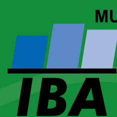
Milan Blaha

Artificial Intelligence

January 2015



This work was supported by the Fund of Development of Masaryk University, project MUNI/FR/0025/2014 – Development of foreign language textbook for the course “Artificial Intelligence” as part of the “Computational Biology” study programme.

Tento text vznikl s podporou Fondu rozvoje MU (MUNI/FR/0025/2014) - Příprava cizojazyčných výukových textů předmětu „Umělá inteligence“ zařazené ve studijním oboru „Matematická biologie“.

Artificial Intelligence
Milan Blaha, Ph.D.

Reviewer: Prof. Ivo Provazník, Ph.D.
Proofreading: Jakub Gregor, Ph.D.
Graphic design and typesetting: Markéta Soukupová, M.Sc.
Cover photo: http://naukawpolsce.pap.pl/

© 2015 Milan Blaha, Ph.D.
© 2015 Masarykova univerzita

ISBN 978-80-210-7728-7

CONTENTS

1 INTRODUCTION 4

1.1 BASIC INFORMATION 4

1.2 LEARNING OUTCOMES 4

1.3 ARTIFICIAL INTELLIGENCE (AI) – INTRODUCTION 4

1.4 ASSESSING THE INTELLIGENCE OF A MACHINE LEARNING ALGORITHM 6

1.5 LITERATURE 8

2. STATE SPACE SEARCH 9

2.1 BASIC INFORMATION 9

2.2 LEARNING OUTCOMES 9

2.3 INTRODUCTION 9

2.4 DEFINITION OF THE STATE SPACE 9

2.5 SEARCH METHODS 12

2.6 LITERATURE 25

3 EXPERT SYSTEMS 26

3.1 BASIC INFORMATION 26

3.2 LEARNING OUTCOMES 26

3.3 EXPERT SYSTEM (ES) 26

3.4 COMPONENTS OF EXPERT SYSTEMS 26

3.5 RULE-BASED EXPERT SYSTEMS 27

3.6 NON-RULE-BASED EXPERT SYSTEMS 30

3.7 UNCERTAINTY IN ES 32

3.8 LITERATURE 35

4. NEURAL NETWORKS – THE SINGLE NEURON 36

4.1 BASIC INFORMATION 36

4.2 LEARNING OUTCOMES 36

4.3 INTRODUCTION TO NEURAL NETWORKS 36

4.4 SINGLE NEURON 40

4.5 ADAPTATION DYNAMICS OF THE NEURON 42

4.6 CLASSIFICATION CAPABILITIES OF THE SINGLE NEURON 47

4.7 LITERATURE 51

5 NEURAL NETWORKS – PERCEPTRONS 52

5.1 BASIC INFORMATION 52

5.2 LEARNING OUTCOMES 52

5.3 FEED-FORWARD NEURAL NETWORKS 52

5.4 SINGLE-LAYER PERCEPTRON 55

5.5 MULTILAYER PERCEPTRON 56

5.6 LITERATURE 68

6. NETWORKS WITH MUTUAL RELATIONS 69

6.1 BASIC INFORMATION 69

6.2 LEARNING OUTCOMES 69

6.3 GENERAL CHARACTERISTICS OF ARTIFICIAL NEURAL NETWORKS WITH MUTUAL RELATIONS 69

6.4 HOPFIELD NETWORK 70

6.5 BOLTZMANN MACHINE 76

6.6 BIDIRECTIONAL ASSOCIATIVE MEMORY (BAM) 78

6.7 LITERATURE 79

7. COMPETITIVE NETWORKS 81

7.1 BASIC INFORMATION 81

7.2 LEARNING OUTCOMES 81

7.3 A SIMPLE COMPETITIVE NETWORK MAXNET 81

7.4 HAMMING NETWORK 84

7.5 SELF-ORGANIZING MAPS 85

7.6 LITERATURE 89

8. INTRODUCTION TO GENETIC ALGORITHMS (GA) 90

8.1 BASIC INFORMATION 90

8.2 LEARNING OUTCOMES 90

8.3 BASIC CONCEPTS OF GENETIC ALGORITHMS 90

8.4 GA TASKS 94

8.5 LITERATURE 95

1 INTRODUCTION

1.1 BASIC INFORMATION

The following text forms an integral part of studying materials for the course of Artificial Intelligence (AI) and is primarily intended for students of the Computational Biology study programme. This introductory chapter describes the artificial intelligence as a scientific discipline, and explains the basic terms of AI. Students will get to know the main areas in which AI algorithms are applied.

1.2 LEARNING OUTCOMES

After studying the text, the students should be able to:

- get to know and understand the basic terms of artificial intelligence,
- give examples of AI applications in practice
- get to know the basic classification of artificial intelligence according to the type of solved problems and the ways of their solution

1.3 ARTIFICIAL INTELLIGENCE (AI) – INTRODUCTION

1.3.1 What is artificial intelligence, definition

At the very beginning, let us have a think about the term "artificial intelligence", which is the name of this empirical science. This compound noun consists of two words: "artificial" and "intelligence". The meaning of "artificial" can be easily understood as "assigned to machines", or as "possible to be done by machines". The term "machine" in this context should be perceived rather as an abstraction, a nonliving entity, a model of which way of specific implementation is not important. In other words, we emphasise the fact that the intelligence is artificial, different from the natural intelligence. But who is naturally intelligent? How can intelligence be measured? Are all people intelligent? Are people more intelligent than, for example, than a community of ants in a rainforest? And would all university graduates survive in a rainforest?

There are many definitions of intelligence. With regard to the focus of the AI course, we can use the definition from the Czech version of Wikipedia, which will be entirely sufficient for our needs. "**Intelligence** (derived from the Latin word inter-legere, to comprehend or perceive) is a cognitive ability to resolve newly arising problems or difficulties; an ability to learn from experience; an ability to adapt; an ability to pinpoint significant links and relations which might be then used to solve new problems and to orient oneself in difficult situations."

This and many other definitions of intelligence have one common feature, and that is the statement that intelligence is an ability which makes it possible for its bearer to solve effectively a given type of problems in a defined, but changing environment. The environment is significant here; intelligent behaviour is always defined in relation to the knowledge space (variables).

Does it imply that animals are intelligent? Yes, they are intelligent in their environment, although teaching your dog how to play chess can be somewhat tricky. Let us simplify the situation and suppose that all people are intelligent to a certain measure. Self-importantly, we consider ourselves humans, our ways of behaviour and thinking, to be the benchmark of intelligence. Human intelligence is rather structured, focusing on individual areas, so we can talk about the verbal intelligence, social intelligence, mathematical intelligence, and so on.

We will therefore assess intelligence with regard to the extent of knowledge as perceived by humans. Intelligence is usually considered to be a person's ability to solve problems in a given area as effectively as possible. Psychologists have attempted to quantify these abilities by intelligence tests; the most familiar of them is an IQ test that was introduced in 1912 by William Stern.

Intelligence can thus be perceived as the ability to think and act as a smart human.

The above-mentioned definition of the term "artificial intelligence" can be supplemented by two other definitions. First, a somewhat pessimistic definition stating that "AI is the study of how to make computers do things at which, at the moment, people are better" (Rich, Knight, 1991).

And second, the widely used definition by Marvin Minsky from 1976, to which we will keep: "Artificial intelligence is the science of making machines do things that would require intelligence if done by men."

1.3.2 Intelligent machines

We have stated that a human is the benchmark of intelligence. But there are two aspects of intelligence and we must distinguish between them. The first aspect is whether the intelligence bearer acts in an intelligent way. In other words, we assess only external manifestations of a given subject when solving specific tasks. The second aspect is somewhat deeper: is the given subject conscious, and does it realize his outward signs of intelligent behaviour?

What do we expect from intelligent machines? Both humans and machines **react externally** to a specific problem in a certain way, and their reactions are based on their **internal reasoning**. But do humans always act in a rational way? They do in most cases, but not always. They are frequently influenced by emotions and by stress: tense situations make them act in ways which are far from rational. People in such situations act according to their feelings which might (but do not necessarily have to) result in the most rational solution to a give problem, such as putting out the fire, or dealing with a car accident.

In terms of intelligent machines, do we want machines that:

A. *Act like humans (model of human behaviour)?*
B. Act in a rational way, i.e. make the "right choices" (when it comes to finding the best solution to a problem)?
C. Infer in the same way as humans (do they come to the solution in the same manner as humans do)?
D. Infer in a rational way (do they use the most effective algorithm)?

This cannot be decided unequivocally. In fact, it would be very difficult to evaluate the internal reasoning (or "computational processes") in both humans and machines, as indicated in C above. The question is whether the above-mentioned classification is worthwhile at all. None of us can tell what specific process a given person has used to achieve the result (did the student copy the solution from someone else, or did he use his/her knowledge?). When it comes to machines, the situation is often simpler: in many cases, we are able to evaluate the internal algorithm used by the machine in order to achieve external results.

From the viewpoint of artificial intelligence, the above-mentioned point B is the most important criterion to assess a given machine: we expect from intelligent machines to find a rational solution to any given problem, which makes them so different from humans. At the same time, it is often required from a machine to use the most effective algorithm as mentioned in D above; this might not be the most important requirement, although it is certainly justified in many cases. AI as a scientific discipline generally aims to seek effective and rational algorithms, and such algorithms will also be the goal of our endeavour. This course aims to help students understand the basic algorithms and models which enable machines to simulate a rational human behaviour (to a limited extent and in a limited space), which is based on algorithms as effective as possible. The so-called weak artificial intelligence is implemented in such machines, as mentioned in the following text. In other words, students should acquaint themselves with algorithms and models used for the construction of machines that show signs of rational human behaviour in precisely defined situations.

1.3.3 Weak and strong AI

There is a controversy whether such classification has any sense at all. We can achieve weak artificial intelligence in a machine by modelling a weak understanding. A weak (or Turing) understanding is perceived as a level of understanding when a system makes corresponding reactions to correct input stimuli. The next chapter will introduce the concept of the Turing test, which proves whether a given machines shows signs of a weak understanding, or whether it acts like a smart human.

On the contrary, we can achieve strong artificial intelligence in a machine by modelling a strong understanding. A strong (or Brentan) understanding is perceived as a level of understanding when a system has the same feeling of understanding like a human mind, i.e. when it can think in the same way as humans do. However, any evaluation of strong artificial intelligence is rather problematic, as described in the preceding paragraphs.

In principle, implementation of strong artificial intelligence would lead to the design of a new, artificial human, while the weak artificial intelligence helps to formalize certain specific areas of human thinking and acting. Weak AI designs algorithms that are better than humans when it comes to the solution of specific problems. These are, for example, various games, tasks for machine control, optimisation, modelling, and many other types of problems. The important fact is that all cases of weak AI apply to a clearly defined and described area; a universally intelligent algorithm has not been discovered yet. This course will introduce algorithms that can be applied in the area of weak AI.

1.4 ASSESSING THE INTELLIGENCE OF A MACHINE LEARNING ALGORITHM

1.4.1 Turing test

The Turing test (TT) was defined with the objective of proving the intelligence of a machine or an algorithm. It was designed by Alan Mathison Turing (1912–1954), a prominent British mathematician and founder of the modern computer science [http://en.wikipedia.org/wiki/Alan_Turing]. This became most known for decrypting German electro-mechanical encryption machine Enigma messages during the World War II, or for designing the theoretical concept of the Turing machine, which is one of the basic models in computer science.

Turing test is basically an imitation game during which a human judge is separated from an area where a machine and a human are present (see Fig. 1.1).

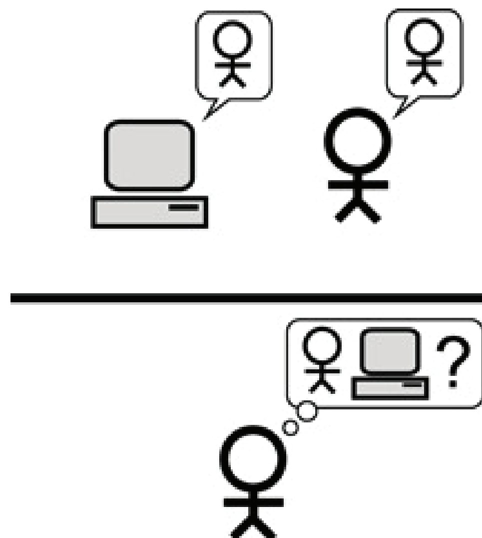


Fig 1.1 Turing test

In the original illustrative example, a human judge engages in natural language conversations with a human and a machine designed to generate performance indistinguishable from that of a human being. All participants are separated from one another. If the judge cannot reliably tell the machine from the human, the machine is said to have passed the test. Passing the Turing test is considered to be the necessary condition to prove a weak intelligence. The Turing test is far from being perfect, as shown on the example of the "Chinese room" in the following text. Interestingly, the Loebner Prize – determined for the first machine which would pass the Turing test – has not been awarded yet. In order to pass the Turing test entirely, a machine would have to be able to process the natural language, to recognise terms, to remember what has been said during the conversation so far, to judge from those facts, to make conclusions, and to adapt to new situations and topics. This is not an easy task for a human, even less for a machine.

1.4.2 Chinese room

Passing the Turing test does not necessarily have to be the manifestation of a strong intelligence. Let us imagine a library containing all meaningful Chinese sentences written on pieces of paper. After being asked a question, the subject in the room will choose an answer from the library, based solely on the similarity with the question, without having to understand that question. To a certain extent, the object complies with the task assigned in the Turing test, but is not intelligent from the viewpoint of strong intelligence reasoning. The object has no idea of what is going on; it can only mechanically correlate similarities between questions and answers, and is able to search the library of answers very quickly. This principle is employed in chatter robots (or chat bots) – computer programmes which communicate with humans in natural English language. ELIZA ***http://www.manifestation.com/neurotoys/eliza.php3)*** and ALICE ***(http://alicebot.blogspot.com)*** are the most familiar examples. Such bots employ a library of standardised answers to standard expectable questions. Most notably, they attempt to repeat, modify or develop human statements or questions, and to generate simple statements such as "Do you really think so?" or "I understand." The entire "conversation" is based on a syntactic processing of the English language, without any understanding to the text. The following example is adopted from ***http://cs.wikipedia.org/wiki/ELIZA***.

> Hello, I am Eliza.
* Hello
> How are you today.. What would you like to discuss?
* I am depressed.
> Did you come to me because you are depressed?
* Yes
> I see.
* It is because I have problem with my parents.
> Can you elaborate on that?
* Yes, I think it is because of my dad.
> Do you really think so?
* Yes, he can be really moody sometimes.
> I understand.
...

1.4.3 Applications of AI

The human and his ability to solve problems present a benchmark for us that we are trying to achieve. Turing test can be considered to be an imperfect test of intelligent machines. The future of AI can be very interesting and can even lead to humanoid (human-like) robots, but the present is much less ambitious. Based on their imperfect sensors, contemporary machines learn to walk and to maintain their balance. Practical applications of AI proved to be successful in several areas of the human activity. This course aims to help you understand the design of effective algorithms solving tasks in the following areas:

Algorithms for state space search perform an "intelligent" search of tasks in a space of given states, and verify them with the objective of finding a possible solution. State space search algorithms are typically applied

on problems for which individual states of a task can be easily generated by an algorithm. Chess and several other games are typical examples of tasks that can be solved by such algorithms. However, algorithms for state space search have also proved successful in other areas of AI, for example as component algorithms in expert systems.

Expert systems are computer programmes intended to decide instead of a human expert, to give advice, and to recommend expert procedures in specific tasks. Control of a device based on external inputs is a typical example of simpler tasks for expert systems. Assistant systems in medical diagnostics are examples of more complicated tasks. Expert systems are mostly based on databases of rules that are applied in the decision-making process, taking into account the available information.

Artificial neural networks (ANNs) are represented by numerous models that have one feature in common: similarly to a human brain, they are composed from a huge number of interconnected, mutually communicating, and relatively primitive computational units – neurons. These units can be arranged and interconnected in numerous topologies, always with respect to a specific task. Feedforward neural networks (perceptrons) are the most common ANNs. Their applications are very universal due to the fact that they are able to learn from presented examples, and do not require any further information about the problem being solved.

Genetic algorithms have been also inspired by biology – or more specifically, by genetics and the evolutionary theory. In this class of algorithms, the initial solution (or a group of such solutions) is represented by an individual (or a population of individuals). The best solutions are subsequently selected from this population, and reproduced. Individuals from the new (and better) population are then verified if they represent the sought solution or not; if not, the best ones are again selected, and so on. In this manner, the population gradually evolves towards its optimum. Although this is a general approach to solving problems rather than a separate area, genetic algorithms are also considered to be part of AI.

