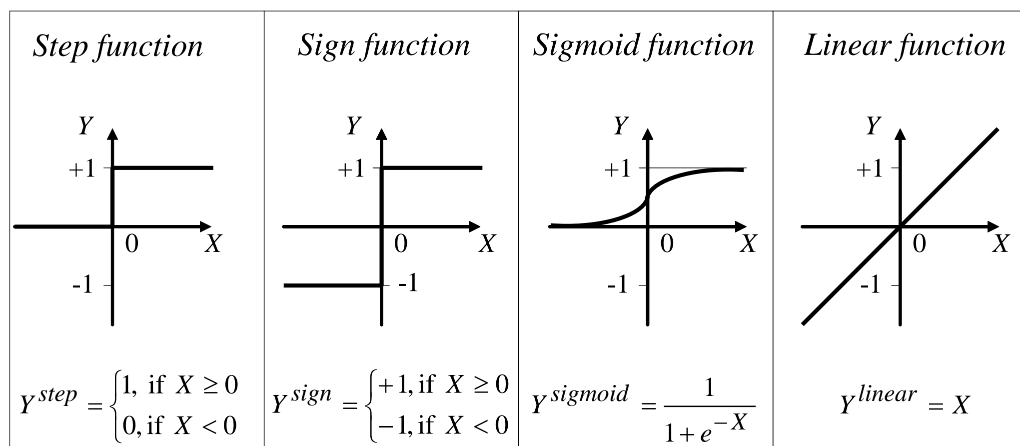
1.1.5. MLP Learning Algorithm

The principle weakness of the Perceptron was that it could only solve problems that were linearly separable. The simple Perceptron, based on units with a threshold activation function, could only solve problems that were linearly separable. Many of the more challenging problems in AI are not linearly separable however, and thus the Perceptron was discovered to have a crucial weakness, and returning to the problem of modeling logic gates, the exclusive-or problem (XOR) is in fact not linearly separable.
To obtain a bilinear solution we could add another layer of weights to the simple Perceptron model, but that brings the problem of assessing what happens in the middle layer. For a simple task such as the XOR problem, we could fairly easily work out what expected outputs for the middle layer of units should be, but finding a solution that would be completely automated would be incredibly difficult.

The essence of the supervised neural network training is to map input to a corresponding output, and adding an additional layer of weights makes this impossible, using the threshold function given in (1.2). A better solution to the problem of learning weights is to use standard optimisation techniques. In this case we identify an error function which is expressed in terms of the neural network output. The goal of the network then becomes to find the values for the weights such that the error function is at its minimum value. Thus gradient descent techniques can then be used to determine the impact of the weights on the value of the error function.

We need to have an error function that is differentiable, which means it should be continuous. The threshold function is not continuous, and so is unsuitable. A function that works in a similar way to the threshold function, but that is differentiable is the Logistic Sigmoid Function (Fig. 1.2.):

$$f(x) = \frac{1}{1+e^{-\lambda x}} \tag{1.6}$$



Hyperbolic tangent

$g(a) = \tanh(a)$

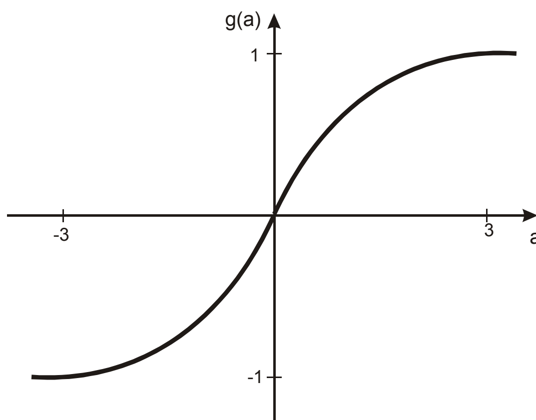


Fig. 1.2. Different types of activation function

Adjusting the value of λ to values of less than one make the slope shallower with the effect that the output will be less clear (more numbers around the centre of the graph, rather than clear indications of firing or not firing. Shallower slopes are useful in interpolation problems.

This function, when viewed in profile behaves in a very similar way to the threshold function, with x values above zero tending to one, and values below zero tending to zero. This function is continuous, and it can be shown that its derivative is as follows:

$$f_{lsf}^{'}(x) = f_{lsf}(x)(1 - f_{lsf}(x)) \quad (1.7)$$

Because the function is differentiable, it is possible to develop a method of adjusting the weights in a Perceptron over as many layers as may be necessary.

The basic MLP learning algorithm is outlined below. This is what you should attempt to implement.

1. Initialise the network, with all weights set to random numbers between -1 and +1.
2. Present the first training pattern, and obtain the output.
3. Compare the network output with the target output.
4. Propagate the error backwards.

    (a) Correct the output layer of weights using the following formula:

    $$w_{ho} = w_{ho} + (\eta \delta_o o_h) \quad (1.8)$$

    where w_{ho} is the weight connecting hidden unit h with output unit o, η is the learning rate, o_h is the output at hidden unit h. δ_o is given by the following:

    $$\delta_o = o_o(1 - o_o)(t_o - o_o) \quad (1.9)$$

    where o_o is the output at node o of the output layer, and t - o is the target output for that node.

    (b) Correct the input weights using the following formula:

    $$w_{ih} = w_{ih} + (\eta \delta_h o_i) \quad (1.10)$$

    where w_{ih} is the weight connecting node i of the input layer with node h of the hidden layer, o_i is the input at node i of the input layer, η is the learning rate. δ_h is calculated as follows:

    $$\delta_h = o_h(1 - o_h) \sum_o (\delta_o w_{ho}) \quad (1.11)$$

5. Calculate the error by taking the average difference between the target and the output vector. For example, the following function could be used:

$$E = \frac{\sqrt{\sum_{n=1}^{p} (t_o - o_o)^2}}{p} \tag{1.12}$$

where p is the number of units in the output layer.

6. Repeat from 2 for each pattern in the training set to complete one epoch.
7. Shuffle the training set randomly. This is important in order to prevent the network from being influenced by the order of the data.
8. Repeat from step 2 for a set number of epochs, or until the error ceases to change.

1.2. List of Problems

1. Traffic signs recognition using a Multilayer Perceptron.
2. Thermal pictures of human faces recognition by a Multilayer Perceptron.
3. Translation system based on a Multilayer Perceptron: pictures of characters and digits into a Braille alphabet signs.
4. Translation system based on Multilayer Perceptron: pictures of characters into a Morse alphabet signs.
5. Neural network as device for a function approximation.
6. Neural network as an inverted pendulum controller.

1.3. Phases of Laboratory Exercises

1. Check and improve the necessary softcomputing knowledge.
2. Prepare and collect the necessary data sets: for training and for testing.

3. Realise the necessary data preprocessing and/or data postprocessing using different types of ready-to-use software or by “hand-made” software prepared by the laboratory group.

4. Prepare your own software to implement the proper softcomputing algorithms. The main goal is to create the correctly working engine, user interface utilities are not so important. The software environments and systems you can use for implementation are limited, but the actual possibilities ought to be discussed with the laboratory supervisor.

5. Turn on and tune the prepared software engine, supply the input training and/or testing data. If the engine works correctly, check what happens when the starting point parameters change, explore the sensitivity of engine for the different sets of available parameters and find the solution of the problem. At the end, check if the used softcomputing solution is correctly fitted to the problem.

6. Prepare the final report including the following parts:

   – the short description of the problem with necessary assumptions,

   – definitions and descriptions of the training and testing sets of input data with description of the preprocessing procedures,

   – definitions and descriptions of the of output data with description of the postprocessing procedures,

   – description of the tuned topology and parameters of the prepared softcomputing engine,

   – detailed results analysis and final remarks.

1.4. Hints for the List of Problems

Problem No. 1

1. The graphical definitions of the road signs are described in the Traffic Law – use them.
2. Create the training and the testing tests by choosing the subset of signs from each category.

3. Set the uniform size – measured in pixels – for all traffic signs pictures, reduce the size as much as possible but remember to preserve the most important details of the picture.
4. Convert the traffic signs pictures to gray-scale or to black and white representation.
5. Each pixel is a single component of the input vector for Multilayer Perceptron, do not forget to normalise the pixel values.
6. Train the Multilayer Perception by the training set of pictures, check it using the testing set and corrupted pictures of the signs, discuss the type of corruptions with the laboratory supervisor.

Problem No. 2

1. Thermal pictures of human faces are available from the laboratory supervisor – use them.
2. Divide the available set of pictures into two separable sets: for training and for testing.
3. Set the uniform size – measured in pixels – for all pictures, reduce the size as much as possible but remember to preserve the most important details of the picture.
4. Convert the pictures to gray-scale representation.
5. Each pixel is a single component of the input vector for Multilayer Perceptron, do not forget to normalise the pixel values.
6. Train the Multilayer Perception by the training set of pictures, check it using the testing set and corrupted pictures of the signs, discuss the type of corruptions with the laboratory supervisor.

Problem No. 3

1. The Braille Alphabet signs represent the visible characters and digits by the set of convexities in special raster. The size of raster is not uniform – so it is necessary to choose the subset of characters and/or digits represented by the raster of the identical size.
2. Code the convexity/no convexity in the raster by binary representation: 1 or 0.
3. Choose the printable font of visible characters in black and white representation.

4. Divide the available set of pictures into two separable sets: for training and for testing.
5. Set the uniform size – measured in pixels – for all pictures of characters, reduce the size as much as possible but remember to preserve the most important details of the picture.
6. Each pixel is single component of the input vector for Multilayer Perceptron.
7. Train the Multilayer Perception by the training set of pictures, check it using the testing set and corrupted pictures of the signs, discuss the type of corruptions with the laboratory supervisor.

Problem No. 4

1. The Morse Alphabet signs represent the visible characters and digits by the vector of dashes and dots. The size of the vector is not uniform – so it is necessary to choose the subset of characters and/or digits represented by the vector of the identical size.
2. Code the dash/dot in the vector by binary representation: 1 or 0.
3. Choose the printable font of visible characters in black and white representation.
4. Divide the available set of pictures into two separable sets: for training and for testing.
5. Set the uniform size – measured in pixels – for all pictures of characters, reduce the size as much as possible but remember to preserve the most important details of the picture.
6. Each pixel is a single component of the input vector for Multilayer Perceptron.
7. Train the Multilayer Perception by the training set of pictures, check it using the testing set and corrupted pictures of the signs, discuss the type of corruptions with the laboratory supervisor.

Problem No. 5

1. Choose the trigonometric function, fix the domain and the calculate the values of the function. Prepare the samples of the function.
2. Use as training set the samples taken from the begin and from the end of the fixed domain.
3. The answer of the Multilayer Perceptron should restore the samples from the middle part of the fixed domain.

4. Check the distance between the correct function value and the value pointed by the Multilayer Perceptron.

Problem No. 6

1. The general idea is to substitute the real PID controller by Multilayer Perceptron based engine. The second Multilayer Perceptron should be used to emulate the inverted pendulum.
2. Use the results of well-known classic solutions of the inverted pendulum controller to train the MLP controller.
3. Take the set of well-known observations describing the real inverted pendulum to train the "MLP pendulum".
4. It is necessary to find and discuss the relations: velocity, force, angle, acceleration.

Part 2. Kohonen Neural Network

2.1. Theoretical Background

2.1.1. Introduction

Suppose that an input pattern has N features and is represented by a vector $\boldsymbol{x}$ in an n-dimensional pattern space. The network maps the input patterns to an output space. The output space in this case is assumed to be one-dimensional or two-dimensional arrays of output nodes, which possess a certain topological ordering. The question is how to train a network so that the ordered relationship can be preserved. Kohonen proposed to allow the output nodes interact laterally, leading to the self-organising feature map. This was originally inspired by a biological model. For example a random sequence of two-dimensional patterns can be mapped to an array of output nodes, with a preserved topology.

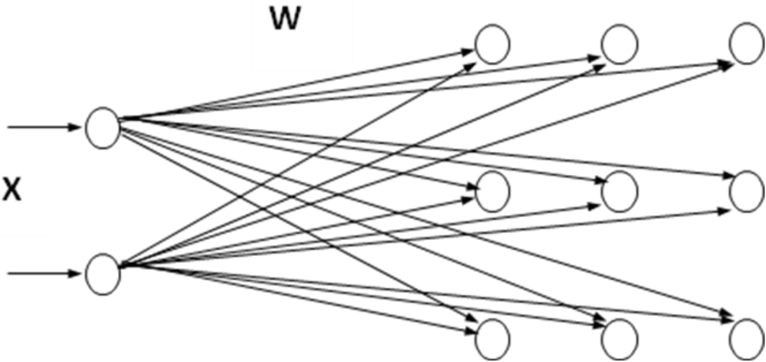


Fig. 2.1. Kohonen Neural Network

2.1.2. Retrieving Phase of Kohonen Neural Network Algorithm

During the retrieving phase all neurons from Kohonen map calculate the Euclidean distance between the weights and the output vector and the winner neuron is the one with the shortest distance. So each neuron from Kohonen map calculates the output value according to the classical weighted sum:

$$Out(i,j)=\sum_{l=0}^{N-1} x_l w_{lij} \tag{2.1}$$

where:

Out(i, j) – output value calculated by single neuron from Kohonen map indexed by *(i, j)* if Kohonen map is rectangular, for 1-D Kohonen map we have only single index *i*,

x_l – component of *N*-elements size input vector,

w_{lij} – weight associated with connection from component of input learning vector x_l and neuron indexed by *(i, j)* if Kohonen map is rectangular, for 1-D Kohonen map we have only a single index *i*.

2.1.3. Classic Learning Algorithm for Kohonen Neural Network

The most prominent feature is the concept of excitatory learning within a neighbourhood of the wining neuron. The size of the neighbourhood slowly decreases with each iteration. The selection of a winner can be modulated by the frequency sensitivity of the output nodes. The other possible way is to modulate the learning rate by the frequency sensitivity. It is hard to say which solution is better or more accurate. So for the discussion presented below we decided for the second possibility.

The learning algorithm is based on the Grossberg rule. All weights are modified according to the following equation:

$$w_{lij}(k+1)=w_{lij}(k)+\eta(k)\Lambda(i^w, j^w, i, j)(x_l-w_{lij}(k)) \tag{2.2}$$

where:

k – iteration index,

η – learning rate function,

x_l – component of input learning vector

w_{lij} – weight associated with connection from component of input learning vector x_l and neuron indexed by *(i, j)* if Kohonen map is rectangular, for 1-D Kohonen map we have only a single index *i*.

Λ – neighbourhood function, (i^w, j^w) – indexes related to winner neuron, (i, j) – indexes related to single neuron from Kohonen map

The learning rate η is a decreasing function, for presented discussion we assume a linear decreasing form. Learning rate function is responsible for the number of iterations – it marks the end of learning process. This way there isn't any factor to determine if number of iterations is satisfactory.

The neighbourhood function – often called Mexican Hat – Λ could be realised in many different ways. The main problem is to determine a group of neurons which are neighbours of the winner neuron. These neurons increase their output value during single learning step.

The maximum gain is related to the winner, if neighbourhood neuron is further to the winner this increase is less significant. Neurons which are not neighbours should decrease their output value or their output value should not change. The presented solution is based on the following description of the neighbourhood function:

$$\Lambda(i^w, j^w, i, j) = \begin{cases} 1 & for \quad r = 0 \\ \dfrac{\sin(ar)}{ar} & for \quad r \in (0, \dfrac{2\pi}{a}) \\ 0 & for \quad other\ values\ r \end{cases} \tag{2.3}$$

where:

a – neighbourhood parameter, can be changed during learning algorithm

r – distance from winner neuron to each single neuron from Kohonen map, calculated by indexes of neurons as follows: (2.4)

$$r = \sqrt{(i^w - i)^2 + (j^w - j)^2}$$

The learning procedure is iterative. The whole algorithm can be described by following steps:

1. All weights are initialised by random values generated from range (-1, 1).
2. The winner neuron for each learning vector is created by calculating the net output using random values of weights with ordinary Kohonen map retrieving algorithm.
3. All weights are modified using Grossberg rule (2.1) for single learning vector x_l using current value of learning rate function as well as current value of neighbourhood function assuming the proper winner neuron created in step 2.
4. The learning rate function value is modified, the neighbourhood parameter a (2.2) is modified and if the learning rate function value is greater than zero step 3 is executed for the next learning vector, else the learning algorithm stops.

2.2. List of Problems

1. Traffic signs recognition using a Kohonen Neural Network.
2. Thermal pictures of human faces recognition by a Kohonen Neural Network.
3. Picture compression using a Kohonen Neural Network.
4. Human recognition based on a fingerprint by a Kohonen Neural Network.

2.3. Phases of Laboratory Exercises

1. Check and improve the necessary softcomputing knowledge.
2. Prepare and collect the necessary data sets: for training and for testing.
3. Realise the necessary data preprocessing and/or data postprocessing using different types of ready-to-use software or by “hand-made” software prepared by the laboratory group.
4. Prepare your own software to implement the proper softcomputing algorithms. The main goal is to create the correctly working engine, user interface utilities are not so important. The software environments and systems you can use for

implementation are limited, but the actual possibilities ought to be discussed with the laboratory supervisor.

5. Turn on and tune the prepared software engine, supply the input training and/or testing data. If the engine works correctly, check what happens when the starting point parameters change, explore the sensitivity of engine for the different sets of available parameters and find the solution of the problem. At the end, check if the used softcomputing solution is correctly fitted to the problem.

6. Prepare the final report including the following parts:

– the short description of the problem with necessary assumptions,

– definitions and descriptions of the training and testing sets of input data with description of the preprocessing procedures,

– definitions and descriptions of the of output data with description of the postprocessing procedures,

– description of the tuned topology and parameters of the prepared softcomputing engine,

– detailed results analysis and final remarks.

2.4. Hints for the List of Problems

Problem No. 1

1. The graphical definitions of the road signs are described in the Traffic Law – use them.
2. Create the training and the testing tests by choosing the subset of signs from each category.
3. Set the uniform size – measured in pixels – for all traffic signs pictures, reduce the size as much as possible but remember to preserve the most important details of the picture.
4. Convert the traffic signs pictures to gray-scale or to black and white representation.
5. Each pixel is a single component of the input vector for Kohonen Neural Network, do not forget to normalise the pixel values.

6. Train the Kohonen Neural Network by the training set of pictures, check it using the testing set and corrupted pictures of the signs, discuss the type of corruptions with the laboratory supervisor.

Problem No. 2

1. Thermal pictures of human faces are available from the laboratory supervisor – use them.
2. Divide the available set of pictures into two separable sets: for training and for testing.
3. Set the uniform size – measured in pixels – for all pictures, reduce the size as much as possible but remember to preserve the most important details of the picture.
4. Convert the pictures to gray-scale representation.
5. Each pixel is a single component of the input vector for a Kohonen Neural Network, do not forget to normalise the pixel values.
6. Train the Kohonen Neural Network by the training set of pictures, check it using the testing set and corrupted pictures of the signs, discuss the type of corruptions with the laboratory supervisor.

Problem No. 3

1. Choose not too large picture recorded in grey scale.
2. Divide the picture into equal rectangular pieces. Each piece is a single training vector for the Kohonen Neural Network.
3. Each pixel is single component of the input vector for the Kohonen Neural Network, do not forget to normalise the pixel values.
4. Train step by step the Kohonen Neural Network using the pieces of the picture. Create this way the set of “neurons-winners” responsible for the pieces of the picture.
5. The weight vectors of the “neurons-winners” aggregate the data about the whole pieces of the picture. The lower number of the “neurons-winners” mean the better compression factor.
6. Calculate the compression factor as the function of number of pieces
7. Repeat the experiment with different pictures – try to choose the pictures characterised by variant dynamic scale.

Problem No. 4

1. Find the available fingerprint database. Choose the subset of fingerprint pictures – each person ought to be described by the same number of prints.
2. Divide the available set of pictures into two separable sets: for training and for testing.
3. Set the uniform size – measured in pixels – for all pictures, reduce the size as much as possible but remember to preserve the most important details of the picture.
4. Convert the fingerprint pictures to gray-scale or to black and white representation.
5. Each pixel is a single component of the input vector for a Kohonen Neural Network, do not forget to normalise the pixel values.
6. Train the Kohonen Neural Network by the training set of pictures, check it using the testing set and corrupted pictures of the fingerprints, discuss the type of corruptions with the laboratory supervisor.

Part 3. Hopfield Neural Network

3.1. Theoretical Background

3.1.1. Hopfield Neural Network

The binary Hopfield net has a single layer of processing elements, which are fully interconnected - each neuron is connected to every other unit. Each interconnection has an associated weight. We let w_{ji} denote the weight to unit j from unit i. In Hopfield network, the weight w_{ij} and w_{ji} has the same value. Mathematical analysis has shown that when this equality is true, the network is able to converge. The inputs are assumed to take only two values: 1 and -1. The network has N nodes containing hard limiting nonlinearities. The output of node i is fed back to node j via connection weight w_{ij}.

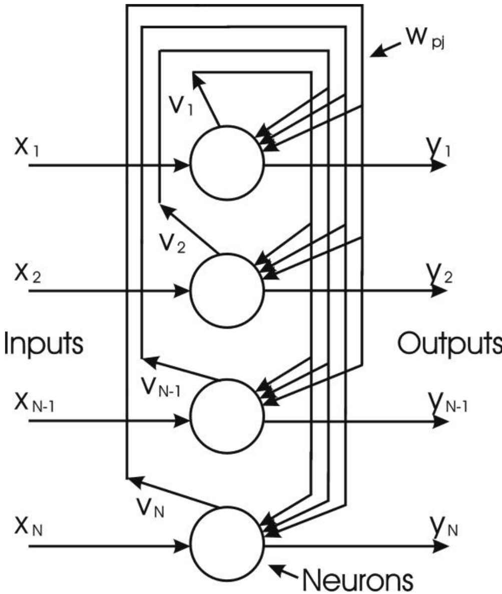


Fig. 3.1. Hopfield neural network

3.1.2. Retrieving Phase

During the retrieving algorithm each neuron performs the following two steps [2]:

Step 1 – computes the coproduct:

$$\varphi_p(k+1) = \sum_{j=1}^{N} w_{pj} v_j(k) - \theta_p \tag{3.1}$$

where:

w_{pj} – weight related to feedback signal,

$v_i(k)$– feedback signal,

Θ_p – bias,

Step 2 – updates the state:

$$v_p(k+1) = \begin{cases} 1 & for \quad \varphi_p(k+1) > 0 \\ v_p(k) & for \quad \varphi_p(k+1) = 0 \\ -1 & for \quad \varphi_p(k+1) < 0 \end{cases} \tag{3.2}$$

The process is repeated for the next iteration until convergence, which occurs when none of the elements changes state during any iteration:

$$\forall_p \quad v_p(k+1) = v_p(k) = y_p \tag{3.3}$$

The initial conditions for the iteration procedure require the following equation:

$$\forall_p \quad v_p(0) = x_p \tag{3.4}$$

The converged state of Hopfield net means the net has already reached one of attractors. An attractor is a point of local minimum of energy function (Liapunov function):

$$E(x) = -\frac{1}{2} \sum_{i=1}^{N} \sum_{j=1}^{N} w_{ij} x_i x_j + \sum_{i=1}^{N} \theta_i x_i \tag{3.5}$$

$$E(x) = -\frac{1}{2} x^T W \ x + \theta^T x \tag{3.6}$$

3.1.3. Hebbian Learning Algorithm

The training patterns are presented one by one in a fixed time interval. During this interval, each input data is communicated to its neighbour N times:

$$w_{ij} = \begin{cases} \frac{1}{N}\sum_{m=1}^{M} x_i^{(m)} x_j^{(m)} & \text{for } i \neq j \\ 0 & \text{for } i = j \end{cases} \tag{3.7}$$

The realisation of Hebbian learning algorithm is very easy, but the algorithm results in rather low capacity of the net:

$$M_{max} = 0{,}138\,N \tag{3.8}$$

where:

M_{max} - maximum number of training vectors,

M - number of training vectors

3.1.4. Delta-Rule Learning Algorithm

The weights are calculated in recurrent way including all training patterns, according to the following matrix equation:

$$W = W + \frac{\eta}{N}\left[x^{(i)} - Wx^{(i)}\right]\left[x^{(i)}\right]^T \tag{3.9}$$

where:

$\eta \in [0{,}7,\ 0{,}9]$ - learning rate,

N - number of neurons,

$\boldsymbol{W}$ - matrix of weights,

$\boldsymbol{x}$ - input vector.

The learning rate has the same influence on the training process as a learning rate that appeared with the multilayer networks. The learning process stops when the next training step generates the changes of weights which are less than the established tolerance ε.

The Delta-Rule learning algorithm provides very good robustness of the network for the failed input vectors and as good as possible capacity of the net – equal to the number of neurons:

$$M_{max} = N \tag{3.10}$$

3.1.5. Pseudoinverse Learning Algorithm

To calculate the proper values of weights we need the full set of the training vectors. The correct weight values means that the input signal generates itself as output and the converged state is available at once:

$$W X = X \tag{3.11}$$

One of the possible solutions can be found as follow:

$$W = X \left(X^T X\right)^{-1} X^T \tag{3.12}$$

The algorithm is sophisticated, but guarantees the robustness and capacity of the network at the level close to the net trained by the Delta-Rule algorithm:

$$M_{max} = N \tag{3.13}$$

3.2. List of Problems

1. Traffic signs recognition using a Hopfield Neural Network.
2. Thermal pictures of human faces recognition by a Hopfield Neural Network.
3. Human recognition based on a fingerprint by a Hopfield Neural Network.

3.3. Phases of Laboratory Exercises

1. Check and improve the necessary softcomputing knowledge.
2. Prepare and collect the necessary data sets: for training and for testing.

3. Realise the necessary data preprocessing and/or data postprocessing using different types of ready-to-use software or by "hand-made" software prepared by the laboratory group.

4. Prepare your own software to implement the proper softcomputing algorithms. The main goal is to create the correctly working engine, user interface utilities are not so important. The software environments and systems you can use for implementation are limited, but the actual possibilities ought to be discussed with the laboratory supervisor.

5. Turn on and tune the prepared software engine, supply the input training and/or testing data. If the engine works correctly, check what happens when the starting point parameters change, explore the sensitivity of engine for the different sets of available parameters and find the solution of the problem. At the end, check if the used softcomputing solution is correctly fitted to the problem.

6. Prepare the final report including the following parts:

   – the short description of the problem with necessary assumptions,

   – definitions and descriptions of the training and testing sets of input data with description of the preprocessing procedures,

   – definitions and descriptions of the of output data with description of the postprocessing procedures,

   – description of the tuned topology and parameters of the prepared softcomputing engine,

   – detailed results analysis and final remarks.

3.4. Hints for the List of Problems

Problem No. 1

1. The graphical definitions of the road signs are described in the Traffic Law – use them.
2. Create the training and the testing tests by choosing the subset of signs from each category.