1.5 LITERATURE

[1] Šíma, J., Neruda, R.: Teoretické otázky neuronových sítí, MATFYZPRESS, 1996, ISBN 80-85863-18-9.
[2] Volná, E.: Neuronové sítě 2. Skripta Ostravská universita v Ostravě, Ostrava, 2008.
[3] Kvasnička, V., Beňušková, L., Pospíchal, J., Farkaš, I., Tiňo, P., Kráľ A.: Úvod do teórie neurónových sietí, IRIS, Bratislava, 1997, ISBN 80-88778-30-1.
[4] Pavlovič, J.: Multikriteriální hybridní evoluční algoritmy pro výběr a optimalizaci dekontaminačních technologií, 2008. Rigorózní práce, Masarykova univerzita, http://is.muni.cz/th/4035/fi_r/.
[5] http://www.milosnemec.cz/clanek.php?id=188
[6] http://survivebatchanimals.7x.cz/uvod-do-filozofie-umele

2. STATE SPACE SEARCH

2.1 BASIC INFORMATION

The following text is part of the course "Artificial Intelligence" and is intended primarily for students of Mathematical Biology study programme. This chapter summarizes the class of methods of artificial intelligence, collectively known as methods of state space search. It describes the basic uninformed methods as well as methods based on heuristic algorithms, including their mutual comparison. Uninformed methods seek task solutions blindly, passing through all possible states in which the task may occur. Unlike the methods using heuristics, these methods do not try to assess the closeness of the state of the given task to a solution.

2.2 LEARNING OUTCOMES

Mastering the learning text will enable students to:

- *Understand and define the basic concepts from the field of state space search, such as state space, transition operator, target task state, target depth, path length, branching factor, and the heuristic function.*
- *Algorithmically describe the various search methods.*
- *Perform mutual comparisons and evaluations of search algorithms with respect to their entirety, space and time complexity, and optimality of the found solution.*
- *Solve model tasks using search algorithms.*

2.3 INTRODUCTION

This chapter discusses a set of algorithms of artificial intelligence that we summarize under the common name "algorithms for state space search". First, it should be clarified which task class is usually solved using these algorithms. Generally, the algorithms are used to solve problems where we are unable to find the solution by direct calculation based on available information, or where the realization of the calculation for finding a solution is prohibitively difficult. However, we must be able to describe the instantaneous state of the problem solution.

An example might be finding the shortest path between two cities, the winning in a game of chess, or solving conundrums like "Towers of Hanoi" or "Loyd's fifteen". The lack of easy solutions implemented by direct calculation does not mean that we are not able to define the control strategy which will find the solution. For example, in the case of the "Loyd's fifteen" conundrum, we try to arrange numbered stones by size in a 4x4 grid from the initial setup of 15 stones and thus reach the conundrum solution. Our strategy can also be as follows: we try to use a trial-error method and implement random moves that are permitted at the given moment. By these permitted moves, we gradually generate new configurations from the initial configuration and always evaluate whether the resulting configuration is the solution (stones are arranged) or not. In this way, we select a control strategy that always randomly chooses one of the possible moves and relocates the stone. Such a procedure is possible but not very effective. Algorithms for state space search try to optimize these search strategies.

2.4 DEFINITION OF THE STATE SPACE

State space S_p can be a pair

$$S_p = (S,\Phi) \qquad (1)$$

where $S = \{s\}$ is a set of all possible states in which the task can occur and $\Phi = \{ \varphi \}$ is a set of operators for transition between states. It is true that the k-th state s_k is obtained when applying the transition operator φ_{ik} to the state s_i; s_i is a predecessor of the state s_k while s_k is the successor of the state s_i.

$$s_k = \varphi_{ki}(s_i) \qquad (2)$$

The initial task state, in which the solution occurs when starting the search, is denoted as s_0.

Next, we define the set G = {g} of target states that represent the sought solution. The solution can be just one, but target states representing the solution may be more. The set G is a subset of the set S.

2.4.1 Representation of the state space

The definition of the state space and the possibilities of its encoding will be explained using the example of solving the "Towers of Hanoi" conundrum.

The task principle lies in the fact that we have three bars and three discs of different diameters set on them.

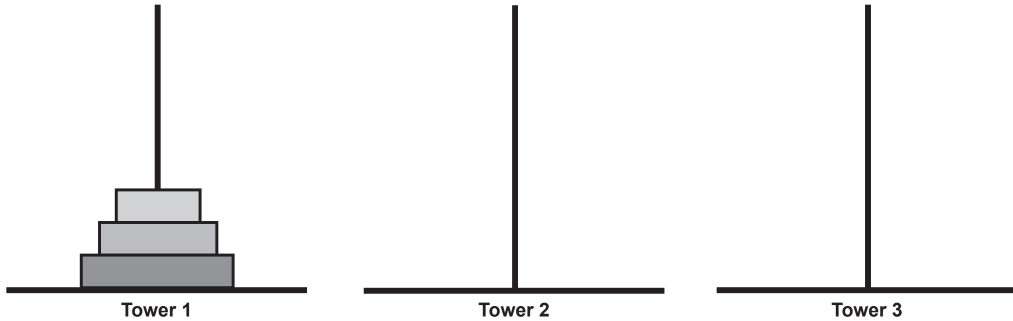


Fig. 2.1 "Towers of Hanoi" conundrum

The task is to move all the discs from the first tower to the second tower using temporary stacking of discs on the third tower. For arranging and moving the discs, the following rules apply:

- *In one move, only one disc can be displaced*
- *A move consists of picking the disc from the top of one tower and its relocation to the top of the other tower*
- *It is prohibited to lay a larger disc on a smaller disc*

The initial state s_0 is, therefore, a configuration when all discs are on the first tower, arranged from the largest disc to the smallest one. The set of target states contains just one state g, a state where all discs are correctly arranged on the second tower.

Introduce a coding that will express the specific state of the task solution using ordered triples of numbers where the position of numbers in the brackets specifies the discs from the largest disc to the smallest one, and the value of the number in a given position expresses the tower on which the disc is located. Therefore, the initial state s0 can be written as (111); if we move the smallest disc to the second tower, the state will be encoded as (112), and the target state g will be (2,2,2).

Overall, the task can have 3n states for three bars where n represents the number of discs. Each of the discs must be located on one of the three bars. In our case, for the three discs when each of them must be on one of the three bars (three location options per one bar for each disc, three discs in total), we obtain 3x3x3 possible states. The set of all possible states S = {s} is thus represented by 27 elements.

In this case, the set of operators has 6 elements "Move the top disc from the tower k on the tower i", i.e. $\Phi = \{ \varphi_{12}, \varphi_{13}, \varphi_{21}, \varphi_{23}, \varphi_{31}, \varphi_{32} \}$.

Individual states can be arranged into a directed graph where the root node is the initial state; from this state, through the application of all relevant operators, the following states are gradually generated. Permissible

operator is an operator whose application is not prohibited by the task rules, such as "it is prohibited to place a larger disc on a smaller one".

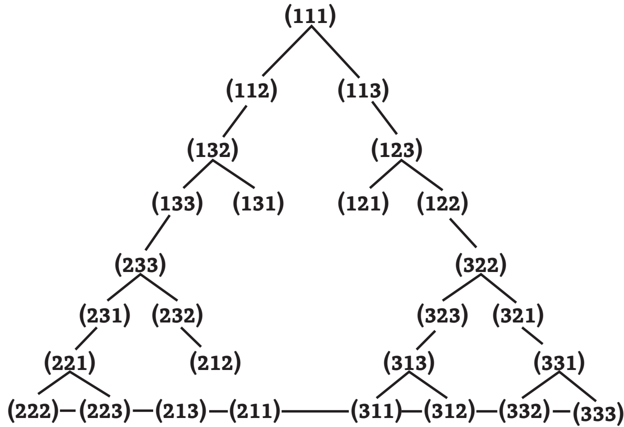


Fig. 2.2 State space of "Towers of Hanoi" conundrum

2.4.2 The task in state space

In the case of simple task of "Towers of Hanoi", it is possible to easily generate its entire state space and visualize it in the form of a directed graph. Therefore, the task solution in state space is finding a sequence of operators whose successive application will lead us from the initial state s0 to one of the target states of the set G. In our case, it is the path from state (111) to state (222).

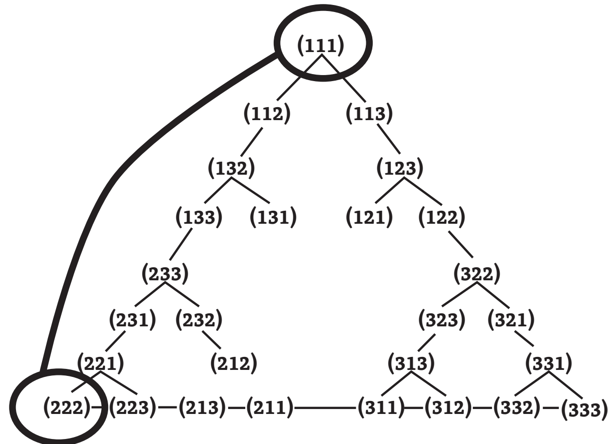


Fig. 2.3 State space of "Towers of Hanoi" conundrum, the target state

The number of applications of operators is called the number of steps or, preferably, the path length. The found sequence of operators, i.e. the actual solution, is then called the found path from the initial to the target state. There can be more target states as well as found paths leading to them; therefore, we usually try to find a path that is optimal in some respect. In the case of uniform assessment of edges, it is the shortest path from the initial state to a target state. From the solution point of view, the implementation of individual edges can be evaluated differently, can have different cost (demandingness) of making the transition; in such case, it is the path with minimum cost. The cost of applying individual operators of the set Φ may not be the same for all these operators.

A real task in the state space is, therefore, solved as follows:

- *Establish a coding of individual states.*
- *Identify the operators and limiting conditions for their application.*
- *Identify the target states representing the solution to the task.*
- *Select the initial state. It is usually given by the task assignment.*
- *Select the strategy for browsing individual states, i.e. the strategy of applying individual operators – space search.*
- *Apply the chosen strategy, start with the initial state.*
- *Check at each step whether the actual achieved state (or the path to it) is the sought solution.*

State space can be very large or generally infinite, and it is not always possible to examine all possible states and transitions between them in finite time as in the case of the previous task. Here, we use different strategies for state space search, which try to optimize this process. These methods will be demonstrated in the following sections.

2.5 SEARCH METHODS

The next chapter briefly focuses on the basic methods of state space search, which occur in a number of modifications. In general, the methods for searching the state space can be divided as follows:

- *Uninformed*
- *Informed*
- *Local*

Uninformed methods apply individual operators blindly; they pass through state space without any a priori information on the benefits of applying a particular operator. Usually, for a certain state, all the transitions (all operator applications) are therefore equally likely. These methods do not assess at all whether the actual state or the path to this state is advantageous, and do not try to estimate its distance from the target. Nevertheless, they are simple and do not require calculating any complicated evaluation functions.

In contrast, the informed methods try to estimate whether the transition from a particular state is preferred over another from the solution point of view, based on simpler or more complex function. If psychology is set aside, these algorithms act like a chess player who tries to decide on the next move based on the assessment of the current situation. At the given moment, the chess player is not able to evaluate all possible combinations (search all states) for more moves ahead; he/she, therefore, uses heuristics and selects the move that seems to be the most advantageous at that moment. The player with better heuristics usually wins.

A special class is represented by local methods where the major advantage is in their relatively small demands on memory and ease of implementation. In principle, these are informed methods that evaluate only the advantageousness of the closest successors of the given state and do not memorize the previous states. Therefore, they behave locally, do not return back in the directed graph, and proceed only forward until the moment when all successors are less suitable than the current node. The biggest problem with these methods is the fact that they can get stuck in a local minimum.

2.5.1 Evaluation of methods

When opting for a suitable method of state space search, we must be able to describe and compare these methods. To do this, we use the basic parameters characterizing the methods. Suppose a directed graph searched by the method in the form of a tree as shown in the following figure.

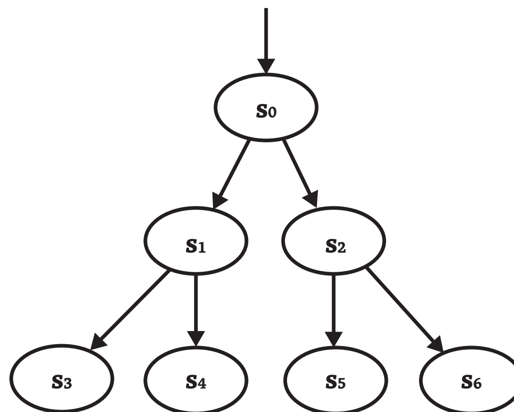


Fig. 2.4 The search tree. The node is represented by the appropriate task state and other information necessary for searching with the given algorithm

Introduce the following concepts:

a) Branching factor, denoted as b.
b) Target depth, denoted as d.
c) Maximum path length, denoted as m.

Ad a) Branching factor b indicates how many successors can be generated for each state on average. In the case of evaluating the performance of some heuristics, we also use the concept of effective branching factor, denoted as b*. The effective branching factor indicates a branching value that would correspond to an evenly balanced tree of depth d with the number of states (nodes) N, after finding the target in depth d and the number of expanded states N. In case of completely perfect heuristics, the effective branching factor would be equal to one.

Ad b) Target depth d denotes the number of tree levels through which we have to pass from the root (the initial state) to the target at the lowest level.

Ad c) The depth is calculated from the root that lies at the 0 (zero) level. The maximum path length corresponds to the longest possible path in state space, which is implemented (generated) during the search.

Regarding the state space search methods, we are interested in:

a) whether the method is complete.
b) whether the method optimal.
c) its time complexity.
d) its space complexity.

Ad a) We say that the method is complete if it finds a solution every time it exists.

Ad b) Optimal method, in addition, finds the best solution. It finds a solution with the shortest path in case of uniformly evaluated transition edges, or a solution with the lowest cost of these transitions in case of their nonuniform evaluation.

Ad c) Time complexity indicates the number of steps, i.e. the number of applications of transition operators, required to find the solution.

Ad d) Similarly, the space complexity indicates how much memory (usually in terms of the number of retained states) is needed for the algorithm.

This will allow us to characterize each algorithm for state space search in terms of guarantee on finding (the best) solution. The parameters of time and space complexity allow us to estimate the real demandingness of their implementation on the PC in terms of the required processor computing time (time complexity) and in terms of the required operating memory of the computer (space complexity).

2.5.2 Uninformed search

As already mentioned, a common feature of uninformed methods is that they do not use any valuation heuristic function when passing through individual states, which would estimate the benefits of successors of the actual state and choose the most gifted of them. Algorithms sequentially search through the search tree only depending on its topology; they are also called blind methods of state space search.

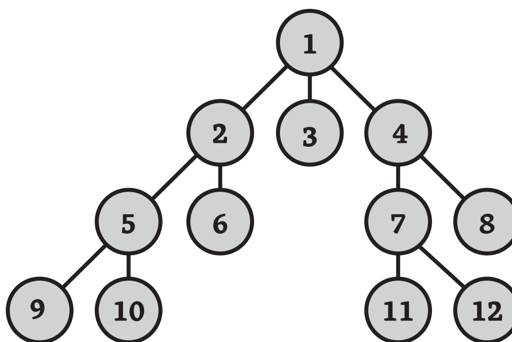


Fig. 2.5 The search tree – an example

Particular algorithms vary according to the order in which the states are generated, i.e. also passed through. Path generation can be expressed using the agreed rules for path generation recording; we start from the initial node 1, registered as [(1)], and gradually generate more nodes that are the result of applying the operator φ to the selected state. For example, when applying all admissible transition operators to the first state, we get the following record: [φ(1)] -> [(2,1)(3,1)(4,1)]. Then, the transition operator φ is applied to the next state from the generated set, e.g. (2,1). In this way, we get the generated sequence of states including the paths that led to them. It should be noted that there is a certain difference between the concepts of "node" and "state". The concept of state means just the task state as such, while the search tree nodes, in addition to the state itself, include also other information. Usually, it is the information on the path to the node (a reference to the parents), applied transition operators, and depth of the node in the search tree. In case of informed search, the information also includes node evaluation by a heuristic function.

In uninformed methods, we usually introduce two sets of states: the OPEN set that contains yet unproven, newly generated, states and the CLOSE set that in turn contains the states already proven, expanded. The states included in the CLOSE set are no longer included in the OPEN set when expanding the node, and are therefore examined only once.

In the following section, we introduce some uninformed state space search.

2.5.2.1 Breadth-first search (BFS)

Description

Along with the depth-first search method, it is a basic algorithm for state space search. BFS algorithm searches the tree of nodes gradually, through each level, and the next level of the tree is searched only after searching all the nodes of the previous level.