3. Set the uniform size – measured in pixels – for all traffic signs pictures, reduce the size as much as possible but remember to preserve the most important details of the picture.
4. Convert the traffic signs pictures to black and white representation.
5. Each pixel is a single component of the input vector for a Hopfield Neural Network.
6. Train the Hopfield Neural Network by the training set of pictures, check it using the testing set and corrupted pictures of the signs, discuss the type of corruptions with the laboratory supervisor.
7. Compare the results of the recognition if the net is trained according to three basic training methods.

Problem No. 2

1. Thermal pictures of human faces are available from the laboratory supervisor – use them.
2. Divide the available set of pictures into two separable sets: for training and for testing.
3. Set the uniform size – measured in pixels – for all pictures, reduce the size as much as possible but remember to preserve the most important details of the picture.
4. Convert the pictures to black and white representation.
5. Each pixel is a single component of the input vector for a Hopfield Neural Network.
6. Train the Hopfield Neural Network by the training set of pictures, check it using the testing set and corrupted pictures of the signs, discuss the type of corruptions with the laboratory supervisor.
7. Compare the results of the recognition if the net is trained according to the three basic training methods.

Problem No. 3

1. Find the available fingerprint database. Choose the subset of fingerprint pictures – each person ought to be described by the same number of prints.
2. Divide the available set of pictures into two separable sets: for training and for testing.

3. Set the uniform size – measured in pixels – for all pictures, reduce the size as much as possible but remember to preserve the most important details of the picture.
4. Convert the fingerprint pictures to black and white representation.
5. Each pixel is a single component of the input vector for a Hopfield Neural Network.
6. Train the Hopfield Neural Network by the training set of pictures, check it using the testing set and corrupted pictures of the fingerprints, discuss the type of corruptions with the laboratory supervisor.
7. Compare the results of the recognition if the net is trained according to the three basic training methods.

Part 4. Genetic Algorithms

4.1. Theoretical Background

4.1.1. Introduction

Genetic Algorithms are a family of computational models inspired by evolution. These algorithms encode a potential solution to a specific problem on a simple chromosome-like data structure and apply recombination operators to these structures so as to preserve critical information. Genetic algorithms are often viewed as function optimizer, although the range of problems to which genetic algorithms have been applied is quite broad.

An implementation of a genetic algorithm begins with a population of (typically random) chromosomes. One then evaluates these structures and allocates reproductive opportunities in such a way that those chromosomes which represent a better solution to the target problem are given more chances to “reproduce” than those chromosomes which are poorer solutions. The “goodness” of a solution is typically defined with respect to the current population.

This particular description of a genetic algorithm is intentionally abstract because in some sense, the term genetic algorithm has two meanings. In a strict interpretation, the genetic algorithm refers to a model introduced and investigated by John Holland (1975) and by students of Holland (e.g., DeJong 1975). It is still the case that most of the existing theory for genetic algorithms applies either solely or primarily to the model introduced by Holland, as well as variations on what will be referred to in this book as the canonical genetic algorithm. Recent theoretical advances in modeling genetic algorithms also apply primarily to the canonical genetic algorithm (Vose, 1993). In a broader usage of the term, a genetic algorithm is any population-based model that uses selection and recombination operators to generate new sample points in a search space. Many genetic algorithm models have been introduced by researchers largely working from an experimental perspective. Many of these researches are application oriented and are typically interested in genetic algorithms as optimisation tools.

4.1.2. Encodings and Optimisation Problems

Usually there are only two main components of most genetic algorithms that are problem dependent: the problem encoding and the evaluation function.

Consider a parameter optimization problem where we must optimize a set of variables either to maximize some target, such as profit, or to minimize cost or some measure of error. We might view such a problem as a black box with a series of control dials representing different parameters; the only output of the black box is a value returned by an evaluation function indicating how well a particular combination of parameter settings solves the optimization problem. The goal is to set the various parameters so as to optimize some output. In more traditional terms, we wish to minimize (or maximize) some function $F(X_1, X_2, ..., X_M)$.

Most users of genetic algorithms typically are concerned with problems that are nonlinear. This also often implies that it is not possible to treat each parameter as an independent variable which can be solved in isolation from the other variables. There are interactions such that the combined effects of the parameters must be considered in order to maximize or minimize the output of the black box. In the genetic algorithm community, the interaction between variables is sometimes referred to as epitasis.

The first assumption that is typically made is that the variables representing parameters can be represented by bit strings. This means that the variables are discrete in an a priori fashion, and that the range of the digitising corresponds to a power of two. For example, with 10 bits per parameter, we obtain a range with 1024 discrete values. If the parameters are actually continuous then this digitisation is not a particular problem. This assumes, of course, that the digitisation provides enough resolution to make it possible to adjust the output with the desired level of precision. It also assumes that the digitisation is in some sense representative of the underlying function.

If some parameter can only take on an exact finite set of values then the coding issue becomes more difficult. For example, what if there are exactly 1200 discrete values which can be assigned to some variable X_i. We need at least 11 bits to cover this

range, but this codes for a total of 2048 discrete values. The 848 unnecessary bit patterns may result in no evaluation, a default worst possible evaluation, or some parameter settings may be represented twice so that all binary strings result in a legal set of parameter values. Solving such coding problems is usually considered to a part of the design of the evaluation function.

Aside from the coding issue, the evaluation function is usually given as part of the problem description. On the other hand, developing an evaluation function can sometimes involve developing a simulation. In other cases, the evaluation may be performance based and may represent only an approximate or partial evaluation. For example, consider a control application where the system can be in any one of an exponentially large number of possible states. Assume a genetic algorithm is used to optimize some form of control strategy. In such cases, the state space must be sampled in a limited fashion and the resulting evaluation of control strategies is approximate and noisy (c.f., Fitzpatrick and Grefenstette, 1988).

The evaluation function must also be relatively fast. This is typically true for any optimization method, but it may particularly pose an issue for genetic algorithms. Since a genetic algorithm works with a population of potential solutions, it incurs the cost of evaluating this population. Furthermore, the population is replaced (all or in part) on a generational basis. The members of the population reproduce, and their offspring must then be evaluated. If it takes 1 hour to do an evaluation, then it takes over 1 year to do 10000 evaluations. This would be approximately 50 generations for a population of only 200 strings.

4.1.3. Genetic Algorithm – Selection, Mutation, Recombination

The first step in the implementation of any genetic algorithm is to generate an initial population. In the canonical genetic algorithm each member of this population will be a binary string of length L which corresponds to the problem encoding. Each string is sometimes referred to as a “genotype” (Holland, 1975) or, alternatively, a

"chromosome" (Schaffer, 1987). In most cases the initial population is generated randomly. After creating an initial population, each string is then evaluated and assigned a *fitness* value.

The notion of *evaluation* and *fitness* are sometimes used interchangeably. However, it is useful to distinguish between the *evaluation function* and the *fitness function* used by a genetic algorithm. The *evaluation function*, or *objective function*, provides a measure of performance with respect to a particular set of parameters. The *fitness function* transforms that measure of performance into an allocation of reproductive opportunities. The evaluation of a string representing a set of parameters is independent of the evaluation of any other string. The fitness of that string, however, is always defined with respect to other members of the current population.

In the canonical genetic algorithm, fitness is defined by: f_i / f where f_i is the evaluation associated with string i and f is the average evaluation of all the strings in the population. Fitness can also be assigned based on a string's rank in the population (Baker, 1985; Whitley, 1989) or by sampling methods, such as tournament selection (Goldberg, 1990).

It is helpful to view the execution of the genetic algorithm as a two stage process. It starts with the current population. Selection is applied to the current population to create an intermediate population. Then recombination and mutation are applied to the intermediate population to create the next population. The process of going from the current population to the next population constitutes one generation in the execution of a genetic algorithm. Goldberg (1989) refers to this basic implementation as a Simple Genetic Algorithm (SGA).

We will first consider the construction of the intermediate population from the current population. In the first generation the current population is also the initial population. After calculating f_i / f for all the strings in the current population, selection is carried out. In the canonical genetic algorithm the probability that strings in the current

population are copied (i.e., duplicated) and placed in the intermediate generation is proportional to their fitness.

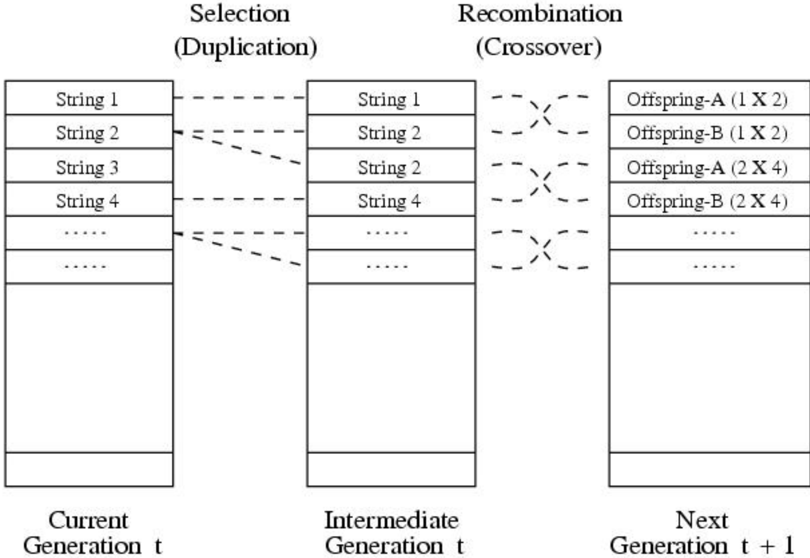


Fig. 4.1. One generation is broken down into a selection phase and recombination phase

There are many ways to do selection. We might view the population as mapping onto a roulette wheel, where each individual is represented by a space that proportionally corresponds to its fitness. By repeatedly spinning the roulette wheel, individuals are chosen using "stochastic sampling with replacement" to fill the intermediate population.

A selection process that will more closely match the expected fitness values is "remainder stochastic sampling". For each string i where f_i / f is greater than 1.0, the integer portion of this number indicates how many copies of that string are directly placed in the intermediate population. All strings (including those with f_i / f less than 1.0) then place additional copies in the intermediate population with a probability corresponding to the fractional portion of f_i / f. For example, a string with $f_i / f = 1.36$ places 1 copy in the intermediate population, and then receives a 0.36 chance of placing

a second copy. A string with a fitness of $f_i / f = 0.54$ has a 0.54 chance of placing one string in the intermediate population.

"Remainder stochastic sampling" is most efficiently implemented using a method known as Stochastic Universal Sampling. Assume that the population is laid out in random order as in a pie graph, where each individual is assigned space on the pie graph in proportion to fitness. Next an outer roulette wheel is placed around the pie with N equally spaced pointers. A single spin of the roulette wheel will now simultaneously pick all N members of the intermediate population. The resulting selection is also unbiased (Baker, 1987).

After selection has been carried out, the construction of the intermediate population is complete and recombination can occur. This can be viewed as creating the next population from the intermediate population. Crossover is applied to randomly paired strings with a probability denoted p_c. (The population should already be sufficiently shuffled by the random selection process). Pick a pair of strings. With probability p_c "recombine" these strings to form two new strings that are inserted into the next population.

Consider the following binary string: 1101001100101101. The string would represent a possible solution to some parameter optimization problem. New sample points in the space are generated by recombining two parent strings. Consider the string 1101001100101101 and another binary string, yxyyxyxxyyyxyxxy, in which the values 0 and 1 are denoted by x and y. Using a single randomly chosen recombination point, 1 point crossover occurs as follows:

11010 \/ 01100101101
yxyyx /\ yxxyyyxyxxy

Swapping the fragments between the two parents produces the following offspring.

11010yxxyyyxyxxy and yxyyx01100101101

After recombination, we can apply a mutation operator. For each bit in the population, mutate with some low probability p_m. Typically the mutation rate is applied with less than 1% probability. In some cases, mutation is interpreted as randomly generating a new bit, in which case, only 50% of the time will the “mutation” actually change the bit value. In other cases, mutation is interpreted to mean actually flipping the bit. The difference is no more than an implementation detail as long as the user/reader is aware of the difference and understands that the first form of mutation produces a change in bit values only half as often as the second, and that one version of mutation is just a scaled version of the other.

After the process of selection, recombination and mutation is complete, the next population can be evaluated. The process of evaluation, selection, recombination and mutation forms one generation in the execution of a genetic algorithm.

4.1.4. Crossover Operation Details

The general idea of the crossover operation is to exchange of “genetic material” between two individuals from population. The exchange between parents generates children – the new individuals. Finally the parents are substituted by children in population.

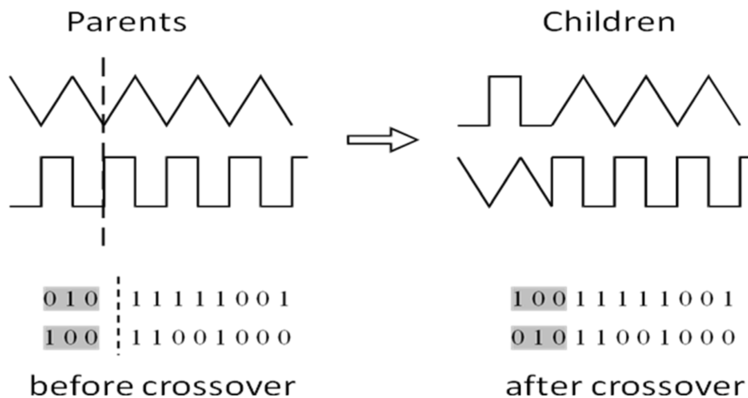


Fig. 4.2. Crossover operation

Steps of the crossover operation:

- select couples of chromosomes from parents set randomly according to a crossover probability $p_c \in (0.5, 1)$,
- choose a point of crossing l_k (gene's position in chromosome) as a number from $[1, L\text{-}1]$, L – length of chromosome,
- descendants creation by parents genes exchange.

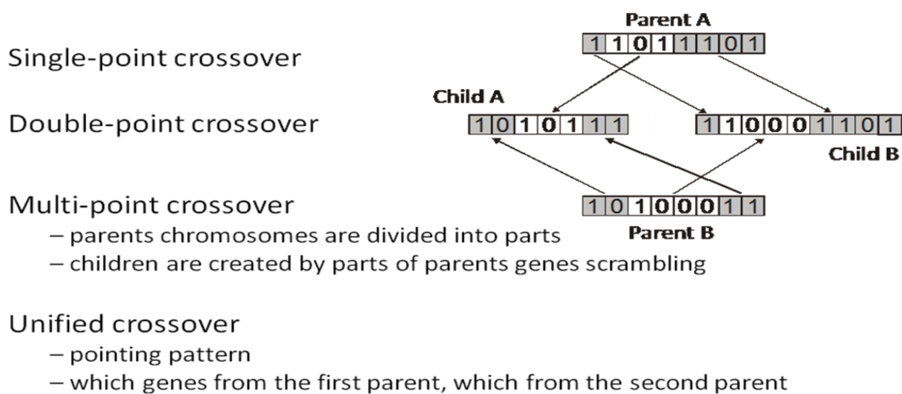


Fig. 4.3. Types of crossover operation

4.2. List of Problems

1. Genetic Algorithms for minima search of multimodal functions.
2. Genetic Algorithms for maxima search of multimodal functions.

4.3. Phases of Laboratory Exercises

1. Check and improve the necessary softcomputing knowledge.
2. Prepare and collect the necessary data sets: for training and for testing.

3. Realise the necessary data preprocessing and/or data postprocessing using different types of ready-to-use software or by “hand-made” software prepared by the laboratory group.

4. Prepare your own software to implement the proper softcomputing algorithms. The main goal is to create the correctly working engine, user interface utilities are not so important. The software environments and systems you can use for implementation are limited, but the actual possibilities ought to be discussed with the laboratory supervisor.

5. Turn on and tune the prepared software engine, supply the input training and/or testing data. If the engine works correctly, check what happens when the starting point parameters change, explore the sensitivity of engine for the different sets of available parameters and find the solution of the problem. At the end, check if the used softcomputing solution is correctly fitted to the problem.

6. Prepare the final report including the following parts:

   – the short description of the problem with necessary assumptions,

   – definitions and descriptions of the training and testing sets of input data with description of the preprocessing procedures,

   – definitions and descriptions of the of output data with description of the postprocessing procedures,

   – description of the tuned topology and parameters of the prepared softcomputing engine,

   – detailed results analysis and final remarks.

4.4. Hints for the List of Problems

Problem No. 1

1. Choose the representative of the polynomial type of function.
2. Fix the domain and the value of the function.
3. Divide the fixed domain into intervals.
4. Using the known set of genetic algorithm operations find the possible minima in each interval.

5. Check the influence of the interval division for the achieved results.
6. Find the optimal interval division for chosen function.

Problem No. 2

1. Choose the representative of the polynomial type of function.
2. Fix the domain and the value of the function.
3. Divide the fixed domain into intervals.
4. Using the known set of genetic algorithm operations find the possible maxima in each interval.
5. Check the influence of the interval division for the achieved results.
6. Find the optimal interval division for chosen function.

Part 5. Expert Systems

5.1. Theoretical Background

5.1.1. Knowledge and its Representation

Knowledge is a theoretical or practical understanding of a subject or a domain. Knowledge is also the sum of what is currently known, and apparently knowledge is power. Those who possess knowledge are called experts. Anyone can be considered a **domain expert** if he or she has deep knowledge (of both facts and rules) and strong practical experience in a particular domain. The area of the domain may be limited. In general, an expert is a skilful person who can do things other people cannot. The human mental process is internal, and it is too complex to be represented as an algorithm. However, most experts are capable of expressing their knowledge in the form of **rules** for problem solving.

IF the 'traffic light' is green
THEN the action is go
IF the 'traffic light' is red
THEN the action is stop

The term ***rule*** in AI, which is the most commonly used type of knowledge representation, can be defined as an IF-THEN structure that relates given information or facts in the IF part to some action in the THEN part. A rule provides some description of how to solve a problem. Rules are relatively easy to create and understand. Any rule consists of two parts: the IF part, called the ***antecedent*** (***premise*** or ***condition***) and the THEN part called the ***consequent*** (***conclusion*** or ***action***).

IF <antecedent>
THEN <consequent>

A rule can have multiple antecedents joined by the keywords **AND** (**conjunction**), **OR** (**disjunction**) or a combination of both.

IF	<antecedent 1>	IF	<antecedent 1>
AND	<antecedent 2>	OR	<antecedent 2>
	.		.
	.		.
	.		.
AND	<antecedent *n*>	OR	<antecedent *n*>
THEN	<consequent>	THEN	<consequent>

The antecedent of a rule incorporates two parts: an ***object*** (*linguistic object*) and its ***value***. The object and its value are linked by an ***operator***. The operator identifies the object and assigns the value. Operators such as *is*, *are*, *is not*, *are not* are used to assign a **symbolic value** to a linguistic object. Expert systems can also use mathematical operators to define an object as numerical and assign it the **numerical value**.

IF ‘age of the customer’ < 18
AND ‘cash withdrawal’ > 1000
THEN ‘signature of the parent’ is required

Rules can represent relations, recommendations, directives, strategies and heuristics:

Relation

IF the ‘fuel tank’ is empty
THEN the car is dead

Recommendation

IF the season is autumn
AND the sky is cloudy
AND the forecast is drizzle
THEN the advice is ‘take an umbrella’

Directive

IF	the car is dead
AND	the ‘fuel tank’ is empty
THEN	the action is ‘refuel the car’

Strategy

IF	the car is dead
THEN	the action is ‘check the fuel tank’;
	step1 is complete
IF	step1 is complete
AND	the ‘fuel tank’ is full
THEN	the action is ‘check the battery’;
	step2 is complete

Heuristic

IF	the spill is liquid
AND	the ‘spill pH’ < 6
AND	the ‘spill smell’ is vinegar
THEN	the ‘spill material’ is ‘acetic acid’

5.1.2. Members of Expert System Development Team

There are five members of the expert system development team: the **domain expert**, the **knowledge engineer**, the **programmer**, the **project manager** and the **end-user**. The success of their expert system entirely depends on how well the members work together.

The ***domain expert*** is a knowledgeable and skilled person capable of solving problems in a specific area or ***domain***. This person has the greatest expertise in a given domain. This expertise is to be captured in the expert system. Therefore, the expert must be able to communicate his or her knowledge, be willing to participate in the expert

system development and commit a substantial amount of time to the project. The domain expert is the most important player in the expert system development team.

The ***knowledge engineer*** is someone who is capable of designing, building and testing an expert system. He or she interviews the domain expert to find out how a particular problem is solved. The knowledge engineer establishes what reasoning methods the expert uses to handle facts and rules and decides how to represent them in the expert system. The knowledge engineer then chooses some development software or an expert system shell, or looks at programming languages for encoding the knowledge. And finally, the knowledge engineer is responsible for testing, revising and integrating the expert system into the workplace.

The ***programmer*** is the person responsible for the actual programming, describing the domain knowledge in terms that a computer can understand. The programmer needs to have skills in symbolic programming in such AI languages as LISP, Prolog and OPS5 and also some have experience in the application of different types of expert system shells. Additionally, the programmer should know conventional programming languages like C, Pascal, FORTRAN and Basic.

The ***project manager*** is the leader of the expert system development team, responsible for keeping the project on track. He or she makes sure that all deliverables and milestones are met, interacts with the expert, knowledge engineer, programmer and end-user. The ***end-user***, often called just the *user*, is a person who uses the expert system when it is developed. The user must not only be confident in the expert system performance but also feel comfortable using it. Therefore, the design of the user interface of the expert system is also vital for the project's success; the end-user's contribution here can be crucial.

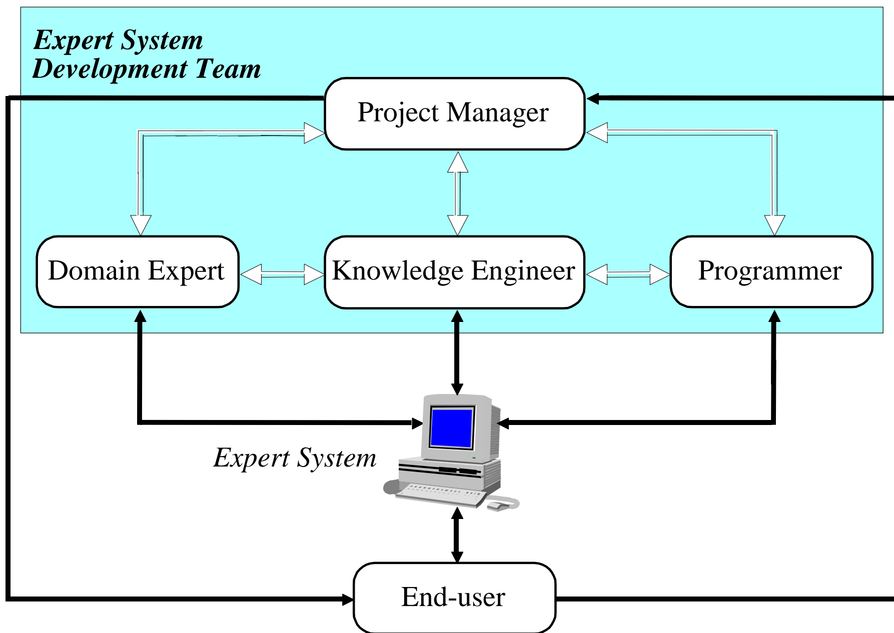


Fig. 5.1. Members of Expert System Development Team

5.1.3. Structure of the Rule-Based Expert System

In the early seventies, a production systems model was proposed as the foundation of the modern rule-based expert systems. The production model is based on the idea that humans solve problems by applying their knowledge (expressed as production rules) to a given problem represented by problem-specific information. The production rules are stored in the long-term memory and the problem-specific information or facts in the short-term memory.

5.1.4. Expert System Characteristic

The **knowledge base** contains the domain knowledge useful for problem solving. In a rule-based expert system, the knowledge is represented as a set of rules.

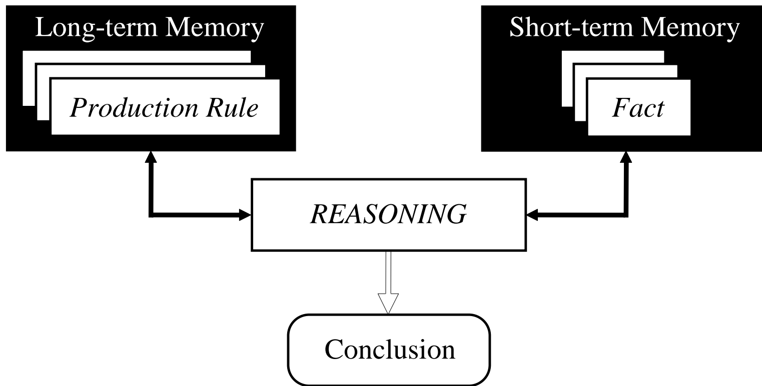


Fig. 5.2. Production System Model

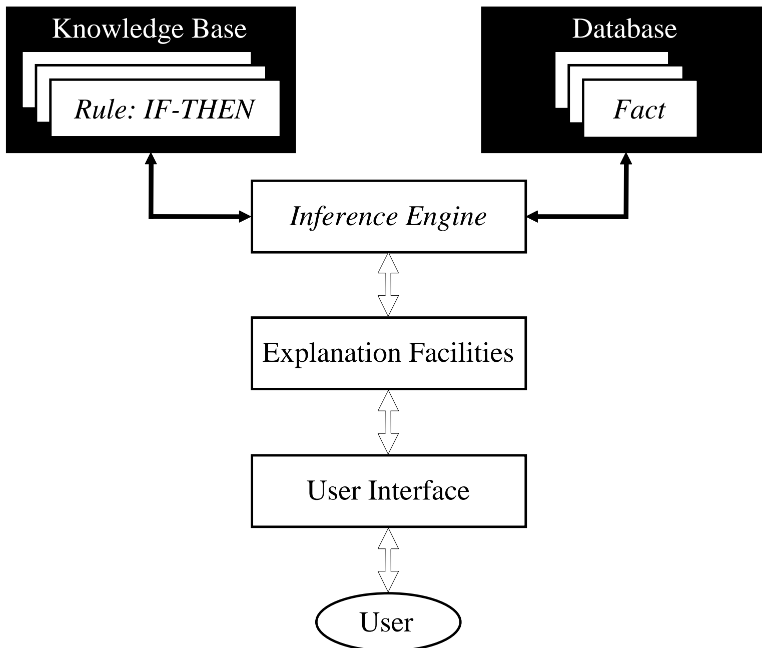


Fig. 5.3. Basic Structure of the Rule-Based Expert System

Each rule specifies a relation, recommendation, directive, strategy or heuristic and has the IF (condition) THEN (action) structure. When the condition part of a rule is satisfied, the rule is said to ***fire*** and the action part is executed. The **database** includes a set of facts used to match against the IF (condition) parts of rules stored in the knowledge base. The **inference engine** carries out the reasoning, whereby the expert system reaches a solution. It links the rules given in the knowledge base with the facts provided in the database. The **explanation facilities** enable the user to ask the expert system ***how*** a particular conclusion is reached and ***why*** a specific fact is needed. An expert system must be able to explain its reasoning and justify its advice, analysis and conclusion. The **user interface** is the means of communication between a user seeking a solution to the problem and an expert system. An expert system is built to perform at a human expert level in a ***narrow, specialised domain***. Thus, the most important characteristics of an expert system is its high-quality performance. No matter how fast the system can solve a problem, the user will not be satisfied if the result is wrong. On the other hand, the speed of reaching a solution is very important. Even the most accurate decision or diagnosis may not be useful if it is too late to apply, for instance, in an emergency, when a patient dies or a nuclear power plant explodes. Expert systems apply **heuristics** to guide the reasoning and thus reduce the search area for a solution. A unique feature of an expert system is its **explanation capability**. It enables the expert system to review its own reasoning and explain its decisions. Expert systems employ **symbolic reasoning** when solving a problem. Symbols are used to represent different types of knowledge such as facts, concepts and rules. Even a brilliant expert is only a human and thus can make mistakes. This suggests that an expert system built to perform at a human expert level also should be allowed to make mistakes. But we still trust experts, even if we recognise that their judgements are sometimes wrong.