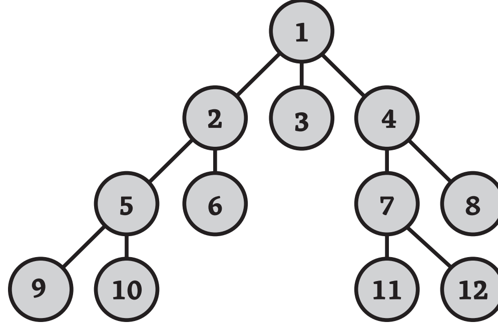


Fig. 2.6 The search tree – an example

For the previous graph, the start of generating the path according to BFS algorithm can be demonstrated by the following abbreviated entry:

[φ (1)] -> [φ (2,1) (3,1) (4,1)] -> [φ (3,1) (4,1) (5,2,1) (6,2,1)] -> [φ (4,1) (5,2,1) (6,2,1)] -> [φ (5,2,1) (6,2,1)(7,4,1) (8,4,1)] -> ...

For this sequence, the form of OPEN and CLOSE sets of browsed states will be as follows:

OPEN[1], CLOSE[] -> OPEN[2, 3, 4], CLOSE[1] -> OPEN[3, 4, 5, 6], CLOSE[1, 2]-> OPEN[4, 5, 6], CLOSE[1, 2, 3] -> OPEN[5, 6, 7, 8], CLOSE[1, 2, 3, 4] -> ...

Generating this path corresponds to the principle of data structure of FIFO (first in, first out) queue when the algorithm always selects the first leftmost node from the OPEN queue (set); this node is checked and expanded if it does not belong to the target states. Newly generated nodes are included at the end of the queue represented by the OPEN set. The algorithm works until it reaches the first target state or until the entire state space is searched through – the OPEN set is empty.

Features of the algorithm

The algorithm is complete for a finite branching factor b. It is also optimal in terms of the path length.

The time complexity of the algorithm is exponential; for the target g in the depth d and for branching factor b, the number of visited nodes can be expressed as:

$$O(BFS) = 1 + b + b^2 + \dots + b^d + b(b^d - 1) = (b^{d+1}) \qquad (3)$$

First, we go through the nodes of all levels preceding the level at which the target is located. This corresponds to the first summation part of the expression $1 + \dots + b^{d-1}$. If the target is in depth d and at the end of this level (in the worst case), then we also have to check all layers at the level where the target occurs, including the target itself – this corresponds to the value of b^d nodes in the expression. In addition, we must apply the φ operator to all nodes in this layer according to the BFS algorithm, except the target state. This fact is represented by the last part of the O(BFS) expression, i.e. $b(b^d-1)$.

Space complexity is exponential, as well as the time complexity. In the OPEN set, all nodes of the given tree level must be kept at depth d, which we have expanded. All nodes of all previous levels, already expanded, are maintained in the CLOSE set. In this case, the space complexity is as follows:

$$O(BFS) = (b^{d+1}) \qquad (4)$$

or, when using only the OPEN set, it can be insignificantly reduced to

$$O(BFS) = (b^{d}) \qquad (5)$$

Advantages and disadvantages

In case of anticipated uniform evaluation of edges, the main advantage is the optimality and completeness of the algorithm (provided that the branching factor b is final). For these benefits, however, the algorithm pays in the form of large memory complexity, i.e. it is practically difficult to use for large spaces because it can easily run out of physically available memory resources of the computer during the implementation. In comparison with other algorithms, the time complexity is just average.

2.5.2.2 Depth-first search (DFS)

Description

Together with BFS, the method belongs to basic search algorithms. DFS algorithm searches the tree of nodes as follows: It gradually proceeds through the leftmost branch of the tree into the maximum possible depth, i.e. to the end of this branch or until the target is found. Then, it returns one step back and searches analogously another branch of the currently processed node. Therefore, the algorithm does not pass through the tree gradually per each level as BFS, but recursively per each subtree of the given node. This search method can be simply understood as a recursive call for the function "Evaluate" – evaluate whether the processed node is the target and, if not, assess this condition for its successor from the left.

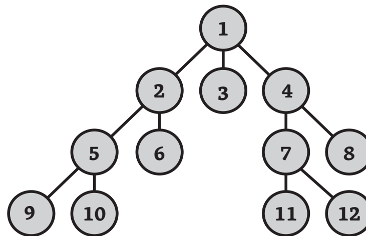


Fig. 2.7 The search tree – an example

For the previous graph, the start of generating the path according to DFS algorithm can be demonstrated by the following abbreviated entry:

[φ (1)] -> [φ (2,1) (3,1) (4,1)] -> [φ (5,2,1) (6,2,1) (3,1) (4,1)] -> [φ (9,5,2,1)(10,5,2,1) (6,2,1) (3,1) (4,1)] -> [φ (10,5,2,1) (6,2,1) (3,1) (4,1)] -> [φ (6,2,1) (3,1) (4,1)] -> [φ (3,1) (4,1)] -> [φ (4,1)] -> [φ (7,4,1) (8,4,1)] …

For this sequence, the form of OPEN and CLOSE sets of browsed (searched) states shall be as follows:

OPEN[1], CLOSE[] -> OPEN[2, 3, 4], CLOSE[1] -> OPEN[5, 6, 3, 4], CLOSE[1, 2] -> OPEN[9, 10, 6, 3, 4], CLOSE[1, 2, 5] -> OPEN[10, 6, 3, 4], CLOSE[1, 2, 5, 9] -> OPEN[6, 3, 4], CLOSE[1, 2, 5,9, 10] -> OPEN[3,4], CLOSE[1, 2, 5,9, 10, 6] -> OPEN[4], CLOSE[1, 2, 5,9, 10, 6, 3] -> OPEN[7, 8], CLOSE[1, 2, 5,9, 10, 6, 3, 4] …

Generating this path corresponds to the principle of data structure of LIFO (last in, first out) stack when the algorithm always selects the first leftmost node from the OPEN set (stack); this node is checked and expanded if it does not belong to the target states. Newly generated nodes are included at the beginning of the set, the stack

top. The algorithm works until it reaches the first target state or until the entire state space is searched through – the OPEN set is empty.

Features of the algorithm

For a generally defined tree, the algorithm is not complete as it can get stuck in an endless branch; however, it can be considered complete in case of finite length of the branches. The algorithm is not optimal; the solution found may not be the solution lying at the minimum depth, i.e. the solution with minimum length of the path from the initial node.

The time complexity of the algorithm is exponential as in the case of BFS; the number of visited nodes corresponds to the maximum depth of immersion m and, for the branching factor b, it can be expressed as

$$O(DFS) = (b^{m+1}) \qquad (6)$$

The space complexity is linear. At least, the memory must retain the OPEN set that contains only the currently searched branch through which the algorithm recursively returns in the case of not finding the target in this branch.

$$O(DFS) = (bm) \qquad (7)$$

Advantages and disadvantages

The main advantage is good space complexity that corresponds to the longest branch in the graph, i.e. relatively small requirements for computer memory in the implementation of the algorithm. Conversely, the disadvantage is the fact that the method is not complete for infinite general trees. However, this limitation can be relatively easily suppressed by method modification, as will be shown below. The method is also not optimal; in case of tasks where we only look for the best possible solution, this can be a reason for not using this method.

Time complexity is again just average. Regarding the time and space complexity in solving a real-life task, however, it should be noted and realized that the depth d of the found target can be much smaller than the maximum depth m, and the algorithm is thus not forced to actually expand the longest branch in all cases.

2.5.2.3 Limited depth-first search (LDFS)

Description

In principle, it is a method completely consistent with DFS but the maximum depth of allowable immersion of algorithm I is defined.

Features of the algorithm

The algorithm is not complete for a target located at a depth d greater than the maximum allowable immersion depth l. Therefore, it is not complete for $l<d(g)$. Otherwise, it is. The algorithm is not optimal.

Time and space complexity of the algorithm corresponds to DFS with limit depth l; therefore, LDFS time complexity is

$$O(LDFS) = (b^{l+1}) \qquad (8)$$

LDFS space complexity is

$$O(LDFS) = (bl) \qquad (9)$$

Advantages and disadvantages

They are again the same as in the case of DFS; LDFS only ensures that the algorithm is terminated in finite time and that the solution is found at smaller depth than the limit depth.

2.5.2.4 Iterative depth-first search (IDFS)

Description

The method is based on DFS or, more precisely, on the LDSF algorithm. The modification is as follows: Parameter l, indicating the maximum depth of immersion in LDFS method, is not constant but gradually acquires values from l = 1, 2 … n. Therefore, the method consists in repeated starts of the simple LDSF algorithm where the LDFS search is always triggered from the beginning for each gradually increasing parameter l until finding the target or finishing the state space.

Features of the algorithm

For final branching b, this algorithm is complete. It is also optimal

The time complexity is exponential, as in BFS and DFS.

$$O(IDFS) = d(b^1) + (d-1)b^2 + \dots + 1(b^d) \approx (b^m) \qquad (10)$$

The record of time complexity actually reflects the following fact: during repeated start of DFS algorithm, for a target in the depth d, we pass d-times through the first-level nodes of the tree, (d-1)-times through the nodes of the next level, etc. The DFS algorithm is, therefore, started d-times always with the maximum depth l greater by one. In reality, the time-complexity of IDFS method is often better than in BFS that adds expansion of one layer of nodes (at the level of the tree containing the target).

Space complexity is linear, absolutely the same as in DFS.

$$O(IDFS) = (bm) \qquad (11)$$

Advantages and disadvantages

First, when not knowing the size and the maximum depth of the investigated space, the method usually represents a chosen uninformed search method. It combines the advantages of DFS and BFS methods, i.e. relatively low memory requirements and especially, the completeness and optimality. Time complexity is only average. Regarding practical tasks, it is worse than DFS, but often better than BFS. For example, according to equations (10) and (3), for b = 10 and d = 5, the time complexity is as follows:

O(IDFS) = 50 + 400 + 3 000 + 20 000 + 100 000 = 123 450

O(BFS) = 10 + 100 + 1 000 + 10 000 + 100 000 + 999 990 = 1 111 100

2.5.2.5 Uniform cost search (UCS)

Description

This is a modification of the BFS algorithm. The method also belongs among the uninformed methods; however, it is not true that the transitions between states are equally likely. Compared to BFS, the edges of individual transitions in the graph are evaluated by different known values reflecting the demandingness, the cost of the transition in applying a given operator. Browsed (searched) nodes of successors are arranged in a queue prioritizing the lower cost of the transition to a successor, and are therefore browsed (searched) in that order.

Features of the algorithm

The algorithm features are again equivalent to the BFS algorithm. The algorithm is complete and optimal for non-zero cost of transitions greater than ε, where ε is a constant.

Time and space complexity is equal to

$$O(UCS) = O(b^{1+[C^*/\epsilon]}), \text{ where } C^* \text{ is the cost optimal solution.} \quad (12)$$

Advantages and disadvantages
They are exactly the same as in the first BFS algorithm.

2.5.2.6 Summary of uninformed methods

Due to its versatility, the preferred uninformed search algorithm is the IDFS algorithm. A summary of the features of described algorithms is shown in the following table.

Feature	BFS	UCS	DFS	LDFS	IDFS
Completeness	yes (for finite b)	yes	no (yes for finite branch lengths)	Yes, l >= d	yes
Optimal	yes	yes	no	no	yes
Time complexity	$O(b^{(m+1)})$	$O(b^{(1+[C^*/\epsilon]})$	$O(b^{(m+1)})$	$O(b^l)$	$\approx O(b^m)$
Space complexity	$O(b^{(m+1)})$	$O(b^{(1+[C^*/\epsilon]})$	O(bm)	O(bl)	O(bm)

Tab. 2.1 Summary of the features of uninformed search methods

2.5.3 Informed heuristic search

Uninformed search methods are successful at finding targets; with regard to time and space complexity, however, they cannot be used for large state spaces, not even in optimized variants. Therefore, we try to replace these algorithms (which search the state space using a "brute-force") with "smarter" algorithms. We want to find a method that would utilize some additional knowledge about the problem solved and use a heuristic method based on this knowledge, leading to faster finding of the sought path with high probability and least resources. Consequently, the methods seek to proceed on the basis of some heuristics like people who get lost in the forest, for example, and look for a way out of it. Such people can follow different algorithms; for example, they can always head to places where the trees are least dense, follow water flows, etc. However, they cannot know whether the way leads to the target or whether it is the shortest way; nevertheless, with a high degree of probability, an appropriate heuristics will finally lead them to the target.

In case of informed search, we are again looking for a way out of the initial state to the target states. However, to generate a potential path to the target, we use some other a priori information about the task to be solved. We implement evaluation of each n-th node by the following function

$$f(n)=h(n) + g(n) \quad (13)$$

where $h(n)$ is an estimate of the cost for the path from the n-th node to the target, and $g(n)$ represents the cost for the path from the initial state to the actual n-th node.

The order of nodes searched is then determined by the value of the function $f(n)$ so that the promising states with the smallest $f(n)$ value are checked first. The value of the function $g(n)$ can be expressed for the current state n exactly; it represents an assessment of the path already completed. The function $h(n)$ is an estimate of the cost of getting to the target state. The lower the $h(n)$ value, the closer the given state is to the target. The function $h(n)$ is, therefore, only an estimate; the method of its calculation is determined by people based on their experience and knowledge of the problem solved. The quality of the heuristic function $h(n)$ then directly affects the efficiency of finding a solution.

Selecting the optimal heuristic function may not be trivial. When looking for such a function, we usually proceed by problem relaxation. Problem relaxation is a mechanism of finding the admissible heuristics. Admissible (optimistic) heuristics is considered such heuristics that do not overevaluate the cost of the path to the target, i.e. realistic assessment of the cost of the path to the target will always be higher than the result of the heuristic function.

Relaxation of the problem consists in its substantial simplification; we actually ignore certain limitations of the task that make the problem more complicated. However, the found heuristics may be far from optimal and may not lead to significant improvements in the features of the search algorithm. Finding a good heuristic function is, therefore, very substantial for solution effectivity. Again, let's demonstrate everything on the example of "Loyd's fifteen" conundrum when the task is to order 15 numbered stones in a matrix of 4x4 elements, through their movement using one free bar. We are considering which stone should be moved, i.e. into which subsequent state we should get. We want to determine heuristics h(n) which will predict how far is the given state from the target. There are more possibilities - for example, we can quantify the number of incorrectly spaced stones for each state or the sum of distances of the stones from their correct positions. In the next move, we can choose such a stone move that will lead to a state with the minimum value of this function, as will be shown below for the Greedy search method.

2.5.3.1 Greedy search (GS)

Description

The algorithm represents the simplest variant of informed search. It does not follow the cost of the path to the given node; the function g(n) = 0. Evaluation function f(n) is thus reduced only to estimation of the distance from the n-th node to the target; f(n) = h(n).

The algorithm prefers nodes with the smallest estimated distance to the target. Therefore, it always expands the node that seems to be closest to the target according to the function f(n).

Below, the algorithm is demonstrated using the classic example of finding a way on the map of Romanian cities.

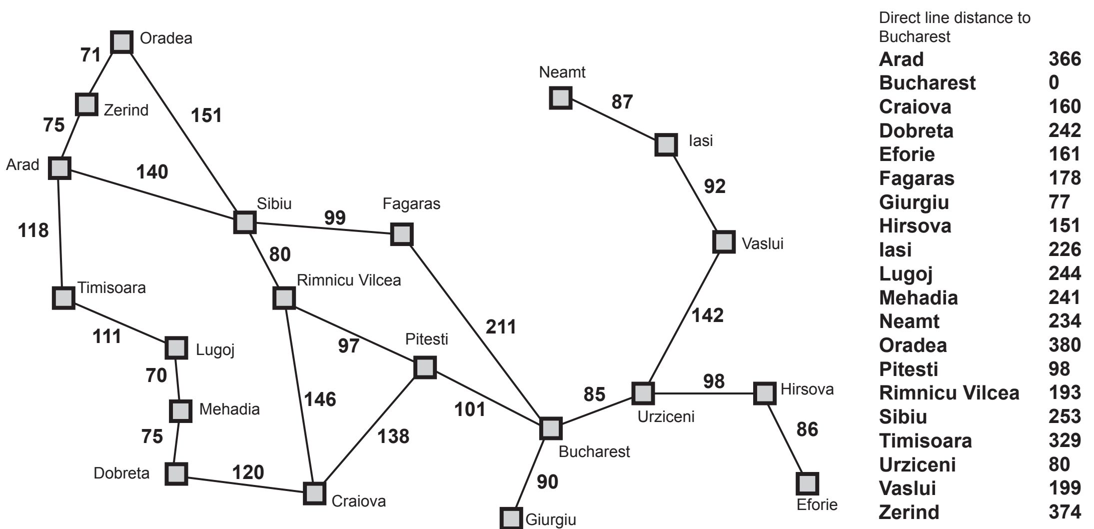


Fig. 2.8 A schematic map of Romanian cities

We have a map of the cities and roads that connect them, including the length of these roads.