Likewise, at least in most cases, we can rely on solutions provided by expert systems, but mistakes are possible and we should be aware of this. In expert systems, **knowledge is separated from its processing** (the knowledge base and the inference

engine are split up). A conventional program is a mixture of knowledge and the control structure to process this knowledge. This mixing leads to difficulties in understanding and reviewing the program code, as any change to the code affects both the knowledge and its processing. When an expert system shell is used, a knowledge engineer or an expert simply enters rules in the knowledge base. Each new rule adds some new knowledge and makes the expert system smarter.

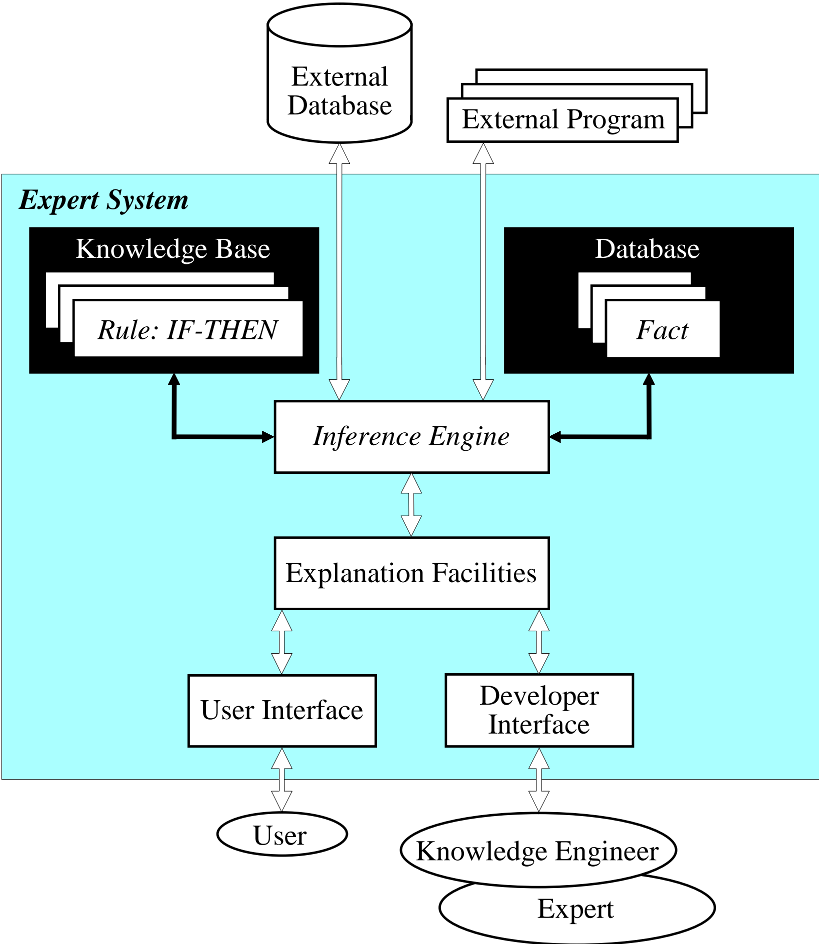


Fig. 5.4. Complete Structure of the Rule-Based Expert System

5.1.5. Forward Chaining and Backward Chaining

In a rule-based expert system, the domain knowledge is represented by a set of IF-THEN production rules and the data is represented by a set of facts about the current situation. The inference engine compares each rule stored in the knowledge base with facts contained in the database. When the IF (condition) part of the rule matches a fact, the rule is **fired** and its THEN (action) part is executed. The matching of the rule IF parts to the facts produces **inference chains**. The inference chain indicates how an expert system applies the rules to reach a conclusion.

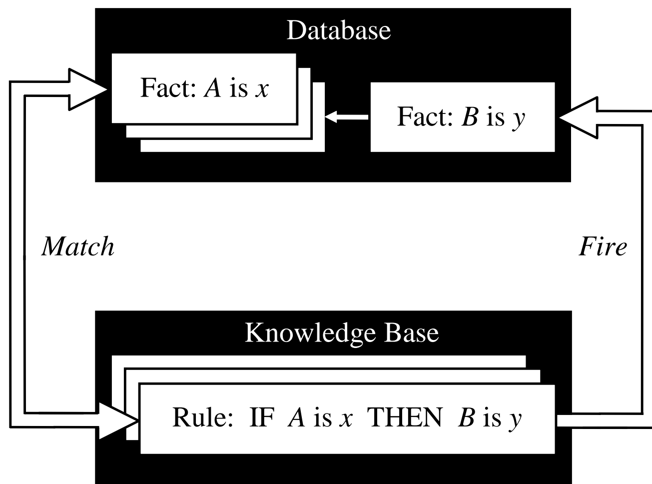
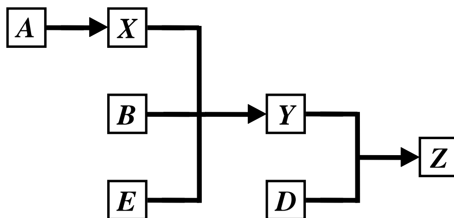


Fig. 5.5. Inference Engine Cycles via Match-Fire Procedure

An example of the inference chain

Rule 1: IF *Y* is true
AND *D* is true
THEN *Z* is true

Rule 2: IF *X* is true



AND *B* is true

AND *E* is true

THEN *Y* is true

Rule 3: IF *A* is true

THEN *X* is true

Forward chaining is the **data-driven reasoning**. The reasoning starts from the known data and proceeds forward with that data. Each time only the topmost rule is executed. When fired, the rule adds a new fact in the database. Any rule can be executed only once. The match-fire cycle stops when no further rules can be fired. Forward chaining is a technique for gathering information and then inferring from it whatever can be inferred. However, in forward chaining, many rules may be executed that have nothing to do with the established goal. Therefore, if our goal is to infer only one particular fact, the forward chaining inference technique would not be efficient.

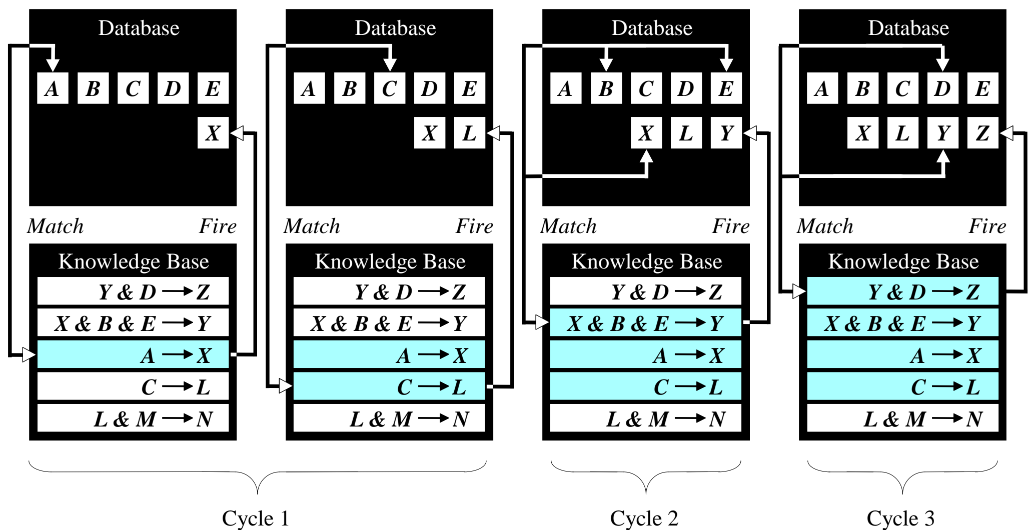


Fig. 5.6. Forward Chaining

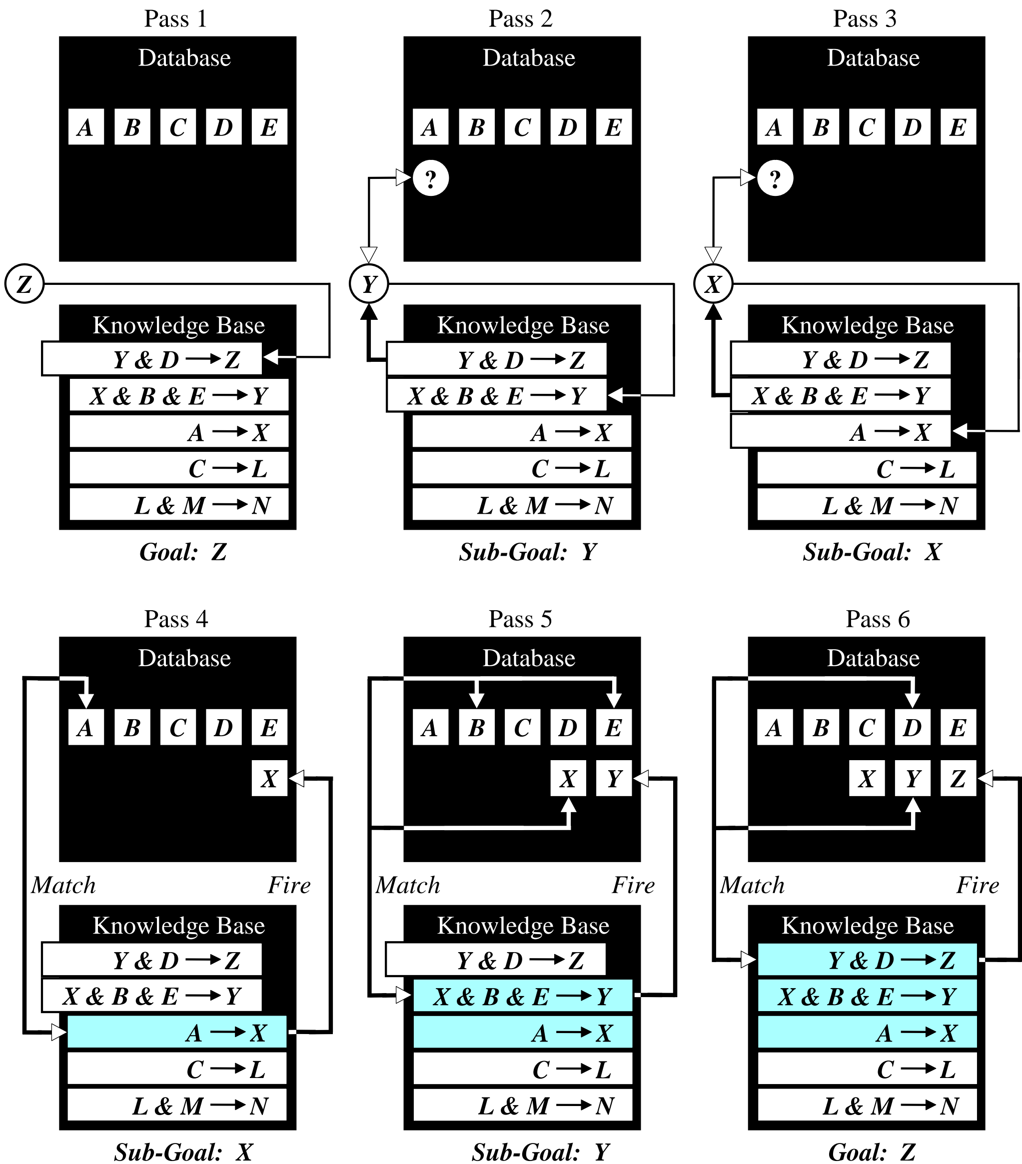


Fig. 5.7. Backward Chaining

Backward chaining is the **goal-driven reasoning**. In backward chaining, an expert system has the goal (a *hypothetical solution*) and the inference engine attempts to find the evidence to prove it. First, the knowledge base is searched to find rules that

might have the desired solution. Such rules must have the goal in their THEN (action) parts. If such a rule is found and its IF (condition) part matches data in the database, then the rule is fired and the goal is proved. However, this is rarely the case. Thus the inference engine puts aside the rule it is working with (the rule is said to ***stack***) and sets up a new goal, a subgoal, to prove the IF part of this rule. Then the knowledge base is searched again for rules that can prove the subgoal. The inference engine repeats the process of stacking the rules until no rules are found in the knowledge base to prove the current subgoal.

If an expert first needs to gather some information and then tries to infer from it whatever can be inferred, he or she chooses the forward chaining inference engine. However, if your expert begins with a hypothetical solution and then attempts to find facts to prove it, he or she chooses the backward chaining inference engine.

5.1.6. Conflict Resolution

Lets consider three simple rules for crossing a road:

Rule 1:

IF the 'traffic light' is green
THEN the action is go

Rule 2:

IF the 'traffic light' is red
THEN the action is stop

Rule 3:

IF the 'traffic light' is red
THEN the action is go

We have two rules, *Rule* 2 and *Rule* 3, with the same IF part. Thus both of them can be set to fire when the condition part is satisfied. These rules represent a conflict

set. The inference engine must determine which rule to fire from such a set. A method for choosing a rule to fire when more than one rule can be fired in a given cycle is called **conflict resolution**. in forward chaining, *BOTH* rules would be fired. *Rule* 2 is fired first as the topmost one, and as a result, its THEN part is executed and linguistic object *action* obtains value *stop*. However, *Rule* 3 is also fired because the condition part of this rule matches the fact *'traffic light' is red*, which is still in the database. As a consequence, the object *action* takes new value *go*.

Fire the rule with the ***highest priority***. In simple applications, the priority can be established by placing the rules in an appropriate order in the knowledge base. Usually this strategy works well for expert systems with around 100 rules.

Fire the ***most specific rule***. This method is also known as the ***longest matching strategy***. It is based on the assumption that a specific rule processes more information than a general one.

Fire the rule that uses the ***data most recently entered*** in the database. This method relies on time tags attached to each fact in the database. In the conflict set, the expert system first fires the rule whose antecedent uses the data most recently added to the database.

5.1.7. Metaknowledge

Metaknowledge can be simply defined as ***knowledge about knowledge***. Metaknowledge is knowledge about the use and control of domain knowledge in an expert system. In rule-based expert systems, metaknowledge is represented by **metarules**. A metarule determines a strategy for the use of task-specific rules in the expert system.

***Metarule* 1:**

Rules supplied by experts have higher priorities than rules supplied by novices.

***Metarule* 2:**

Rules governing the rescue of human lives have higher priorities than rules concerned with clearing overloads on power system equipment.

5.1.8. Advantages of Expert System

- **natural knowledge representation**,
    - an expert usually explains the problem-solving procedure with such expressions as this: “In such-and-such situation, I do so-and-so”,
    - these expressions can be represented quite naturally as IF-THEN production rules.
- **uniform structure**,
    - production rules have the uniform IF-THEN structure. Each rule is an independent piece of knowledge,
    - the very syntax of production rules enables them to be self-documented,
- **separation of knowledge from its processing**,
    - the structure of a rule-based expert system provides an effective separation of the knowledge base from the inference engine.
    - this makes it possible to develop different applications using the same expert system shell.
- ***dealing with incomplete and uncertain knowledge***,
    - most rule-based expert systems are capable of representing and reasoning with incomplete and uncertain knowledge.

5.2. List of Problems

1. Expert system for car purchase assistance.
2. Expert system for heart attack risk estimation.
3. Expert system for diabetes risk estimation.

4. Expert system for a type and make of mobile phone selection based on user needs.

5. Expert system for selection of places for events in Wrocław.

5.3. Phases of Laboratory Exercises

1. Check and improve the necessary softcomputing knowledge.
2. Prepare and collect the necessary data sets: for training and for testing.
3. Realise the necessary data preprocessing and/or data postprocessing using different types of ready-to-use software or by “hand-made” software prepared by the laboratory group.
4. Prepare your own software to implement the proper softcomputing algorithms. The main goal is to create the correctly working engine, user interface utilities are not so important. The software environments and systems you can use for implementation are limited, but the actual possibilities ought to be discussed with the laboratory supervisor.
5. Turn on and tune the prepared software engine, supply the input training and/or testing data. If the engine works correctly, check what happens when the starting point parameters change, explore the sensitivity of engine for the different sets of available parameters and find the solution of the problem. At the end, check if the used softcomputing solution is correctly fitted to the problem.
6. Prepare the final report including the following parts:

   – the short description of the problem with necessary assumptions,

   – definitions and descriptions of the training and testing sets of input data with description of the preprocessing procedures,

   – definitions and descriptions of the of output data with description of the postprocessing procedures,

   – description of the tuned topology and parameters of the prepared softcomputing engine,

   – detailed results analysis and final remarks.

5.4. Hints for the List of Problems

All Problems

1. Choose and collect the data and information related to the problem.
2. Divide the collected data into categories, create the knowledge base and chaining mechanisms.
3. Prepare the set of questions for users.
4. Implement the knowledge base, the chaining, and the user interface.
5. Tune the system, check the available answers and results.

Part 6. Fuzzy Logic

1.1. Theoretical Background

6.1.1 Fuzzy Set Theory

The basic idea of the fuzzy set theory is that an element belongs to a fuzzy set with a certain degree of membership. A fuzzy set is a set with fuzzy boundaries. A proposition is neither true nor false (fuzzy logic), but may be partly true (or partly false) to any degree. This degree is usually taken as a real number in the range [0,1]. This way the fuzzy logic is an extension of classic two-valued logic – the truth value of a sentence is not restricted to true or false.

In the fuzzy set theory a classical example is a set is *tall men*. The elements of the fuzzy set "tall men" are all men, but their degrees of membership depend on their height.

Name	Height, cm	Degree of Membership	
		Crisp	*Fuzzy*
Chris	208	1	1.00
Mark	205	1	1.00
John	198	1	0.98
Tom	181	1	0.82
David	179	0	0.78
Mike	172	0	0.24
Bob	167	0	0.15
Steven	158	0	0.06
Bill	155	0	0.01
Peter	152	0	0.00

Fig. 6.1. *Tall men* fuzzy set

The x-axis represents the universe of discourse – the range of all possible values applicable to a chosen variable. In our case, the variable is the man's height. According

to this representation, the universe of men's heights consists of all tall men. The y-axis represents the membership value of the fuzzy set. In our case, the fuzzy set of *tall men* maps height values into corresponding membership values.

A fuzzy set is a set with fuzzy boundaries. Let X be the universe of discourse and its elements be denoted as x. In the classical set theory, crisp set A of X is defined as function $f_A(x)$ called the characteristic function of A

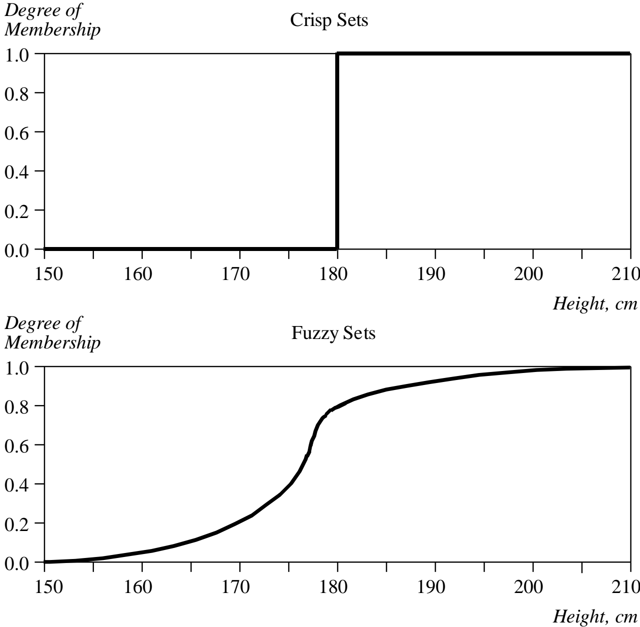


Fig. 6.2. Crisp and fuzzy set of *tall men*

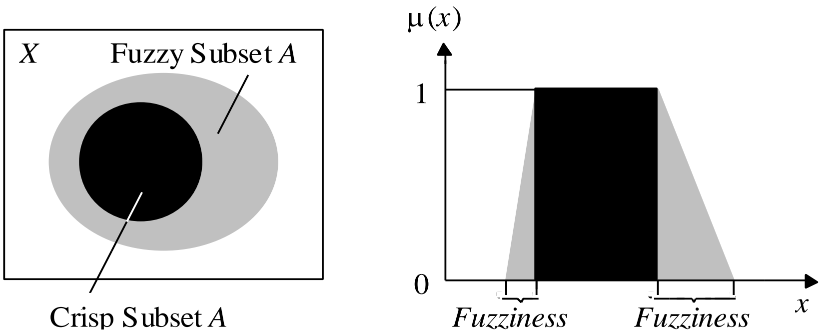
$$f_A(x): X \rightarrow \{0, 1\}, \text{ where } f_A(x) = 1 \text{ if } x \in A \text{ or } f_A(x) = 0 \text{ if } x \notin A$$

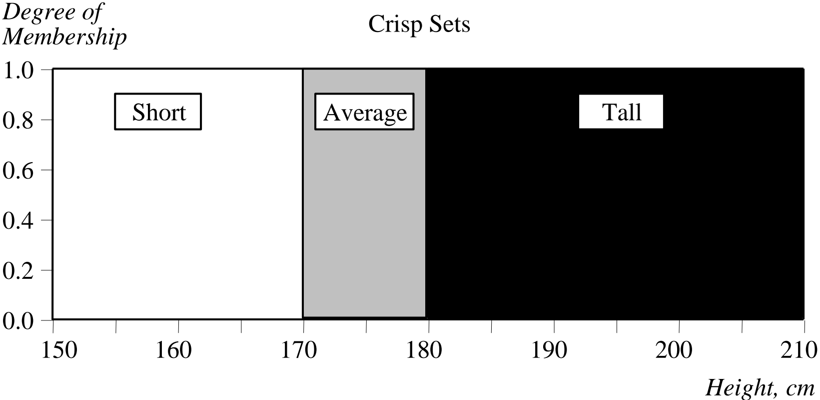
In the fuzzy theory, fuzzy set A of universe X is defined by function $\mu_A(x)$ called the *membership function* of set A

$$\mu_A(x): X \rightarrow [0, 1] \text{ where } \mu_A(x) = 1 \text{ if } x \text{ is totally in } A \text{ or } \mu_A(x) = 0 \text{ if } x \text{ is not in } A$$

$$\text{or } 0 < \mu_A(x) < 1 \text{ if } x \text{ is partly in } A$$

For any element x of universe X, membership function $\mu_A(x)$ equals to the degree to which x is an element of set A. This degree, a value between 0 and 1, represents the degree of membership of element x in set A.





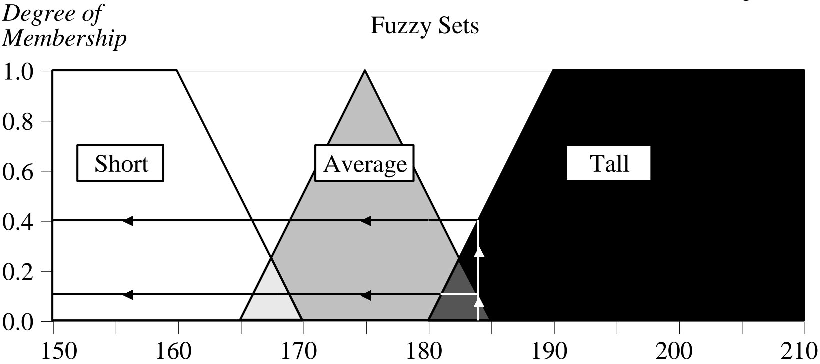


Fig. 6.3. More crisp and fuzzy sets defined on the universe

Typical functions that can be used to represent a fuzzy set are sigmoid, Gaussian and pi. However, these functions are computation-intensive. Therefore, in practice, most applications represent fuzzy subsets by linear fit functions.

A linguistic variable is a fuzzy variable. For example, the statement "John is tall" implies that the linguistic variable *John* takes the linguistic value *tall.* The range of possible values of a linguistic variable represents the universe of discourse of that variable.

For example, the universe of discourse of the linguistic variable *speed* might have the range between 0 and 220 km/h and may include such fuzzy subsets as *very slow*, *slow*, *medium*, *fast*, and *very fast*.

A linguistic variable carries with it the concept of fuzzy set qualifiers, called hedges. Hedges are terms that modify the shape of fuzzy sets. They include *adverbs* such as *very*, *somewhat*, *quite*, *more or less* and *slightly*.

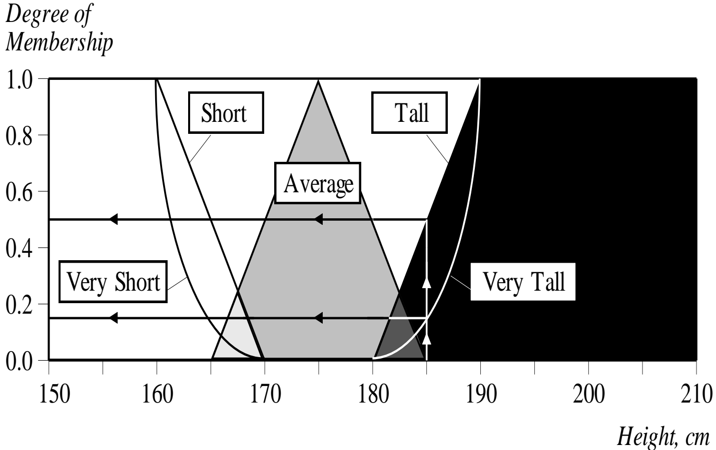


Fig. 6.4. Fuzzy sets with the hedge *very*

1.1.2. Fuzzy Set Operations

- **Complement**

The complement of a set is an opposite of this set. For example, if we have the set of *tall men*, its complement is the set of *NOT tall men*. When we remove the tall men set from the universe of discourse, we obtain the complement. If A is the fuzzy set, its complement $-A$ can be found as follows: $\mu_{-A}(x) = 1 - \mu_A(x)$.

- **Containment**

The set of *tall men* contains all tall men; *very tall men* is a subset of *tall men*. However, the *tall men* set is just a subset of the set of *men*. In crisp sets, all elements of a subset entirely belong to larger set. In fuzzy sets, however, each element can belong less to the subset than to the larger set. Elements of the fuzzy subset have smaller memberships in it than in the larger set.