In addition, we have a table that shows the aerial distance of each city from Bucharest. We want the algorithm to find a way from Arad (initial state) to the target in Bucharest (target state). The algorithm must, therefore, decide whether to go from Arad to Timisoara, Sibiu or Zerindu.

Note that the algorithm cannot see the whole map; it only disposes of the table of distances. In the next step, the algorithm selects a city that is closest to the target according to the aerial distance (heuristics). In this case, it will be Sibiu. Further, the algorithm continues in the same manner; from Sibiu, it directs to Fagaras, and from Fagaras, it directs to the target in Bucharest. The algorithm, therefore, always looks for a local minimum of the heuristic function and hopes to arrive at the global minimum.

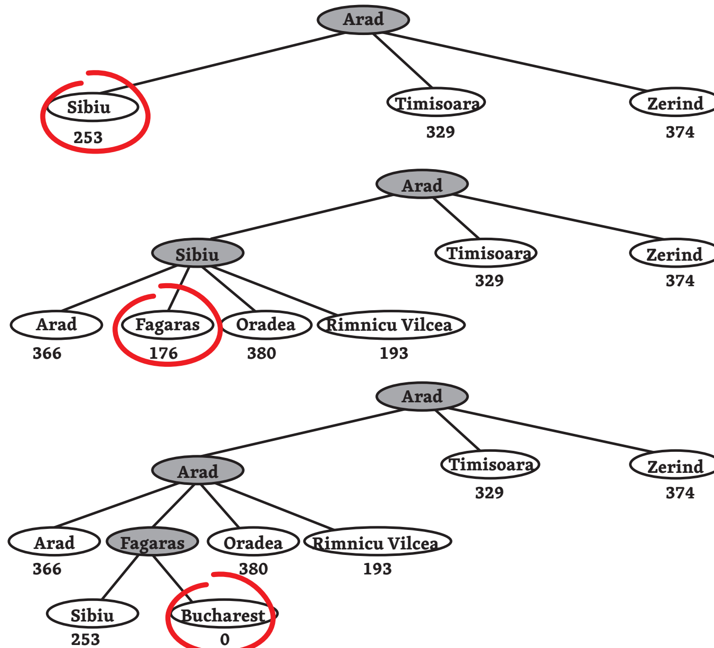


Fig. 2.9 Successive generation of the path – greedy algorithm

As we see, the algorithm was successful; using this heuristic, it directed to the target more efficiently than in the case of an uninformed search that would have gradually searched the entire map, just based on its topology.

However, the found way is not optimal; its length is 450 kilometres while the shortest path measures only 418 kilometres. The greedy algorithm deviated from the optimal solution in Sibiu where it "hungrily" chose Fagaras as the next node instead of the more optimal path through Rimnicu Vilcea. Greedy search is, therefore, always trying to snap the biggest possible piece of the remaining path, regardless of its overall length. Nevertheless, this strategy is relatively successful leading to victory, not only in the case of state space search.

Features of the algorithm

The algorithm is not complete. Also, it is not optimal. Time and space complexity corresponds to uninformed search; in the worst case, the heuristics does not play any role in the given task, i.e.

$$O(GS) = (b^m) \qquad (14)$$

Nevertheless, it is important to realize that the used heuristics can often significantly reduce the search time. The reduction of the time needed to find the target, however, cannot be guaranteed.

Advantages and disadvantages

The advantage of the algorithm is actually only one – it is an informed algorithm using a heuristic function that can significantly reduce the real time requirements for finding the solution. However, the algorithm is neither complete nor optimal, and there is a danger of getting stuck in a local minimum. Consider the task of finding a way from Neamt to Fagaras. Using the method of greedy search, we first expand the only possible node from Neamt to Lasi. After evaluating the heuristics for the two neighbouring cities, i.e. Vaslui and Neamt, we find that the air distance from Neamt to Fagaras is shorter than the air distance from Vaslui. The heuristics would, therefore, return the algorithm to the Neamt node and so over and over. Therefore, it is appropriate to introduce the detection of repeated states (for example) and thus prevent similar loops. This problem is solved by introducing the aforementioned OPEN and CLOSE sets where the nodes located in the CLOSE set are no longer repeatedly searched, which prevents the algorithm from looping. The algorithm also has a relatively large space complexity.

2.5.3.2 A algorithm*

Description

This algorithm again uses the principle of greedy search, but also includes the cost of the path to the n-th node into the heuristic function evaluation, i.e. $f(n) = h(n) + g(n)$. The function $g(n)$ is the cost of the path to node n, and $h(n)$ is an estimate of the cost of the path from the n-th node to the target node representing the problem solution. The function $f(n)$ thus represents an estimate of the cheapest path to the target leading through n.

A* algorithm requires the use of admissible heuristics $0 \leq h(n) \leq C^*$, where C^* is the actual cost of the path to the target. In case it is not, the algorithm is referred to only as the A algorithm. The heuristic function, therefore, performs optimistic estimate of the cost of paths to the target; no real path can have a lower price (be shorter) than the heuristics used. In our example of Romanian cities, this is satisfied because the air distance between two cities must always be shorter or the same as the length of the road connecting them.

In principle, the A* algorithm works exactly like GS; the only difference is that it performs priority ordering of states according to the values of the function $f(n)$.

Let's look at how the A* algorithm would behave in finding the path from Arad to Bucharest.

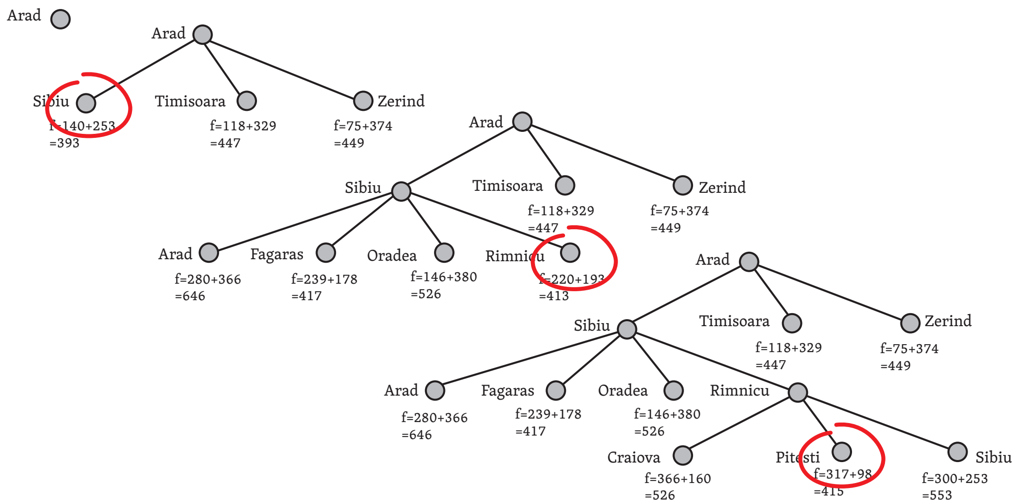


Fig. 2.10 Successive generation of the path – A algorithm*

As far as the Sibiu node, the algorithm selects the same way as GS. Here, it evaluates the function f(n) for all successors. Despite Fagaras being closer to the target in a beeline compared with Rimnicu (178 km compared with 193 km), the A* algorithm selects Rimnicu as the next expanded node. After offsetting the cost of the path from Arad to Fagaras (239 km) or Rimnicu (220 km), the algorithm finds that the way through Rimnicu (413 km) is generally preferred over the route via Fagaras (417 km). The value of the function f(n) is still only an estimated length of the path because, in addition to the exactly quantified cost (length) of the path already completed, it also contains an estimate of the length of the following path based only on the aerial distance between the cities. Nevertheless, admissible heuristics guarantees finding the optimal path.

Features of the algorithm

Since the algorithm expands nodes gradually according to the increasing values of the estimate of the cost of the path to the target, f(n), it must ultimately reach the target for the final number of nodes with $f(n) < C^*$. A* algorithm is, therefore, complete provided that the condition of the final number of nodes with $f(n) < C^*$ is met. The condition of the finite number of such nodes may not be met if the branching factor b is endless or if the graph contains an infinite branch, though with a final cost. It may be the case when h(n) for the nodes on this branch is zero.

Therefore, the condition of completeness in the A* algorithm is the application of admissible heuristics $h(n) > 0$ (i.e. each transition costs something) and the finite branching factor b in the graph.

For evaluating nodes of the functions $f(n) < C^*$, where C^* is the actual cost of the path to the target, the A* algorithm expands all nodes with $f(n) < C^*$; it checks some nodes with $f(n) = C^*$ for the presence of the target and does not check any nodes with $f(n) > C^*$.

Admissible heuristics ensures that the A* algorithm is optimal.

Suppose that we are in a particular node from which two edges lead off. One edge leads directly to the node with suboptimal target g_{sub}; the second edge leads to a general node n whose subtree contains the optimal target g_{opt} with the actual cost of path C^*.

a) Node gsub is the target; therefore $h(g_{sub}) = 0 \Rightarrow f(g_{sub}) = g(g_{sub})$.

b) Admissibility of the heuristics implicates that $f(n) \leq C^*$

c) We know that the A* algorithm makes decisions according to the increasing value of the function f(n). If the algorithm selects node g_{sub}, than $f(g_{sub}) \leq f(n)$

d) Points b) and c) implicate that $f(g_{sub}) \leq C^*$. When substituting from point a), we obtain that $g(g_{sub}) \leq C^*$

Point d), however, is in clear contradiction to the fact that the cost of the path $g(g_{sub})$ to the suboptimal target is less than the actual cost of the path to target C^*.

Therefore, the A* algorithm cannot select a suboptimal target g_{sub} with the function value $f(g_{sub}) = g(g_{sub})$ from any node because the cost estimate f(n) for node n on the path to the optimal target g_{opt} with path cost C^* is always lower, with regard to admissibility (i.e. always optimistic estimate) of the heuristics.

Time complexity of A* algorithm is exponential; in the worst case (if h(n) is constant "$\forall$n), the complexity corresponds to BFS but the algorithm is always better than BFS because it does not need to expand nodes with $f(n) = C^*$.

$$O(A^*) = (b^m) \qquad (15)$$

Heuristics, however, is often able to significantly reduce the search time.

In the basic variant, the space complexity is also exponential and is equal to

$$O(A^*) = (b^m) \qquad (16)$$

However, it is also possible to use modifications for its reduction; for example, we can use the IDS algorithm according to the maximum cost I – for each further iteration of IDS, we select the initial node with the lowest value of the function f(n) that is simultaneously higher than the current l. This method is referred to as IDA*. Another option is to cut non-promising states and paths to these states. In this way, we save memory capacity at the cost of suboptimality of such a solution.

Advantages and disadvantages

It is a successful algorithm that usually and significantly reduces the time complexity of the search, depending on the quality of heuristics, of course. In addition, it is a complete and optimal algorithm that does not extend the already long paths.

Like in the case of GS, it is necessary to treat the possibility of algorithm loops using the CLOSE set. The relatively large memory complexity can be reduced at the cost of suboptimality of the algorithm.

2.5.4 Local search methods

A separate area of methods for searching the state space is represented by the so-called local search methods. Let's briefly mention at least two of them, the hill climbing (a gradient search method) and simulated annealing. A common feature is that these methods evaluate only their immediate surroundings; when searching, they set off in one direction that seems to be locally optimal at the given moment, based on evaluating the function f(n). Therefore, they behave like GS. Local methods, however, completely forget the previous nodes and thus lack the ability to return.

Compared to a simple gradient search, the simulated annealing algorithm differs in introducing the concept of the temperature T. This temperature is high at the beginning of the search (it allows wider state space search at the beginning of the algorithm run) and gradually falls to zero during its course (the algorithm converges to a solution). When selecting a successor, the algorithm does not always choose only the best successor in each node as in the case of gradient search; it selects transition to another node with certain probability depending on the temperature T. The probability of selecting a suboptimal successor decreases with decreasing temperature; the algorithm gradually converges to a steady state. The algorithm is more likely to reach the global minimum (the target). By introducing the temperature, we limit the possibility of getting stuck in a local minimum because the algorithm is able to get out of it with certain probability and search also those parts of the state space where the simple gradient method would not enter. In case of failure, this algorithm is often started repeatedly from different starting points of the state space.

The advantages of local algorithms are especially their ease of implementation and minimal memory requirements. The disadvantage is mainly their suboptimality and incompleteness; they can easily get stuck in a local minimum. Nevertheless, they are successfully used for many problems.

2.6 LITERATURE

[1] Kubalík, J.: Základy umělé intelligence (Fundamentals of Artificial Intelligence), course website https://cw.felk.cvut.cz/wiki/courses/y33zui/program
[2] Barták, R.: Umělá inteligence I (Artificial Intelligence I), course website http://ktiml.mff.cuni.cz/~bartak
[3] Pavliš, M.: Algoritmy pro hledávání cesty (Algorithms for the Path Search), Thesis, 2007, http://autnt.fme.vutbr.cz/szz/2007/BP_Pavlis.pdf
[4] Berka, P.: Úvod do umělé intelligence (Introduction to Artificial Intelligence), course website http://sorry.vse.cz/~berka/4IZ229/
[5] http://cs.wikipedia.org/wiki/Patn%C3%A1ctka

3 EXPERT SYSTEMS

3.1 BASIC INFORMATION

The following text forms part of the learning texts for the course "Artificial Intelligence" and is intended primarily for students of Mathematical Biology. It provides basic information on the field of artificial intelligence that deals with expert systems (ES). This learning text briefly describes the principles of expert systems and introduces the methods that can address the inclusion of uncertainty in ES decision-making mechanisms.

3.2 LEARNING OUTCOMES

Mastering the learning text will enable students to:

- *Understand the basic concepts such as*
- *Expert system*
- *Rule-based and non-rule-based expert systems*
- *Forward and backward chaining in ES*
- *Acquire inference in ES using propositional logic and modus ponens*
- *Become familiar with and understand the methods of dealing with uncertainty in ES*
- *Bayesian conditional probability*
- *Factors of certainty*
- *Fuzzy logic*

3.3 EXPERT SYSTEM (ES)

Expert systems belong to the methods and models of artificial intelligence. They represent a computer model designed to replace decision-making by human experts. In accordance with the definition of artificial intelligence, it is then a type of intelligent machines emulating human decision making. Expert systems are created as specialized models, i.e. models focusing on given areas (medicine, process management, business ...) of human activity. The aim of such expert systems is to optimally emulate decision-making of experts, i.e. professionals specialized in a certain area, that are able to solve problems in this area better than other people, based on their knowledge and experience.

Compared to a human expert, the successfully created expert system has several advantages: economical to operate, readily available, time unlimited decision-making stability, and precise description of the decision-making process. Understandably, creating such a system can be problematic in cases when it is not easy to convert expert human decision-making to a system operating according to set rules as, more often than not, the human expert is unable to define the procedure of arriving at the correct decision – intuition based conclusion.

3.4 COMPONENTS OF EXPERT SYSTEMS

The basic idea of an expert system, regardless of its specific implementation, is that it contains a knowledge base and an inner inferential mechanism that performs the ES reasoning. The ES, therefore, receives variable inputs, makes its own decisions and responds with its conclusions – outputs.

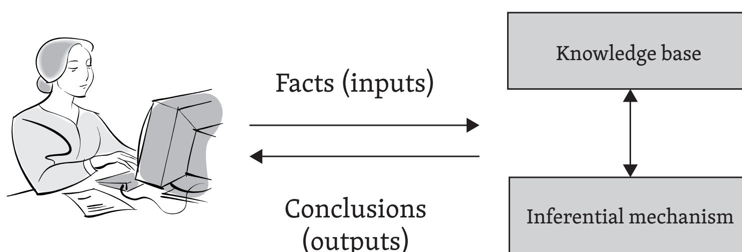


Fig. 3.1 Expert system (according to [1])

However, other specialized modules can be defined in addition to knowledge base (which can be viewed as a database of decision rules) and inferential mechanism:

- *Working memory to store actual facts*
- *Knowledge acquisition module ensuring updates to the knowledge base*
- *Explanatory module that allows the user to view the process through which the ES has drawn the respective conclusions*
- *User interface for interacting with ES user*

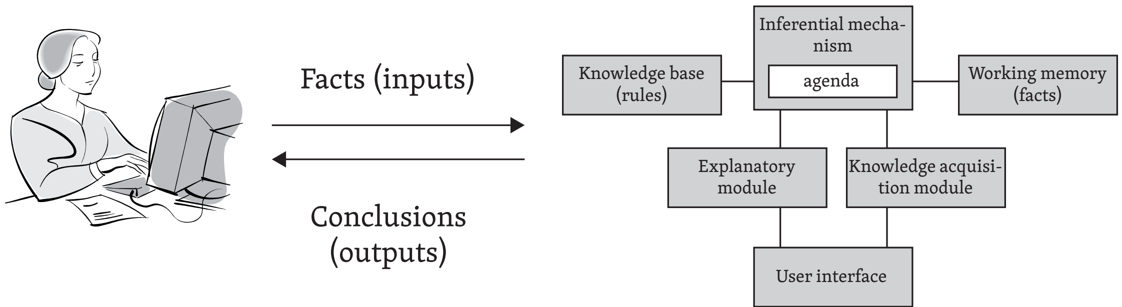


Fig. 3.2 Block diagram of the expert system (according to [1])

3.5 RULE-BASED EXPERT SYSTEMS

3.5.1 Knowledge representation

In rule-based ES, knowledge is represented by a database of rules. None-rule based ES are based on meeting certain assumptions and generating conclusions that result from such presumptions. Therefore, these systems apply rules such as if -> then, assumption -> consequence (action, conclusion, consequence). These expert systems are also called production ES. Formally, the decision rule can be written as "IF the assumption(s) applies (apply), THEN the following conclusion (s) applies (apply)", or "IF p THEN q", or also as p ==> q, where p can be a composed expression linked by logical operators AND, OR, and q can be an expression connected by logical operators AND. The expression p is actually an observation leading to the hypothesis q, assuming that it occurred.

As examples, we can present simple rules like "IF (temperature_increased_above_38C OR headache) THEN (use_500mg_of Paracetamol AND go_to_bed)".

The inferential mechanism in production ES is based on repeated application of the modus ponens rule.

$$\frac{p, p \Rightarrow q}{q} \qquad (1)$$

The modus ponens rule expresses the fact that if the assumption p and $p \Rightarrow q$ rule apply simultaneously, then the hypothesis q applies as well.

p	q	p->q	(p->q)∧p	((p->q)∧p)->q
T	T	T	T	T
T	F	F	F	T
F	T	T	F	T
F	F	T	F	T

Tab. 3.1 Modus ponens (according to [1])

From Table 1, it is evident that modus ponens is a tautology, a statement valid for all combinations of p and q.

Let's assume, for example, the existence of a rule given by the following statement *(p ⇒ q)* "If the sun shines, I will go to the swimming pool". What does this statement say in case that p = "The sun shines"? In such case, it is clear that I will go to the swimming pool and, therefore, the statement q = "I will go to the swimming pool" is true as well. Validity of statement q has been, thus, unambiguously inferred – I have to go to the swimming pool.

However, what would be the situation if the sun does not shine, i.e. p = "The sun does not shine". In this case, we can say nothing about statement q; if the sun does not shine, I promise nothing; I can go to the swimming pool even if the sun does not shine; or I can stay home. We do not have any clearly defined rule for this situation. Similarly, when we assume the same rule *(p ⇒ q)*. Someone might see us at the swimming pool and is thus able to define compliance with the statement q = "I will go to the swimming pool." Based on this observed fact, it is not possible to judge whether the sun really shone although this statement resembles the modus ponens. The validity of the statement q does not enable us to infer any valid statement for p.

Through repeated inferences, the expert system tries to create an inferential chain that directs input facts toward ES conclusions. It, therefore, relates to finding such ordered sequence of ES rules the successive application of which creates a path from initial facts to conclusions produced by the ES.

Basically, the ES can create the inferential chain in two ways – based on the principle of forward chaining or backward chaining.

Let's take these rules:

veiled crucian(x) ->crucian(x)

crucian(x) -> fish(x)

fish(x) -> lives in water(x)

lives in water(x) -> put into an aquarium(x)

In the box, we were given a gift, a new animal Nela. We have no instructions for this animal, and we do not know what to do with it. We only know only that it is a veiled crucian. We're trying to find out what to do with it. Using the very simplified case of our rule base, we try to pair the available facts, i.e. the veiled crucian (Nela) with known rules, and come to a conclusion. Thus, we can make the inference according to the following schedule and come to the conclusion that the best solution will be to put Nela into an aquarium.

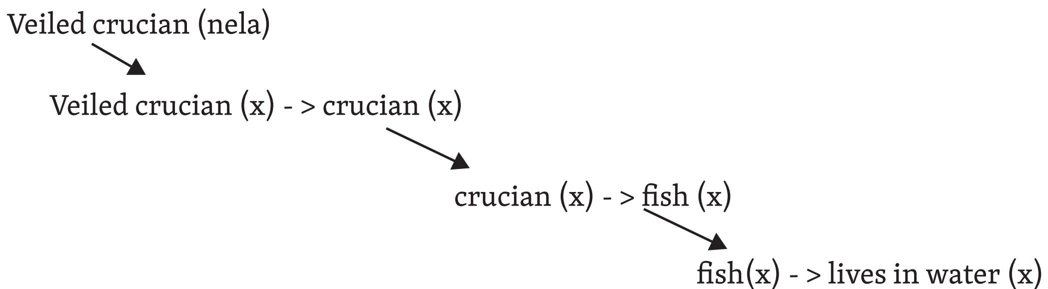


Fig. 3.3 Simple inference in the ES

The typical steps of the forward chaining are:

- *Comparison – we pair supplied facts and find out which rules are workable*
- *Conflict resolution – in the set of workable rules, we have to select and perform some rule (randomly, according to priorities ...)*
- *Execution – execution can result in the emergence of a new fact, the removal of fact, the introduction of a new rule...*

This procedure corresponds to forward chaining (data-driven) where we proceed from the known facts and infer possible conclusions. This method is useful for tasks that begin in the present and, on the basis of its knowledge, infer what will happen or what is to happen in the future.

Note the analogy with searching through a defined space where we have to choose a strategy of gradual implementation of various workable rules. We put new workable rules at the end or beginning of the queue and determine their priority. Forward chaining is usually implemented as a defined space searching in width. A typical area of application is planning and management.

Conversely, backward chaining (target-driven) verifies all available conclusions and attempts to prove their validity by building a reverse inferential chain. In this respect, it is based on the outputs that were generated in the past by previous processes and attempts to detect and describe these processes. Backward chaining is applicable only to tasks with relatively limited number of possible conclusions. It is usually implemented as space searching in depth. Typical examples are diagnostic tasks.

3.5.3 Advantages and disadvantages of rule-based ES

In particular, the basic advantages of rule-based ES include:

- *Modularity - Unequivocal expression of the given knowledge by a rule and the ability to easily expand this set of rules. The knowledge acquisition module is simple.*
- *Simple interpretation - Thanks to the unambiguously given algorithm and the sequence of rules performed, it is easy to create the explanatory module and thus submit a comprehensible description of the decision-making process represented by the inferential chain.*
- *Similarity with human reasoning - If I see and know something, I behave accordingly. Rule-based ES proceed analogously.*

Disadvantages of rule-based ES consist in their applicability only for simpler problems that are easily transferable to an acceptable number of rules.

- *Not very efficient algorithms for comparing rules and space searching*
- *Risk of cycles*
- *The inability to capture complex problems with a reasonable amount of clear rules*

3.6 NON-RULE-BASED EXPERT SYSTEMS

In non-rule-based ES, knowledge is not represented by a set of rules; it is contained in a different form. Most often, it is knowledge representation using semantic networks or frames and objects and the relationships between them.

3.6.1 Semantic networks

The semantic network is represented by a bi-directionally oriented graph that represents ES knowledge. The nodes represent individual model objects and the edges represent the relations between them. All knowledge is thus spread out in a single network layer – shallow knowledge structure. The most commonly used relations between the objects are relations such as "is" or "is part of" or "has". Semantic networks support transitivity and inheritance. However, it must be noted that these relations are not standardized as well as semantic networks; each EC may, therefore, come up with its own definitions or modifications.

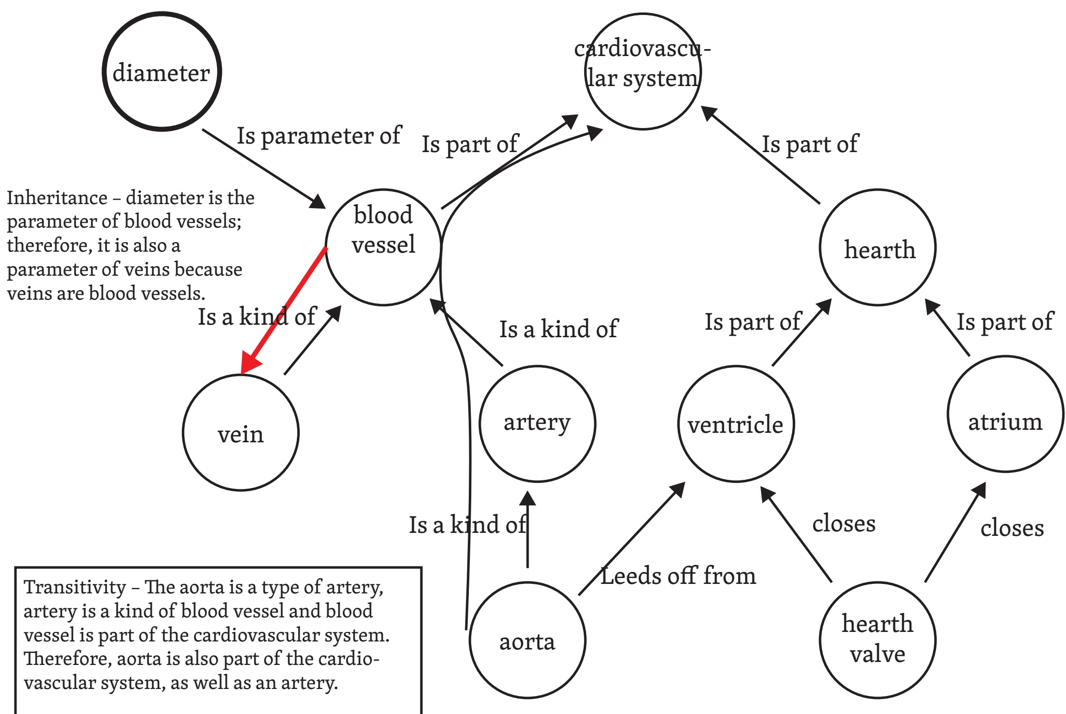


Fig. 3.4 Semantic network of the cardiovascular system (modified according to [1])

The main advantage of semantic networks is their clear expression and easy search. On the contrary, the disadvantage is in the non-standardized relationships between objects, which can lead to different ways of interpretation.

3.6.2 Frames and objects

Knowledge representation using frames and object finds its analogy in the structured data variants of procedural languages or in object-oriented programming.

Frames are based on the concept that people, when dealing with new situations, use similarities with the already known schemes, i.e. they look for stereotypical solutions to a new situations on the basis of analogy, only with modified input conditions. Frames can be represented analogously to the data type of "recording procedural languages" where individual fields of the record are called frame slots, and the values of these fields are called contents. Contents may also be other frames or special procedures that are called, for example, when

changing the value of a frame slot, similarly to the treatment of events, e.g. in Javascript. The frames are mutually arranged in schemes.

- *Frame* ***"family vehicle"***
  - ***Name:*** vehicles
  - ***Type:*** range (passenger cars, vans, motorcycles)
  - ***Owner:*** if needed (procedure find_the owner_)
  - ***Location:*** range (garage, shed, parking-site)
- *Frame* ***"car"***
  - ***Name:*** car
  - ***Specialization:*** family vehicles
  - ***Type:*** passenger cars
  - ***Location:*** garage
- *Frame* ***"Petr's car"***
  - ***Name:*** Petr's_car
  - ***Specialization:*** car
  - ***Owner:*** Petr Nový

Frames can be both general and specific; they support the definition of relationships between frames. Compared to semantic networks, it is possible to create more complex knowledge structures by means of diagrams. Adaptation to a new task may be problematic if the scheme or definition of individual frames does not match the input task. Frames are, therefore, very close to objects that constitute the next step in knowledge representation of ES.

The definition of knowledge using objects is close to object-oriented programming. The basic concepts are "class" and "object". The class is a template, a general definition of the structure and behaviour of future objects. In addition, classes can be mutually arranged hierarchically; they thus represent the relations between future objects. In a class, the structure of future objects is defined by variables contained in the object declaration, and object behaviour is given by the declared object methods. Therefore, the aim is to map the classes of real-world objects and their relationships to their simplified computer models.

The object is a unique instance of the given particular class; it exists separately in the memory and bears the specific values of its attributes. In the memory, there can be more objects of the same class, and each class has its own space.

The basic properties of objects are:

- ***Abstraction*** *from the details of the world and the internal implementation of the object. The object captures only those features of reality that are essential for solving problems.*
- ***Encapsulation*** *of internal attributes and methods. The object represents a black box. Outwardly, it only provides a clearly defined interface irrespective of the internal implementation. This ensures stable behaviour of the object, which is dependent only on the input variables.*
- ***Inheritance*** *– objects (descendants) inherit their properties (attributes, methods) from their parent or even more parents at the same time. They can complement them, or even transform, e.g. change the internal implementation of the given method. The hierarchy of objects is determined by the arrangement of their patterns – classes.*
- ***Polymorphism*** *– for various descendants of the same class, internal implementation and the execution of methods can be different although the objects of different classes look equal in relation to their ancestors. Any descendant knows, at least, the same as its parent.*

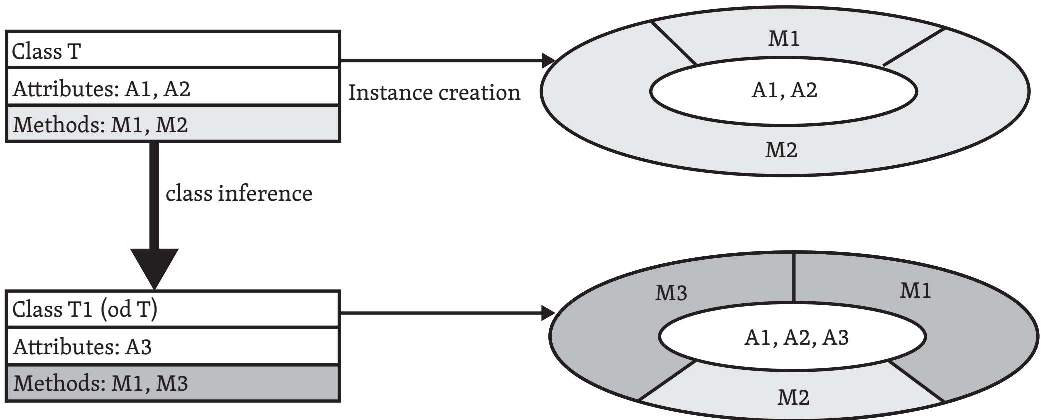


Fig. 3.5 Class and its instance (according to [2])

3.7 UNCERTAINTY IN ES

We often find ourselves in a situation where we want to solve a task despite the fact that we do not have absolutely correct information. Information is insufficient, inaccurate, biased, ambiguous, or is burdened with error in principle. Even decision rules in ES knowledge base may not be entirely clear. Despite these handicaps, we want to conclude using ES. The mentioned facts are the reason for introducing uncertainty into ES decision-making. Most often, the uncertainty is modelled by a numeric parameter that represents the degree of pertinence to a certain object or probability of performing a particular transition, a rule.

3.7.1 Conditional probability

The classic method of processing uncertainty in the complete system of events, that influence each other, is the Bayesian probability. The rule (expressed by relation (1) and used in production ES) says that when observing an event p (i.e. the statement is true, p happened), q is true as well. Conditional probability allows us to define the probability P (q|p) which says that the probability of validity of q is not absolute assuming the validity of p. Consequently, observing the event p only supports the hypothesis q with probability P (q|p) in the interval <0,1>.

Suppose a complete system of events where events p1, p2,..., pi may occur.

It is true that

$$0 <= P(p_i) <= 1 \qquad (2)$$

It is also true that

$$\sum_i P(p_i) = 1 \qquad (3)$$

The probability that two independent events will happen, i.e. p_1 or p_2, is equal to the sum of probabilities of both events.