- **Intersection**

In classical set theory, an intersection between two sets contains the elements shared by these sets. For example, the intersection of the set of *tall men* and the set of *fat men* is the area where these sets overlap. In fuzzy sets, an element may partly belong to both sets with different memberships. A fuzzy intersection is the lower membership in both sets of each element. The fuzzy intersection of two fuzzy sets A and B on universe of discourse X:

$$\mu_{A \cap B}(x) = min\ [\mu_A(x), \mu_B(x)] = \mu_A(x) \cap \mu_B(x), \text{ where } x \in X$$

- **Union**

The union of two crisp sets consists of every element that falls into either set. For example, the union of *tall men* and *fat men* contains all men who are tall OR fat. In fuzzy sets, the union is the reverse of the intersection. That is, the union is the largest

membership value of the element in either set. The fuzzy operation for forming the union of two fuzzy sets *A* and *B* on universe *X* can be given as: $\mu_{A \cup B}(x) = max\ [\mu_A(x), \mu_B(x)] = \mu_A(x) \cup \mu_B(x)$, *where* $x \in X$

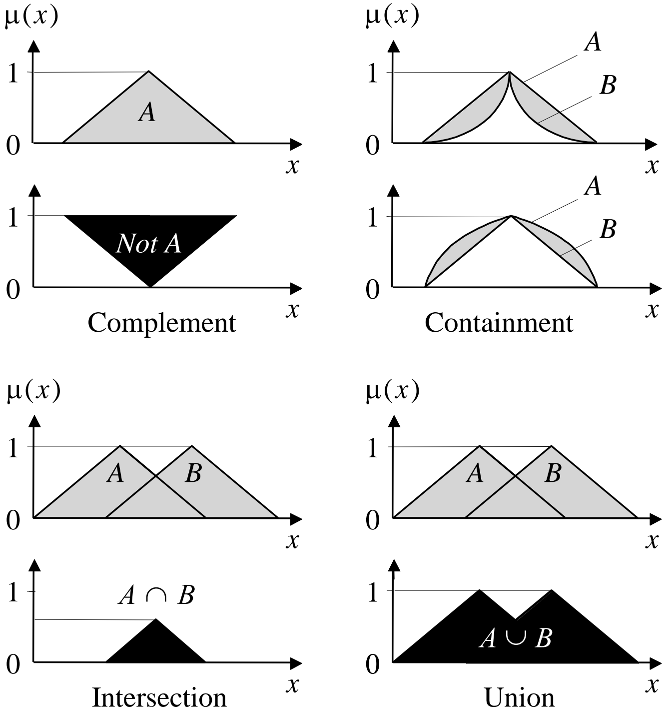


Fig. 6.5. Operations of fuzzy sets

1.1.3. Fuzzy Logic

Fuzzy logic is an extension of classic two-valued logic – the truth value of a sentence is not restricted to true or false. Example of fuzzy sentences: Φ = Height(John, tall) $|\Phi| = 0.9$, Ψ = Speed(Mazda, fast) $|\Psi| = 0.3$. The truth value of each logic connective can be defined:

$|-\Phi| = 1 - |\Phi|$

$|\Phi \wedge \Psi| = \min \{|\Phi|, |\Psi|\}$ /* t-norm */

$|\Phi \vee \Psi| = \max\{|\Phi|, |\Psi|\}$ /* t-conorm */

$|\Phi \rightarrow \Psi| = \min\{1 - |\Phi| + |\Psi|, 1\}$ /* Lukasiewicz */

There are four types of fuzzy logics:

- Classic logic
  - Crisp sentence: Height(John, 180) → Weight(John, 60)
  - Crisp data: Height(John, 180)
- Truth-functional multi-valued logic (Fuzzy knowledge base)
  - Fuzzy sentence: Height(John, tall) → Weight(John, heavy)
  - Crisp data: Height(John, 180)
- Possibilistic logic (Uncertain knowledge base)
  - Crisp sentence: Height(John, 180) → Weight(John, 60)
  - Fuzzy data: Height(John, tall)
- Uncertain fuzzy knowledge base
  - Fuzzy sentence: Height(John, tall) → Weight(John, heavy)
  - Fuzzy data: Height(John, tall)

Fuzzy Inference

- Classic logic
  - Resolution – sound and complete: $\{l_1 \vee l_2, L_1 \vee \neg l_2\} \vdash_{res} \{l_1 \vee L_1\}$
- Possibilistic logic
  - Possibilitic resolution principle – sound and complete

$$\{(l_1 \vee l_2, a_1), (L_1 \vee \neg l_2, a_2)\} \vdash_{pres}$$

$$\{l_1 \vee L_1, \min\{a_1, a_2\}\}$$

Truth-functional multi-valued logic (Fuzzy knowledge base)

- Generalized Modus Ponens – sound and complete
- According to Lukasiewicz:

  $|\Phi \rightarrow \Psi| = \min \{1 - |\Phi| + |\Psi|, 1\}$

  $\Rightarrow |\Phi \rightarrow \Psi| \leq 1 - |\Phi| + |\Psi|$

  => if set $|\Phi \rightarrow \Psi| = 1 - |\Phi| + |\Psi|$ => then $|\Psi| = |\Phi| + (|\Phi \rightarrow \Psi| - 1)$

- $|\Psi| = |\Phi|$ if $|\Phi \rightarrow \Psi| = 1$, i.e., the truth value of $\Phi \rightarrow \Psi$ is always true, which serves as the basis for using "clipping" to get output in Fuzzy Control or Knowledge-based Systems

1.1.4. Fuzzy Knowledge – Based Systems

Fuzzy Knowledge-based Systems or Fuzzy Control Systems are a special type of truth-functional multi-valued logic. Fuzzy rules are used to relate fuzzy sets, The truth value of each fuzzy rule is 1. Fuzzy rule representation:

IF *x* is *A* THEN *y* is *B*

where *x* and *y* are linguistic variables; and *A* and *B* are linguistic values determined by fuzzy sets on the universe of domains *X* and *Y*, respectively.

A fuzzy rule can have multiple antecedents:

IF project_duration is long
AND project_staffing is large
AND project_funding is inadequate
THEN risk is high

The consequent of a fuzzy rule can also include multiple parts:

IF	**temperature is hot**
THEN	**hot_water is reduced;**
	cold_water is increased

Difference between crisp and fuzzy rules:

– crisp rule:

IF **speed is >100**

THEN stopping_distance is long

– fuzzy rule:

IF **speed is fast**

THEN stopping_distance is long

In a fuzzy system, all rules fire to some extent, or in other words they fire partially. If the antecedent is true to some degree of membership, then the consequent is also true to that same degree.

Operations for Mamdani fuzzy inference model we can discuss in four steps. The first one is a fuzzification of the input variables. The rule evaluation is the second. Next we have to aggregate the rule outputs and finally do the defuzzification is necessary.

We examine a simple two-input one-output problem that includes three rules:

Rule: 1		**Rule: 1**	
IF	***x* is *A*3**	**IF**	***project_funding* is *adequate***
OR	***y* is *B*1**	**OR**	***project_staffing* is *small***

THEN z **is** $C1$ **THEN** ***risk*** **is** ***low***

Rule: 2   **Rule: 2**

IF x **is** $A2$	**IF** ***project_funding*** **is** ***marginal***
AND y **is** $B2$	**AND** ***project_staffing*** **is** ***large***
THEN z **is** $C2$	**THEN** ***risk*** **is** ***normal***

Rule: 3   **Rule: 3**

IF x **is** $A1$	**IF** ***project_funding*** **is** ***inadequate***
THEN z **is** $C3$	**THEN** ***risk*** **is** ***high***

Step 1: Fuzzification

Take the crisp inputs, $x1$ and $y1$ (*project funding* and *project staffing*), and determine the degree to which these inputs belong to each of the appropriate fuzzy sets.

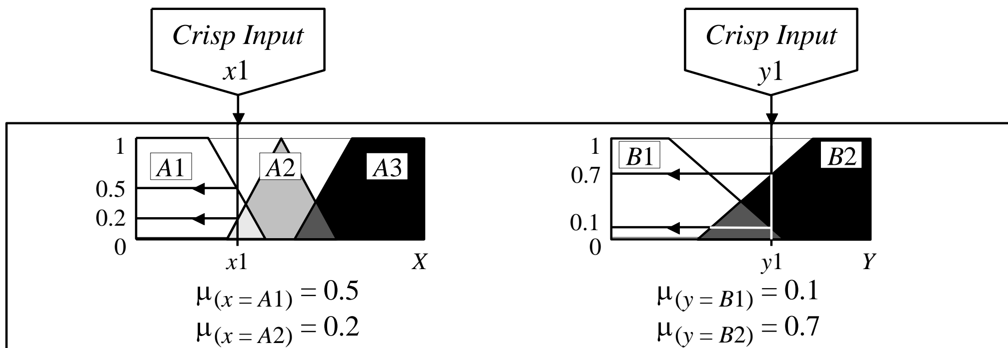


Fig. 6.6. Fuzzification

Step 2: Rule Evaluation

Take the fuzzified inputs, $\mu_{(x=A1)} = 0.5$, $\mu_{(x=A2)} = 0.2$, $\mu_{(y=B1)} = 0.1$ and $\mu_{(y=B2)} = 0.7$, and apply them to the antecedents of the fuzzy rules. If a given fuzzy rule has multiple

antecedents, the fuzzy operator (AND or OR) is used to obtain a single number that represents the result of the antecedent evaluation. This number (the truth value) is then applied to the consequent membership function. To evaluate the disjunction of the rule antecedents, we use the OR fuzzy operation. Typically, fuzzy expert systems make use of the classical fuzzy operation union: $\mu_{A \cup B}(x) = max\, [\mu_A(x), \mu_B(x)]$.

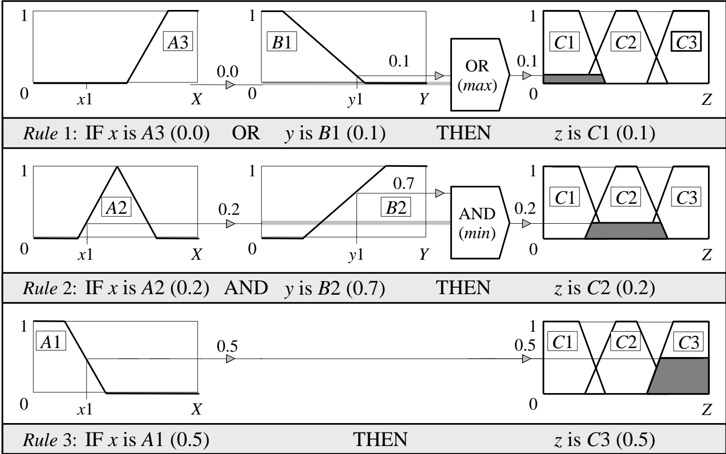


Fig. 6.7. Mamdani–style rule evaluation

Similarly, in order to evaluate the conjunction of the rule antecedents, we apply the AND fuzzy operation intersection: $\mu_{A \cap B}(x) = min\, [\mu_A(x), \mu_B(x)]$.

Clipping: The most common method of correlating the rule consequent with the truth value of the rule's antecedent is to cut the consequent membership function at the level of the antecedent truth. Since the top of the membership function is sliced, the clipped fuzzy set loses some information. Clipping is still often preferred because it involves less complex and faster mathematics, and generates an aggregated output surface that is easier to defuzzify.

Scaling: The original membership function of the rule's consequent is adjusted by multiplying all its membership degrees by the truth value of the rule antecedent.

Scaling offers a better approach for preserving the original shape of the fuzzy set; it generally loses less information and can be very useful in fuzzy expert systems.

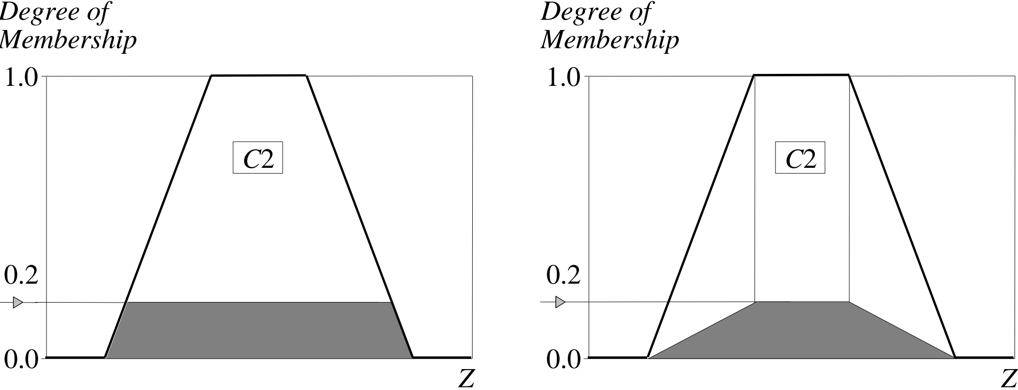


Fig. 6.8. Clipped and scaled membership functions

Step 3: Aggregation of the rule outputs

Aggregation is the process of unification of the outputs of all rules. We take the membership functions of all rule consequents previously clipped or scaled and combine them into a single fuzzy set. The input of the aggregation process is the list of clipped or scaled consequent membership functions, and the output is one fuzzy set for each output variable.