Providing that a particular event p is observed, the conditional probability of i-th hypothesis q_i can be expressed by Bayes relation for conditional probability:

$$P(q_i \mid p) = \frac{P(p \mid q_i) * P(q_i)}{(P(p)} \qquad (4)$$

3.7.2 Dempster-Shafer theory

Another method of dealing with uncertainty in ES is the Dempster-Shafer theory. It works with the concept of environment of mutually disjoint events, hypotheses q_i.

The environment is defined as

$$\Theta = \{q_1,q_2,\ldots,_i\}. \qquad (5)$$

All combinations of these hypotheses, that can occur, then compose the subsets of the environment; e.g. for the three hypotheses q_1, q_2, q_3

$$\Omega(\Theta) = \{\emptyset, \{q_1\},\{q_2\},\{q_1, q_2\},\{q_1, q_3\},\{q_2, q_3\},\{q_1, q_2, q_3\}\} \qquad (6)$$

Next, we define the notion of quantity of case m, which supports the plausibility of the occurrence of a certain combination of hypotheses, i.e. a certain subset X. It is true that

$$m: \Omega(\Theta) \in <0, 1> \qquad (7)$$

$$m(\emptyset) = 0 \qquad (8)$$

$$\sum_{(X \in \Omega(\Theta)} m(X)=1 \qquad (9)$$

Gradual observation, i.e. event occurrence, that supports the given environment subset X, leads to better specification of the amount of case m(X) of the given environment subset in the interval <0,1>. The complement of plausibility of the given subset 1-m(X) is called implausibility, and is assigned to environment plausibility. Therefore, it is not assigned to any particular subset.

When registering more events supporting the plausibility of subsets, we proceed according to the Dempster's combination rule that defines the combined amount of cases for the given subset X_i.

$$m_3(X_3) = \sum_{X_1 \cap X_2} m_1(X_1)\, m_2(X_2) \qquad (10)$$

Therefore, it is a sum of products of the amounts of cases of subsets over all conjunctions X_i.

The application of Dempster's rule can be demonstrated using the following example. Assume again that we have an environment containing three hypotheses q_1, q_2, and q_3. Then, according to (6), all subsets are:

$\Omega(\Theta) = \{\emptyset, \{q_1\},\{q_2\},\{q_1, q_2\},\{q_1, q_3\},\{q_2, q_3\},\{q_1, q_2, q_3\}\}$

Let $X_1 = \{q_1\}$ and $X_2 = \{q_1, q_2\}$.

Furthermore, we recorded two events. The first event supports the plausibility of subset X_1, e.g. $m_1(X_1) = 0.6$; observation of the second event supports the plausibility of subset X_2, $m_2(X_2) = 0.9$.

The combined amount of the case for the two subsets is, therefore, composed of two observations, i.e.

$m(X_1) = m_1(X_1)\, m_2(X_2) + m_1(X_1)\, m_2(\Theta) = 0.6*0.9 + 0.6*0.1 = 0.6$
$m(X_2) = m_2(X_2)\, m1(\Theta) = 0.9*0.4 = 0.36$
$m(\{\Theta\}) = m1(\{\Theta\})\, m2(\{\Theta\}) = 0.4 * 0.1 = 0.04$

The observed events helped increase our knowledge about the environment and refined the estimates of subsets X1 and X2, whose plausibility they support. However, the ignorance is still not zero.

In the Dempster-Shafer theory, the concept of the amount of cases is an analogy to the conditional probability; however, this theory is more general. The conditional probability assigns a probability to each hypothesis, even if we do not have any prior information. In this case, we consider all hypotheses as equally probable and the following applies to each of them

$$\sum_i P(q_i) = 1 \qquad (11)$$

For dichotomy, for the hypothesis probability qi we then get

$$P(q_i) = 1 - P(\sim q_i) \qquad (12)$$

The Dempster-Shafer theory does not require these conditions, often restrictive in practice, and allows us to work even with ignorance.

3.7.3 Fuzzy sets

In the real world, we often try to express the pertinence of a given object to some class. For example, consider a human. We often say that someone is very small, big, tall or very obese. Surely, he/she can be small and simultaneously rather fat, i.e. the person can belong to several classes with different classes of pertinence. We do not define exactly how many centimetres are indicated by the expression "small" or how many kilograms are indicated by the term "fat".

Fuzzy sets solve the assignment of object x to a given class using the so-called degree (function) of pertinence A(x) for a certain set A (e.g. the set of fat people), usually in the interval <0,1>.

In the universe U = {x}, the fuzzy set A is therefore equal to a set of pairs

$$A = \{(A(x_1),x_1), (A(x_2),x_2), \ldots, (A(x_n),x_n)\} \qquad (13)$$

There are operations defined over fuzzy sets A and B, allowing work with these sets. The basic operations include:

Complement	$A'(x) = 1 - A(x)$
Union	$(A \cup B)(x) = \max(A(x),B(x)) = A(x) \vee B(x)$
Bold union	$(A \underline{\cup} B)(x) = 1 \wedge (A(x) + B(x)) = A(x) \oplus B(x)$
Intersection	$(A \cap B)(x) = \min(A(x),B(x)) = A(x) \wedge B(x)$
Bold intersection	$(A \underline{\cap} B)(x) = 0 \vee \max(A(x) + B(x) - 1) = A(x) \otimes B(x)$
Concentration	$con(A)(x) = A^2(x)$
Dilatation	$dil(A)(x) = 2A(x) - (A(x))^2$

Tab. 2 Operations over fuzzy sets (according to [1]).

The operations are used to create new fuzzy sets. For example, it is possible to obtain a new fuzzy set of small fat people by calculating the intersection of these sets. Using concentration and dilation operators on a fuzzy set of tall people can be seen as creating a set of really tall people (con) or a set of people who are rather tall (dil).

Therefore, through these operations and fuzzy sets, the fuzzy logic allows us to declare vague rules.

Again, repeat the rule valid in the production expert systems (1), i.e.

$$\frac{p, p \Rightarrow q}{q}$$

Assumptions p, as well as conclusions q, can be modelled as fuzzy sets. Suppose an event that happened and which we observed; suppose that we entered a specific value describing this event to ES as a new fact. This fact can belong to several assumptions p, always with a varying degree of pertinence to that fact. Therefore, the validity of various conclusions q is also vague, defined by a varying degree of pertinence to the given hypothesis. For example, if we have temperature increase to 38°C, ES can advise us to use Paralen with a high degree of pertinence, but it can also recommend, perhaps with a lower degree of pertinence, travelling to the North Pole.

In the final stage of the decision-making process, the expert system using fuzzy logic will reach a point where it has to perform defuzzification, i.e. conversion of vague conclusions to a specific action. It may be the choice of the most likely hypothesis (e.g. according to the maximum degree of pertinence). It may also be an inference of the specific value from vague conclusions; for example, after considering all of the different facts, the aircraft autopilot reaches conclusions that it should rather (specifically given by the pertinence function) decelerate, preferably let the acceleration unchanged, and simultaneously also strongly accelerate. However, the result must be a single value, e.g. a change in acceleration by a certain percentage. In this case, for example, it is possible to proceed by calculating the centre of gravity area under the pertinence functions of fulfilled conclusions. In this case, the result of defuzzification would probably be partial acceleration, for example by 10%.

3.8 LITERATURE

[1] Provazník, I., Kozumplík, J.: 1999: Expert Systems, University of Technology in Brno, 1999, 101 p., ISBN 80-214-1486-3.

[2] Dvořák, J.: Expertní systémy (Expert Systems). Learning text, University of Technology in Brno, Faculty of Mechanical Engineering.
URL:http://www.uai.fme.vutbr.cz/~jdvorak/Opory/ExpertniSystemy.pdf.

[3] Berka, P. et al.: Expertní systémy (Expert Systems). Lecture notes. Prague, School of Economics, 1998.

[4] Zvárová, J.: Základy statistiky pro biomedicínské obory I. (Introduction to Statistics for Biomedical Fields I.). Prague: Karolinum, 1998. 218 p. ISBN 80-7184-786-0.

4. NEURAL NETWORKS – THE SINGLE NEURON

4.1 BASIC INFORMATION

The following text forms part of the learning texts of the course on "Artificial Intelligence" and is intended primarily for students of Mathematical Biology study programme. This chapter can be considered as an introduction to a block devoted to artificial neural networks. It introduces the basic concepts, defines a mathematical model of the neuron, and explains its classification options. Understanding the mathematical model of the single neuron is a necessary starting point for understanding the concept of complex structures using interconnected sets of single neurons - artificial neural networks.

4.2 LEARNING OUTCOMES

Mastering the learning text will enable students to:

- *Learn and understand the analogy between artificial neural networks (ANN) and their biological motivation, understand the concepts of ANN topology – neuron, layer, connection*
- *Parametrically define a mathematical model of the single neuron and define the meaning of individual parameters and the types of neuronal transfer functions*
- *Understand the active dynamics of the neuron and its geometric interpretation, and formalize it mathematically*
- *Learn the basic concepts and algorithms of adaptive dynamics of the single neuron*
  - Hebb rule
  - Delta rule
  - Learning according to Widrow
  - Training and testing sets
- *Apply the AND, OR, NOT and XOR mathematical logic functions in relationship with the classification capabilities of individual neurons*
- *Implement a model proposal of the AND, OR, NOT logic functions by setting the neuron parameters*
- *Understand the classification ability of the single neuron with two inputs and a binary output, and its geometrical interpretation in the plane*

4.3 INTRODUCTION TO NEURAL NETWORKS

4.3.1 Biological analogy

In the past, the concept of artificial neural networks was inspired by, and then created based on, the biological analogy with the nervous system. In its simplest form, the nervous system can be divided into the central nervous system (CNS), which is represented by the brain and the spinal cord, and the peripheral nervous system comprising mainly of peripheral nerves. The main function of the CNS is to control the body, process the signals (impulses) that enter the CNS from sensory receptors through centripetal pathways, and issue directives on the basis of evaluating these signals, which are then sent as efferent signals through the centrifugal neural pathways to effectors, such as the muscles. In this active dynamics, we can observe the existence of input information, its processing by the respective units (processors) of the CNS and the subsequent generation of an output from the CNS, which is then transformed into action at the effector, for example, hand movement when throwing the ball at a target. What happens if the target is missed? We can re-evaluate this situation at the receptors (e.g. visually) and try to correct and repeat the throw with less intensity in case the previous attempt was long. Humans can find a deviation from the desired ideal state at the output and train our nervous system based on the feedback, so that it is able to perform the given task as best as it can. Humans gain the ability to learn and remember the acquired information.

The entire human nervous system is represented by a vast number of interrelated biologic functional units, neurons. For example, the brain contains up to 1011 neurons, whereas each of these neurons can have up to 5000 connexions with other neurons. These connexions, synapses, can be of excitatory or inhibitory nature; they may thus increase or suppress the response of neurons to incoming stimuli, impulses.

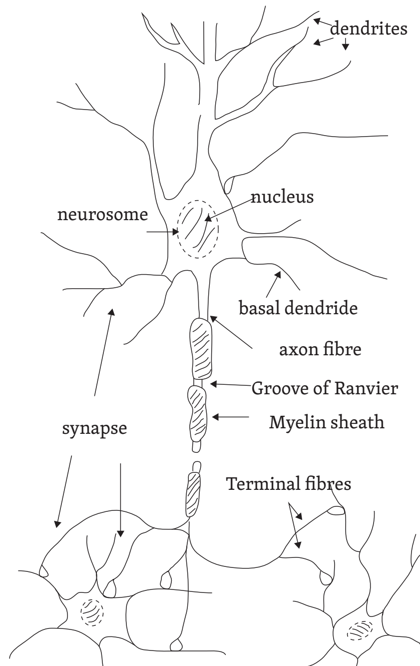


Fig. 4.1 Biological neuron (according to [3])

If we adhere to the simplistic description of the biological neuron, which should lead us to its mathematical model, we can conclude that each biological neuron consists of a body and two types of projections, the dendrites and the axon. In terms of the spread of excitation, dendrites represent inputs to the neuron body, through which signals from neighbouring neurons are received. These are flexible connexions capable of branching, extension and multiplication. It is assumed that the plasticity of dendrites represents the base for long-term memory. Axon is the output of the neuron, which spreads excitation from the neuron body and which is connected to the dendrites of other neurons through synapses.

Initially, the artificial neural networks were inspired by a brief biological analogy; nevertheless, after the establishment of core technical concepts, their development was conducted completely independently, without any direct link to the original motivation. Compared to known biological reality, the mathematical representation and models of artificial neural networks are, therefore, only very simplified models, often quite distant.

4.3.2 History of ANN

Briefly in points, let's indicate the baseline of the development of artificial neural networks. In 1943, Warren McCulloch and Walter Pitts created a simple mathematical model of the neuron representing the theoretical model of part of the nervous system - the biological neuron. They did not expect that their proposal would be applied in practice; however, the model became a subject of investigation for other researchers. In their model, each neuron had several inputs and one output. Inputs were divided into excitatory and inhibitory, and the model accepted only m binary inputs and one output. Therefore, it was quantifying the function from {0.1}m -> {0.1}. The principle was as follows: when excitation at input outweighs inhibition, the neuron output is excited as well. Otherwise, the neuron is not excited and the neuron output features a zero (0) value. Input synapses were not weighted in any way and serve only for transmitting the value.

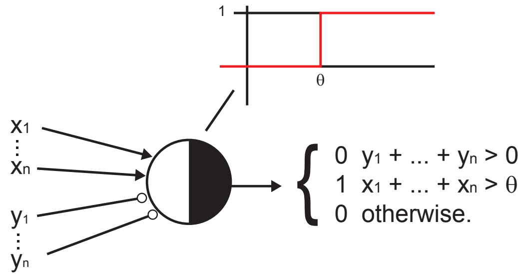


Fig. 4.2 Model of the neuron by McCulloch-Pittse (according to [3])

In 1949, again on the basis of biological analogy and the study of conditioned reflexes, Donald Hebb defined a rule that allowed the learning of neurons by changing the weights of their inputs. His assumption was that if a neuron is excited correctly, it will strengthen the connexions that led to the excitation. Conversely, if the neuron is excited incorrectly, the connexions would be weakened.

In 1957, Frank Rosenblatt generalized the McCulloch-Pittse's neuron and introduced an artificial neural network, the perceptron. This network was used for recognizing signs projected on canvas and was again inspired by biology – i.e. the discovery of light-sensitive cells in the retina of the eye and the way of signal transmission to the CNS. The architecture of Rosenblatt's perceptron is shown in Figure 4.3.

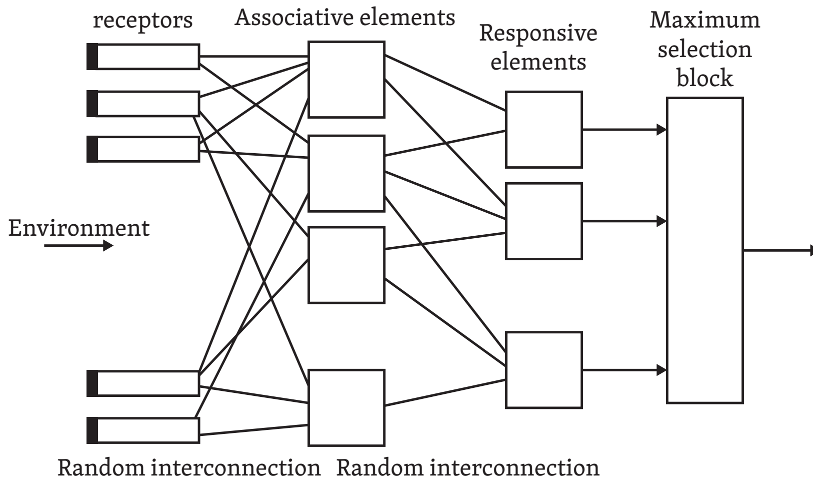


Fig. 4.3 Rosenblatt's perceptron (according to [3])

In 1960, Bernard Widrow presented another generalization of the basic neuron, the adaptive linear element (ADALINE). He performed generalization of real input values and introduced a new learning rule for the neuron.

In 1969, development in the field of artificial neural networks ran into difficulties because one significant person of artificial intelligence, Marvin Minsky, somewhat purposefully discredited the concept of artificial neural networks. To do so, he used the so-called XOR problem when the single-layer neural network is not able to quantify the logical XOR function. According to him, the solution was to use multiple layers of the neural network but there was no formalized algorithm that would allow learning of such networks. He erroneously assumed that it would not be possible to find the learning algorithm due to the complicated network structure. His

conclusions resulted in redirection of the grant funds to other areas of artificial intelligence and the concept of artificial neural networks began to stagnate. However, in 1986, a group of researchers derived an algorithm that enabled the learning of multilayer networks on the principle of retrograde propagation of errors (error back-propagation). This finding overcame the XOR problem and led to intense interest in the area of neural networks, which continues to this day.

4.3.3 The concept of artificial neural network

Artificial neural network is composed of mathematical neurons, primitive units, where each unit processes weighted input signals and generates output.

The neural network is a topological arrangement of single neurons in the structure communicating via oriented and rated connexions. Inter alia, each artificial neural network is thus characterized by the type of neurons, their topological arrangement and adaptation strategy in network training, learning.

The basic idea of the concept of artificial neural networks will be shown on the principle of the most commonly used feed-forward neural network. The network is called "feed-forward" because the signal propagates through the network from input unidirectionally toward the network outputs. The arrangement of neurons for the forward network is shown in Figure 4.4.

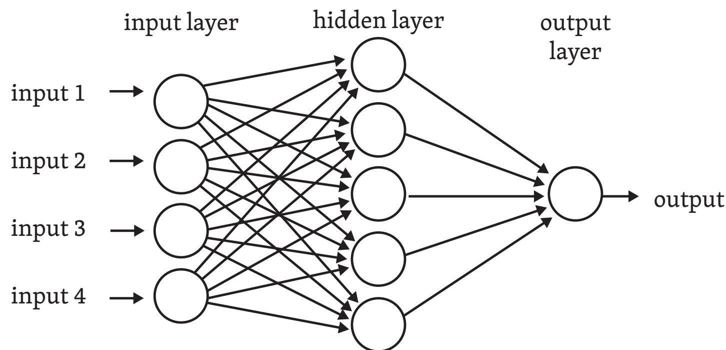


Fig. 4.4 Arrangement of neurons into layers in the feed-forward neural network

In the figure, it can be recognized that the various neurons of this network are arranged in layers. There is no connection between the neurons of one layer; usually, there is a complete connection between neurons of adjacent layers. The connexions between neurons that represent pathways for signal propagation are oriented, and each connexion is evaluated by weight modifying the intensity of the signal passing through. The absence of a specific connexion can be modelled by connexion with weight equal to zero. The first layer of the feed-forward network is called the input layer, the last layer is called the output layer and the other intermediate layers are called hidden layers. Typically, the feed-forward neural networks are implemented as networks with one, possibly two, hidden layers.

Feed-forward neural network is a comprehensively parallel distributed dynamic system where each unit, neuron, is working locally, separately. Such a neural network is characterized by considerable robustness, is resistant to damage and usually provides relevant outputs even if some elements are damaged. These properties are suitable for the construction of neurocomputers, i.e. computers based on neural networks. It is much easier to create an element with high degree of integration and some faulty elements than an element with lower integration where all elements are perfect.

In the neural network with a given topology as well as in the single neuron, we can distinguish two phases of activity. The phase when the network produces outputs on the basis of received inputs, topology and the network

settings remains generally constant. This phase is preceded by the learning phase when the network modifies its parameters (or even its topology, in some cases) using a training algorithm. These phases are called active, with adaptive dynamics of the network or neuron, in accordance with [1].

The presented feed-forward network topology is only one, albeit the most common, of the possible arrangements of neurons. There are many other concepts where the connection is realized differently; for example, the structure includes feedback, or each neuron is connected with all other neurons. Some of these concepts will also be presented; due to the universality of solutions; however, we will focus in detail on the feed-forward neural networks.

4.4 SINGLE NEURON

4.4.1 Mathematical model and neuron active dynamics

A schematic model of the single neuron is shown in the following Figure 4.5. A neuron can be divided into several parts: synapses valued by weights w, that carry inputs x in the neuron body; the neuron body in which the neuron inner potential ξ is obtained; the block of the activation transfer function σ; and neuron output y. Constant threshold h represents another input into the neuron body.

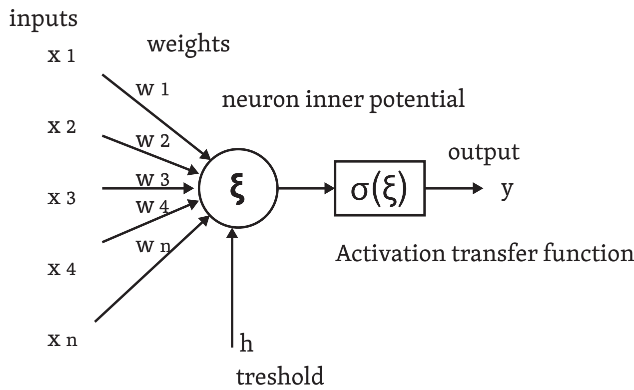


Fig. 4.5 Model of neuron

The neuron input is a vector with n elements, which may be generally feasible. Similarly, the threshold may be a real value and is often modelled as one of the inputs, as will be shown below. The neuron inner potential is a value that arises by comparing the weighted and numbered inputs with the threshold. Weights are again real numbers; therefore, they may be positive or negative, and thus model activation as well as inhibitory synapses. In general, the activation transfer function performs the nonlinear transformation of the inner potential to one, generally again real value. The neuron, therefore, performs a projection from {R}n -> R.

Let's now look more closely at the active dynamics of a neuron. After applying the input vector x, the neuron inner potential ξ can be quantified as

$$\xi = \sum_{i=1}^{n} w_i x_i - h \qquad (1)$$

Each element of the input vector is multiplied by the respective weight; this value is summed up in the neuron body for all elements of the input vector, and the threshold is deducted from the resulting value. The obtained value of the inner potential then becomes an argument of generally non-linear activation transfer function.

For practical reasons, the threshold is usually modelled as one of the weights so that the input vector as well as the vector of weights is extended by zero position. Input at the zero position is always considered as equal to 1 and zero weight is set to value equal to -h. In this case, the threshold becomes one of the weights and is subject to adaptation during training.

The overall response, i.e. the neuron output can then be expressed as

$$y = \sigma(\xi) = \sigma(\sum_{i=0}^{n} w_i x_i)$$

where $x_0 = 1$ and $w_0 = -h$. **(2)**

The activation transfer function σ is generally a nonlinear function transforming the value of the neuron inner potential. Assume its simplest type, sharp non-linearity, the following applies

$$\sigma(\xi) = \begin{cases} 1 \text{ if } \xi \geq 0 \\ 0 \text{ otherwise} \end{cases} \quad (3)$$

Further, assume a neuron with two real inputs x_1, x_2 and corresponding weights w_1, w_2 (n=2). The defined neuron implements projection from $\{R\}^2$ -> {0,1}. Consider that the inputs x_1 and x_2 represent points in the two-dimensional Euclidean space. In its active dynamics, the neuron responds to inputs received from this area and assigns them the values of 0 or 1. In this manner, it actually performs the classification of these plane points into two groups according to the output value of the activation transfer function, the neuron output.

When we take a closer look at the relationship expressing neuron response, we find that the classification of points in this particular case is determined by their positions toward the line defined by the activation function or, more precisely, by the neuron weights. In two-dimensional space, the dividing line distinguishing the two groups of points has, therefore, the following equation

$$w_1x_1 + w_2x_2 + w_0 = 0 \quad (4)$$

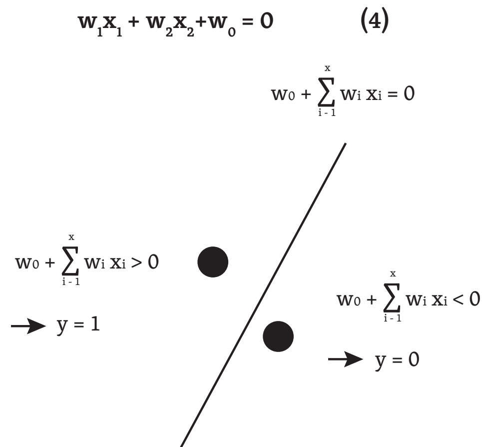


Fig. 4.6 Illustration of the neuron classification in a plane (according to http://www.byclb.com/TR/Tutorials/neural_networks/ch8_1.htm)

The given illustration can be generalized to n-dimensional Euclidean spaces where the n-dimensional vector of neuron weights represents a hyperplane dividing the space into two half-spaces.

Activation transfer functions most often used in feed-forward neural networks are shown in the Figure 4.7; with regard to the possibility of network learning, however, it is possible to choose any differentiable function.

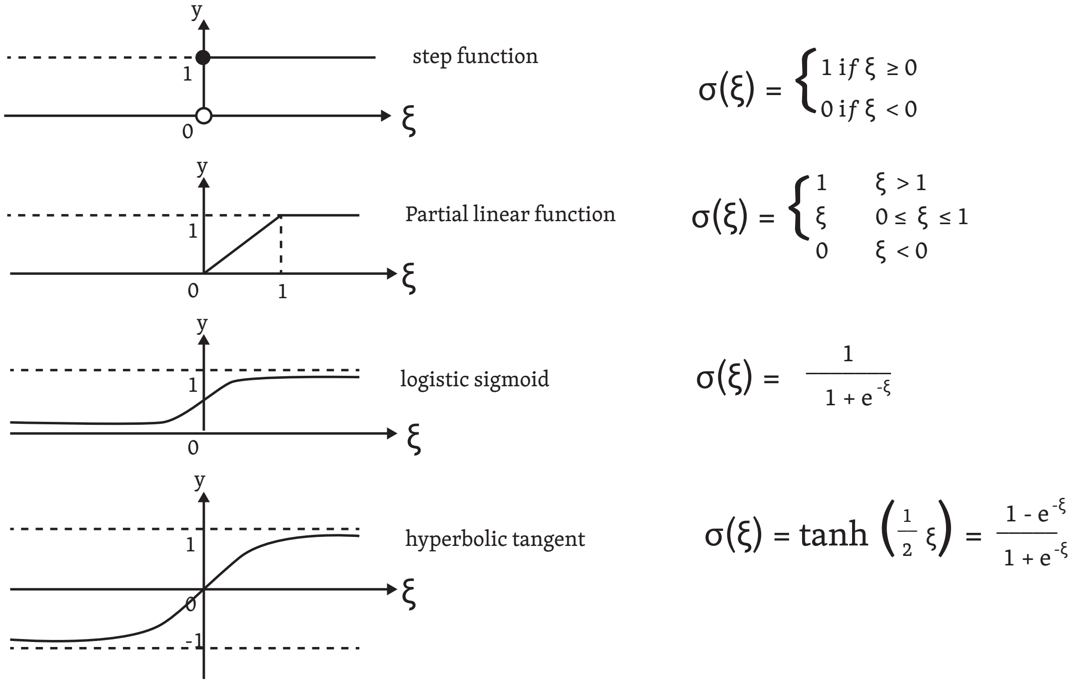


Fig. 4.7 Activation transfer functions

Concluding this section, it should be at least noted that different concepts of the neuron are also used, where the neuron output is calculated in another way. As an example, we can mention networks with radial basis (RBF networks), where the neuron calculates the distance of the input vector x from the weight vector w. Another case is represented by wavelet networks that quantify the correlation coefficient between the input signal x and the dilated and shifted mother wavelet function of each neuron. Wavelet networks are mainly used in the analysis of signals and images.

4.5 ADAPTATION DYNAMICS OF THE NEURON

4.5.1 General principles of neuron learning

During active dynamics, the neuron performs the transformation of input vectors to the output value. At this moment, the neuron parameters are constant. In contrast, adaptive dynamics is a process where the task is to configure these neuron parameters so that the neuron performs a desired transformation. The parameters adapted during neuron learning are usually only weights of the input neuron synapses, including the synapse representing threshold. Therefore, the input weights represent neuron memory. Neuron learning is ensured by an adaptive algorithm that adjusts the value of neuron weights. This process is generally carried out iteratively and repeatedly, based on the presented examples, when the adaptation algorithm disposes of a set of pairs of input values and their corresponding outputs. In exceptional cases, the sought weights can be determined by direct calculation.

Adaptive algorithms act like a human when looking for a solution based on analogy with known examples. Setting the neuron weights thus corresponds to finding the most pregnant transformation based on known values, assuming that the found transformation will be sufficiently general also for other unknown examples of the given domain.

Adaptation principles of the single neuron mentioned in next chapters can be easily generalized to feed-forward stratified (layered) neural networks. The basic advantage of neural networks is the possibility of finding a transformation even for those tasks that are analytically intractable or unsolvable. On the contrary, a disadvantage is represented by the fact that the resulting transformation is hidden in the network structure and cannot be used to explain the solution to the problem. Although the networks are powerful computational models for solving many difficult tasks, it should also be noted that they do not guarantee finding a sufficiently general transformation of the neural network.

4.5.2 Unsupervised learning

Adaptive algorithms can be divided into two main areas, known as unsupervised learning (learning without any teacher) or supervised learning (learning with a teacher).

Unsupervised learning is characterized by the fact that the adaptation algorithm has no criterion of correctness regarding the sought transformation of input data. It works on the principle of clustering, looking for "self-similar" elements in the input area. Based on the parameters of the presented samples of input data, it performs their sorting into groups without the possibility of assessing the accuracy of classification. The number of searched groups can be given in advance. No arbiter enters the adaptation; the entire learning is based only on the information contained in the input data. The algorithm can, therefore, be also used at the moment of network active dynamics when it is possible to simultaneously adjust also the network parameters on the basis of each new sample presented. In this way, learning can be permanent. In case of the single neuron, unsupervised learning is of no importance; however, it is used (for example) for the whole class of neural networks known as self-organizing maps that will be discussed in a separate subsection.

4.5.3 Supervised learning

In case of supervised learning, the adaptive algorithm disposes of a countable (pmax of elements) set of M pairs x and yd that represent inputs and corresponding correct outputs of the task solved. Therefore, it disposes of examples of proper behaviour, proper transformation from input vectors to output vectors. For instance, these examples may be obtained by observing and recording the input and output values of the system that we want to model. Direct transformation of input values to output values is not known. In this case, the analogy with human expert reasoning is obvious.

$$\mathbf{M = \{[x^1,y_d^1], [x^2,y_d^2] .... [x^{pmax},y_d^{pmax}]\}} \qquad (4)$$

The set of all available values represents a known part of the system behaviour. This set is then used by the adaptation algorithm for learning the network and also for testing its function. Generally, the set M of all available data is divided into two parts, the training set and the test set. The ratio of the number of elements in the training and test sets is not fixed. It must be chosen with regard to the nature of the problem solved and the data available for adaptation algorithm. With a sufficient number of input data, the training set usually constitutes about 70–80%, the rest of the pairs are then included in the test set.

Neuron training (and also the training of feed-forward networks, after generalization) is usually carried out iteratively; the algorithm gradually presents different elements of the training set to the neuron, determines its response to the presented entry, and corrects the neuron weights based on the deviation of its output from the desired output. Interval, in which all patterns of the training set are presented at least once, is called the learning epoch. Tens to thousands of epochs may be required to learn the network, depending on the complexity of the problem.

The moment of stopping the adaptation can be defined in various ways; most often by reaching the desired small error of the transformation over the training set or stopping its decline, by reaching the maximum number of epochs. In order to evaluate the moment when it is appropriate to terminate the adaptation (the found transformation is accurate enough and simultaneously does not lose generality), we often extract a third validation set (again non-overlapping) from the training set or the data available. During the adaptation, the performance

of the found transformation is periodically tested over it, and the adaptation process is terminated when the error decline over the validation set stops.

After completing the adaptation, the performance of the learned neuron or neural network is verified on the test set. The training and test set are, therefore, chosen as non-overlapping in order to ensure independent testing of successful adaptation over the test set. There can be more criteria for network performance and success of adaptation; we mostly use the mean-square deviation between outputs of the found transformation and the expected outputs.

The scope and content of the training set is often limited by the data already available. Nevertheless, we try to ensure utmost examples for the adaptation algorithm, covering the input space as evenly as possible

The success of the transformation found over the training set is then evaluated over the test set; in this manner, we estimate the success of the adaptation and generality of the transformation found also for other inputs than those occurring in the training set. Performance over the test set is an evaluated criterion of adaptation success.

If performance over the test set is good, we assume that it will be similarly good also for the inputs completely outside of the training and test set. However, it should be noted that we do not have this guarantee of generality for inputs yet unknown.

Typical steps of iterative learning of the neuron or supervised feed-forward neural network.

1. Pre-processing of input data
2. Definition of the training, test or validation set
3. Definition of the network structure /neuron parameters
4. Initialization of neuron weights, usually small random numbers
5. n=0; counter of learning epochs
6. Start: set the learning epoch to n=n+1
7. If the number of performed learning epochs >= max => stop learning, go to 14
8. n-th learning epoch:
9. Selection and presentation of one neuron input vector from the training set (deterministic, random)
10. Getting neuron responses, evaluation of classification errors by comparing the actual and expected out put => classification error
11. Correction of the neuron weights based on received errors
12. If the learning epoch is not finished (not all inputs from the training set are tested), go to point 9
13. At the end of the epoch, evaluate the error in classification over the whole training set. If the error is $<\varepsilon$, stop learning (point 14), otherwise go to point 4. (The error can be evaluated even more often; we can also use the validation set and other criterion for stopping adaptation)
14. Evaluate the success of network activity on the test set; if it is insufficient, go to point 4 or 3. Alternatively, stop the algorithm as unsuccessful.