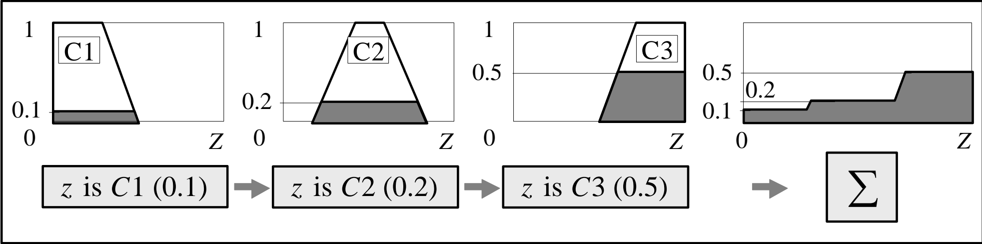


Fig. 6.9. Aggregation of the rule outputs

Step 4: Defuzzification

The input for the defuzzification process is the aggregate output fuzzy set and the output is a single number. Centroid technique based defuzzification methods. It finds the point where a vertical line would slice the aggregate set into two equal masses. Mathematically this centre of gravity (*COG*) can be expressed as:

$$COG = \frac{\int_a^b \mu_A(x)x\,dx}{\int_a^b \mu_A(x)\,dx} \tag{6.1}$$

COG can be obtained by calculating it over a sample of points.

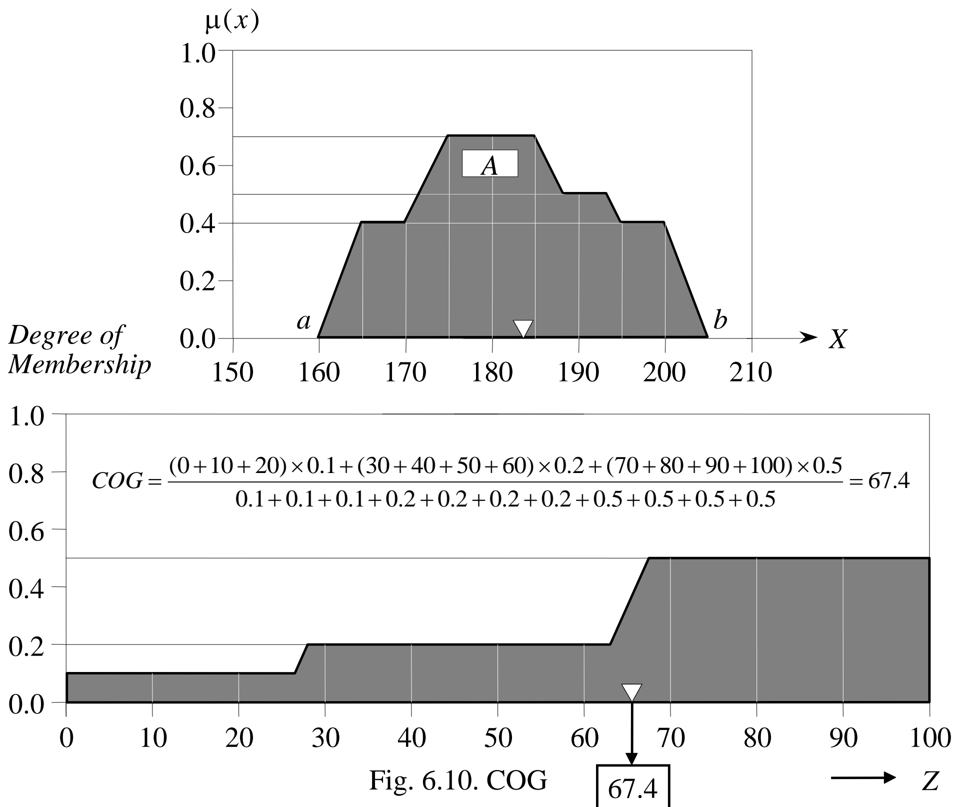


Fig. 6.10. COG

Finding the centroid of a two-dimensional shape by integrating across a continuously varying function is not computationally efficient. For Sugeno fuzzy inference model we use a single spike, a *singleton*, as the membership function of the rule consequent. A singleton, or more precisely a fuzzy singleton, is a fuzzy set with a membership function that is one at a single particular point on the universe of domain and zero everywhere else. Rule representation of Sugeno fuzzy inference model is very similar to the Mamdani method. Sugeno changed only a rule consequent. Instead of a fuzzy set, he used a mathematical function of the input variable.

IF $\quad$ ***x* is *A***

AND $\quad$ ***y* is *B***

THEN ***z* is *f*(*x, y*)**

where *x*, *y* and *z* are linguistic variables; *A* and *B* are fuzzy sets on universe of domains *X* and *Y*, respectively and *f* (*x, y*) is a mathematical function.

The most commonly used zero-order Sugeno fuzzy model applies fuzzy rules in the following form:

IF $\quad$ ***x* is *A***

AND $\quad$ ***y* is *B***

THEN ***z* is *k***

where *k* is a constant. In this case, the output of each fuzzy rule is constant. All consequent membership functions are represented by singleton spikes.

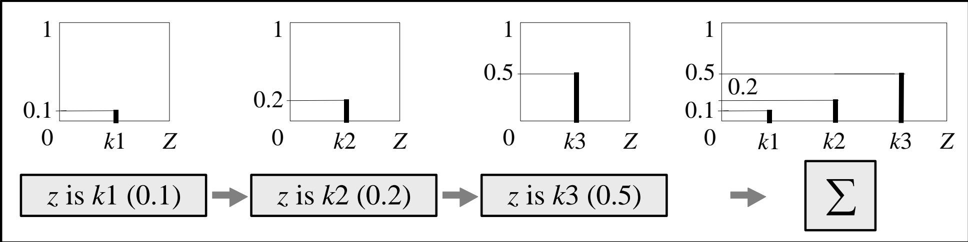


Fig. 6.11. Sugeno-style aggregation of the rule outputs

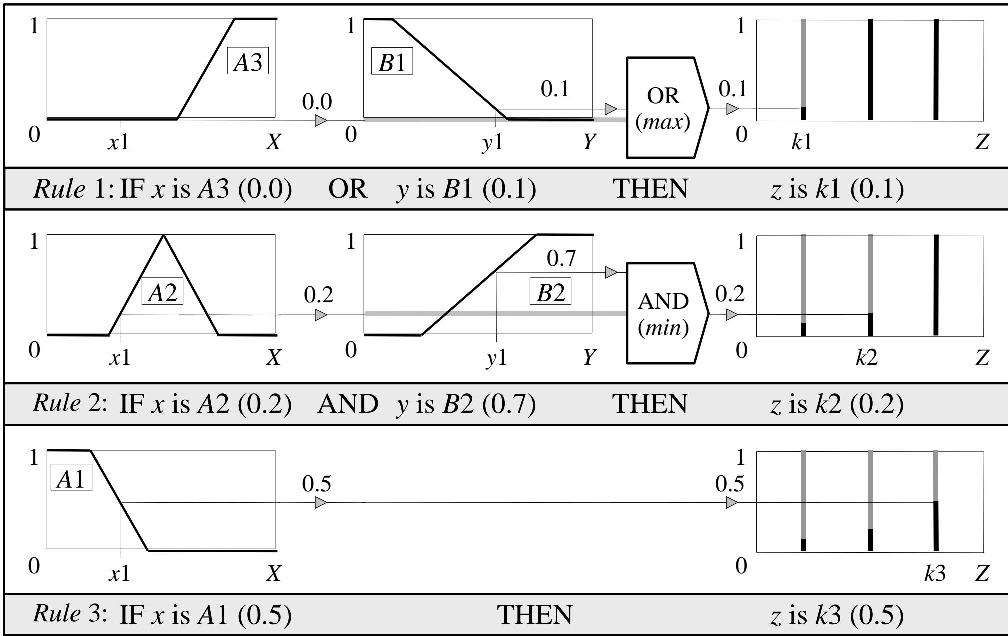
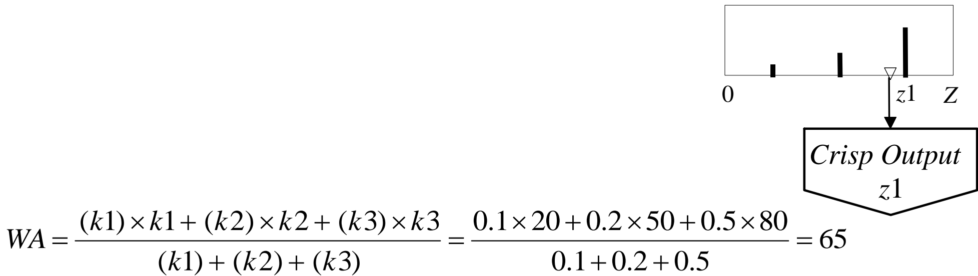


Fig. 6.12. Sugeno-style rule evaluation



$$WA = \frac{\mu(k1)\times k1 + \mu(k2)\times k2 + \mu(k3)\times k3}{\mu(k1)+\mu(k2)+\mu(k3)} = \frac{0.1\times 20 + 0.2\times 50 + 0.5\times 80}{0.1+0.2+0.5} = 65$$

Fig. 6.13. Sugeno-style defuzzification: Weighted average (WA)

Mamdani method is widely accepted for capturing expert knowledge. It allows us to describe the expertise in more intuitive, more human-like manner. However, Mamdani-type fuzzy inference entails a substantial computational burden. On the other hand, Sugeno method is computationally effective and works well with optimization

and adaptive techniques, which makes it very attractive in control problems, particularly for dynamic nonlinear systems.

1.1.5. Development of Fuzzy Expert Systems

- Specify the problem and define linguistic variables.
- Determine fuzzy sets.
- Elicit and construct fuzzy rules.
- Encode the fuzzy sets, fuzzy rules and procedures to perform fuzzy inference into the expert system.
- Evaluate and tune the system.

1.1.6. Tuning Fuzzy Systems

- Review modelled input & output variables, and if required redefine their ranges.
- Review the fuzzy sets, and if required, define additional sets on the universe of domain. The use of wide fuzzy sets may cause slow performance.
- Provide sufficient overlap between neighbouring sets. It is suggested that triangle-to-triangle and trapezoid-to-triangle fuzzy sets should overlap between 25% to 50% of their bases.
- Review the existing rules, and if required add new rules to the rule base.
- Examine the rule base for opportunities to write hedge rules to capture the pathological behaviour of the system.
- Adjust the rule execution weights. Most fuzzy logic tools allow control of the importance of rules by changing a weight multiplier.

- Revise shapes of the fuzzy sets. In most cases, fuzzy systems are highly tolerant of a shape approximation.

1.2. List of Problems

1. Fuzzy logic controller for an inverted pendulum.
2. Fuzzy logic used in simple games: tic-tac-toe (cross and circle), sea battle, etc.

1.3. Phases of Laboratory Exercises

1. Check and improve the necessary softcomputing knowledge.
2. Prepare and collect the necessary data sets: for training and for testing.
3. Realise the necessary data preprocessing and/or data postprocessing using different types of ready-to-use software or by "hand-made" software prepared by the laboratory group.
4. Prepare your own software to implement the proper softcomputing algorithms. The main goal is to create the correctly working engine, user interface utilities are not so important. The software environments and systems you can use for implementation are limited, but the actual possibilities ought to be discussed with the laboratory supervisor.
5. Turn on and tune the prepared software engine, supply the input training and/or testing data. If the engine works correctly, check what happens when the starting point parameters change, explore the sensitivity of engine for the different sets of available parameters and find the solution of the problem. At the end, check if the used softcomputing solution is correctly fitted to the problem.
6. Prepare the final report including the following parts:

   – the short description of the problem with necessary assumptions,

   – definitions and descriptions of the training and testing sets of input data with description of the preprocessing procedures,

– definitions and descriptions of the of output data with description of the postprocessing procedures,

– description of the tuned topology and parameters of the prepared softcomputing engine,

– detailed results analysis and final remarks.

1.4. Hints for the List of Problems

Problem No. 1

1. The general idea is to substitute the real PID controller by fuzzy logic based engine. The Multilayer Perceptron should be used to emulate the inverted pendulum.
2. Use the typical for fuzzy logic observations to create the necessary control mechanisms for the inverted pendulum driving.
3. Take the set of well-known observations describing the real inverted pendulum to train the "MLP pendulum".
4. It is necessary to find and discuss the relations: velocity, force, angle, acceleration.

Problem No. 2

1. The general idea is to use fuzzy logic combined to other softcomputing methods to build better than typical algorithm for a simple game.
2. The proposed method ought to improve the game algorithm based on the previous runs.
3. Try to compare the results when the softcomputing methods are in use to the typical algorithm results.

References

1. Ch. M. Bishop, *Neural Networks for Pattern Recognition,* Clarendon Press, Oxford, 1996
2. B. Bouchon Meunier, R. R. Yager, L. A. Zadeh, *Fuzzy Logic and Soft Computing,* Advances in Fuzzy Systems – Applications and Theory, Vol. 4, World Scientific Publishing, 1995
3. R. Hecht-Nielsen, *Neurocomputing,* Addison-Wesley, 1994
4. O. Castilo, A. Bonarini, *Soft Computing Applications*, Advances in Soft Computing, Springer Berlin Heidelberg, 2008
5. M. Caudill, Ch. Butler, *Understanding Neural Networks,* MIT Press 1995
6. E. Damiani, L. C. Jain, M. Madravio, *Soft Computing in Software Engineering, Studies in Fuziness and Soft Computing,* Springer Berlin Heidelberg, 2009
7. S. Y. Kung, *Digital Neural Networks*, Prentice-Hall, 1998
8. D. K. Pratihar, *Soft Computing,* Science Press, 2009
9. S. N. Sivanandam, S. N. Deepa, *Principles of Soft Computing,* Wiley, 2008
10. A. K. Srivastava, *Soft Computing,* Narosa Publishing House, 2009
11. D. A. Waterman, *A Guide to Expert Systems,* Addison-Wesley, 2005
12. D. Zhang, *Parallel VLSI Neural System Design,* Springer Berlin Heidelberg, 2006