4.5.4 Hebb learning

It represents the oldest intuitive rule for a neuron with binary inputs and output. The rule was defined in 1949 by Donald Hebb in the study of conditioned reflexes. He assumed that the conditioned reflexes are established on the basis of strengthening or weakening the bonds between single neurons. He assumed the following: the bond between two neighbouring neurons strengthens if they are active at the same time and weakens in dissentient activation. The Hebb rule is, therefore, based on a neurophysiological analogy. For the neuron with binary input x, the weights w and expected output y, the Hebb rule can be written as follows:

1. If the neuron is excited correctly ($y=1$; $y_d=1$), then the weight of connexions wi inducing this excitation are in the next discrete time step n+1 strengthen by

$$\Delta\ ({}_{n+1}w_i = w_i + \Delta,\ \forall i{:}x_i = 1 \qquad (5)$$

2. If the neuron is excited incorrectly ($y=1$; $y_d=0$), then the connexions inducing this excitation weaken by

$$\Delta\ ({}_{n+1}w_i = w_i - \Delta,\ \forall i{:}x_i = 1) \qquad (6)$$

3. If the neuron is not excited ($y=0$), nothing happens (the weights do not change)

4.5.5 Delta rule

Another rule, originally heuristic, applicable also for generally real neuron inputs and outputs, is the delta rule. It can be written as

$${}_{n+1}w_i = {}_{n}w_i + \mu^*(y_d-y) \qquad (7)$$

or as (using vectors, bold font)

$${}_{n+1}\mathbf{w} = {}_{n}\mathbf{w} + \mu^*(y_d-y) \qquad (8)$$

where μ is a suitably chosen constant between (0,1), affecting the speed of adaptation.

The delta rule applies exactly to linear neurons, i.e. neurons with linear activation transfer function; after modification; however, it is also useful for neurons with nonlinear activation transfer function, as will be shown later in the chapter devoted to learning of multilayer feed-forward networks.

4.5.6 Neuron learning according to Widrow

It is based on the geometric interpretation of neuron learning for a neuron with binary output. In accordance with the chapter "Mathematical model and neuron active dynamics", the idea is that the neuron divides the output space into two half-spaces and thus performs classification of input vectors into one of them.

If the classification of the given pattern from the input set is correct, there is no need to adjust the weights. If the classification is erroneous, the classified point is located in the wrong part of the half-space split by the dividing hyperplane.

$$\mathbf{w} \cdot \mathbf{x} = 0 \qquad (9)$$

The distance of this erroneously classified point from the dividing hyperplane can be expressed as

$$d = \left| \frac{\mathbf{w}}{|\mathbf{w}|} \cdot \mathbf{x} \right| \qquad (10)$$

where this value can be used as a quantifier of classification error.

We introduce an error function that expresses the total distance from the hyperplane for all misclassified patterns X_f of the training set as a function of vector **w**.

$$E(\mathbf{w}) = \sum_{\mathbf{x} \in \mathbf{X}_f} \left| \mathbf{w} \cdot \mathbf{x} \right| = \sum_{\mathbf{x} \in \mathbf{X}_f} \mathbf{w} \cdot \pm \mathbf{x} \quad (11)$$

where +**x** will be used in case of false negative classification, and -**x** will be used in case of false positive classification.

The aim of the adaptation algorithm is to minimize the error function E over all patterns of the training set by changing the weights **w**. Minimisation of the error function is performed in the direction (the function already contains change of signs according to misclassification) of gradient E(**w**) and the following applies:

$$_{n+1}\mathbf{w} = {}_{n}\mathbf{w} + \mu^{*}\nabla E(\mathbf{w}_n) \quad (12)$$

$$\nabla E\left({}_{n}\mathbf{w}\right) = \sum_{\mathbf{x} \in \mathbf{X}_f} \pm \mathbf{x} \quad (13)$$

Therefore, the function E(**w**) is an error function over all classified patterns of the training set in one learning epoch. Of course, this procedure can be subsequently repeated for the next epochs until completing the adaptation of weights. Under this rule, we can proceed not only through individual epochs but also iteratively for each input pattern x, and thus approach the minimum of error function E(**w**) in parts.

Geometric interpretation of the binary neuron adaptation is as follows: during the adaptation, the dividing hyperplane (defined by the vector of neuron weights) gets shifted. The weight vector **w** represents a normal vector of the dividing hyperplane.

The algorithm adjusts the weight vector **w** according to the rule for misclassified patterns $_{n+1}\mathbf{w}_n = \mathbf{w}_n \pm \mathbf{x}$; a new weight vector is thus obtained by adding up the vector of original weights and the enclose, the erroneously classified vector. If the enclosed pattern **x** is misclassified and if simultaneously applied| $|\mathbf{x}| >> |\mathbf{w}|$ for the given step n, this iteration performs a significant correction of vector **w**, and vector **w** considerably approaches vector **x** in the space. Usually, due to gradual learning with increasing iterations, the rotation angle of vector **w** decreases and the location of dividing hyperplane stabilizes.

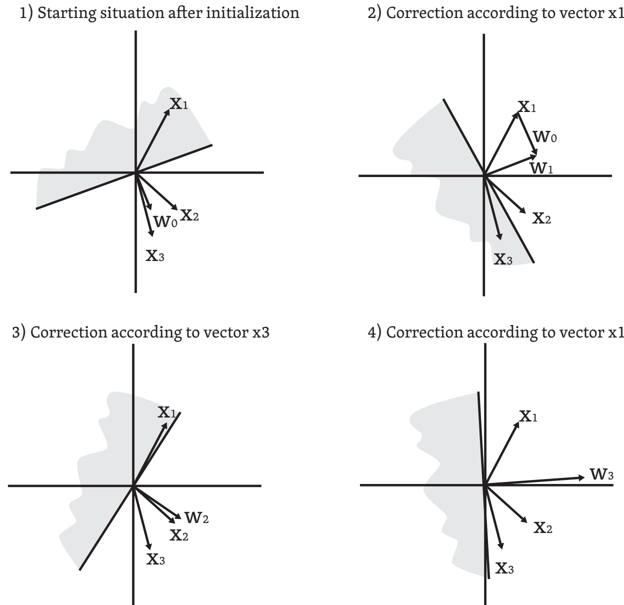


Fig. 4.8 Geometric illustration of the adaptation in case the vectors X_1, X_2, X_3 should be correctly classified into the white section of the half-plane

4.6 CLASSIFICATION CAPABILITIES OF THE SINGLE NEURON

Within the description of adaptation and active dynamics of the single neuron with binary output, we observed that the vector of neuron weights forms a hyperplane normal vector. The single neuron is thus able to classify the input vectors into two classes. In the following sections, we will introduce the classification capabilities and limitations of the single neuron using a model of logic functions. We assume a neuron $\{0,1\}^2 \rightarrow \{0,1\}$ with two binary inputs (x_1, x_2), one binary output and threshold equal to $x_0 = 1$.

4.6.1 Implementation of logical function AND

Logical function AND of two variables is defined as

X_1	X_2	X_1 AND X_2
0	0	0
0	1	0
1	0	0
1	1	1

Tab. 4.1 AND logic function

In the plane, the AND logic function can be shown according to Figure 4.8.

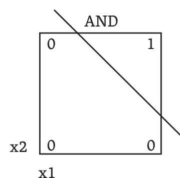


Fig. 4.9 AND logic function in the plane

The aim is to adjust the neuron weights so that the neuron implements the logic function AND.

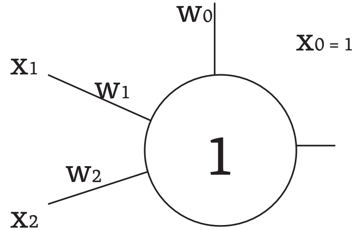


Fig. 4.10 Neuron after implementing the function AND

Adjustment of the weights can be done by inequalities when the following conditions must be met for correct classification

$(0, 0, 1) \cdot (w_1, w_2, w_0) < 0$
$(1, 0, 1) \cdot (w_1, w_2, w_0) < 0$
$(0, 1, 1) \cdot (w_1, w_2, w_0) < 0$
$(1, 1, 1) \cdot (w_1, w_2, w_0) < 0$

thus

$w_1 0 + w_2 0 + w_0 1 < 0$
$w_1 0 + w_2 1 + w_0 1 < 0$
$w_1 1 + w_2 0 + w_0 1 < 0$
$w_1 1 + w_2 1 + w_0 1 > 0$

The solution to this system of inequalities are values such as $w_1 = 1$, $w_2 = 1$ and $w_0 = -1.5$.

$1*0 + 1*0 - 1.5*1 < 0 \Rightarrow y = 0;$
$1*0 + 1*1 - 1.5*1 < 0 \Rightarrow y = 0;$
$1*1 + 1*0 - 1.5*1 < 0 \Rightarrow y = 0;$
$1*1 + 1*1 - 1.5*1 > 0 \Rightarrow y = 1;$

Therefore, according to this procedure, the AND logic function can be smoothly implemented by the single neuron.

To illustrate the method of solving the problem, suppose for a moment that w_0 is constant, $w_0 = -1.5$. We are, therefore, looking for configuration of weights w_1 and w_2 with regard to which the neuron outputs will be correct for all four possible combinations of inputs x_1 and x_2. If plotting the dependence of the number of erroneous neuron outputs on the weights w_1 and w_2 in the interval <-2.5;2.5> in the graph, we get the picture shown in Figure 4.11.

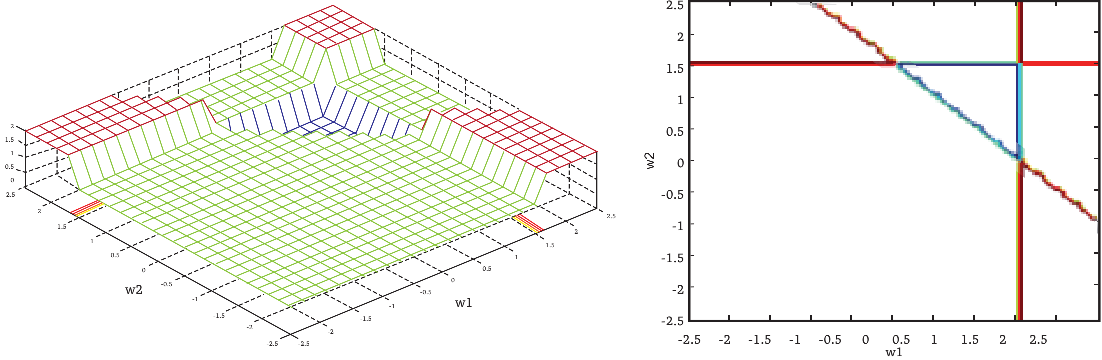


Fig. 4.11 Graphical solution to the adjustment of neuron weights for implementing the function AND

From the figure, it is evident that the solution, i.e. possible values of searched weights, is represented by the blue triangle. Previously verified adjustment of weights w1 = 1 and w2 = 1 is located inside the blue area. In this area, the error function is zero, the neuron implements the AND logic function. The area, where the error function value is equal to 2, is marked in red; the area with the error function value equal to 1 is marked in green.

In general, adaptive algorithm is not able to achieve zero value of the error function over the training set, as in the case of implementation of AND logic function. Nevertheless, it proceeds in single steps through the space of error function and tries to find its minimum value by correction of neuron weights. The complexity of this function depends on the number of inputs to the training set and the number of neuron weights.

4.6.2 Implementation of OR and NOT logic functions

The analogous conclusion is reached in case of implementing OR and NOT logic functions using the single neuron.

X_1	X_2	X_1 OR X_2	NOT X_1
0	0	0	1
0	1	0	1
1	0	0	0
1	1	1	0

Tab. 4.2 OR and NOT logic functions

Solution to the system of inequalities for OR logic function can be as follows, i.e. $w_1 = 1$, $w_2 = 1$ and $w_0 = -0.5$.

$w_1 0 + w_2 0 + w_0 1 < 0$ $\quad 1{*}0 + 1{*}0 - 0.5{*}1 < 0 \Rightarrow y = 0;$
$w_1 0 + w_2 1 + w_0 1 > 0$ $\quad 1{*}0 + 1{*}1 - 0.5{*}1 > 0 \Rightarrow y = 1;$
$w_1 1 + w_2 0 + w_0 1 > 0$ $\quad 1{*}1 + 1{*}0 - 0.5{*}1 > 0 \Rightarrow y = 1;$
$w_1 1 + w_2 1 + w_0 1 > 0$ $\quad 1{*}1 + 1{*}1 - 0.5{*}1 > 0 \Rightarrow y = 1;$

Based on solving NOT logic function, we then get when $w_2 = 0$

$w_1 1 + w_0 1 < 0$ $\quad -1{*}1 + 0.5{*}1 < 0 \Rightarrow y = 0;$
$w_1 0 + w_0 1 > 0$ $\quad -1{*}0 + 0.5{*}1 > 0 \Rightarrow y = 1;$

4.6.3 Implementation of XOR logic function

The previous paragraph demonstrated that the computational power of the single neuron is quite sufficient for calculating AND, OR and NOT logic functions. What is the situation in case of implementing XOR logic function?

Assume the same neuron as in the previous case; for simplification, the threshold is now designated as h (we know that it is implemented by the weight w_0 and constant input x_0).

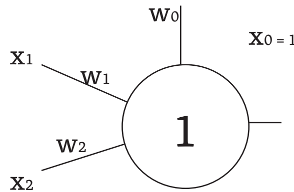


Fig. 4.12 Neuron for the implementation of OR and NOT functions

Table of values of XOR function and its graphical representation is shown in Table 4.3.

X_1	X_2	X_1 XOR X_2
0	0	0
0	1	1
1	0	1
1	1	0

Tab. 4.3 XOR logic function

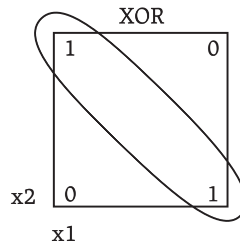


Fig. 4.13 XOR logic function in the plane

It is necessary to simultaneously meet the following requirements:

$$w_1 0 + w_2 0 < h \Rightarrow 0 < h \qquad (13)$$

$$w_1 0 + w_2 1 > h \Rightarrow w_2 > h \qquad (14)$$

$$w_1 1 + w_2 0 > h \Rightarrow w_1 > h \qquad (15)$$

$$w_1 1 + w_2 1 < h \Rightarrow w_1 + w_2 < h \qquad (16)$$

If analysing the relationships in more detail, we find that:

(13) implies that the threshold is positive.

(13) and (14) imply that w_2 is positive and greater than the threshold.

(13) and (15) imply that w_1 is positive and greater than the threshold.

Can the sum of two positive values, both greater than the threshold, be simultaneously less than the threshold, as required in point (16)? It is obvious that this cannot be. The prerequisites for the implementation of XOR function by a single neuron thus cannot be met, and the single neuron cannot implement XOR logic function.

4.6.4 Summary of the classification capabilities of the single neuron

The single neuron with binary output is able to distinguish only two linearly separable classes. The neuron divides the input space into two subspaces by the hyperplane, the normal vector of which is formed by the vector of neuron weights. Neuron learning is usually represented by an iterative process when the vector of neuron weights is adjusted and the position of the dividing hyperplane is consequently changed based on evaluating the classification of input vectors of the training set into one of the output classes. Evaluation of the classification success is realized through the error function where learning represents an attempt to minimize it.

The XOR logic function is not linearly separable; therefore, it cannot be implemented by the single neuron. This limitation led to a crisis in the development of artificial neural networks, and was used by Minsky in his arguments against the concept of artificial neural networks.

However, a way out of this situation exists; it consists in using feed-forward neural networks with multiple layers of interconnected neurons.

4.7 LITERATURE

[1] Šíma, J., Neruda, R.: Teoretické otázky neuronových sítí (Theoretical Questions of Neural Networks), MATFYZPRESS, 1996, ISBN 80-85863-18-9.

[2] Jan, J.: Číslicová filtrace, analýza a restaurace signálů (Digital Filtering, Analysis and Restoration of Signals). University of Technology in Brno, VUTIUM, 2002. ISBN 80-214-1558-4.

[3] V. Kvasnička, L. Beňušková, J. Pospíchal, I. Farkaš, P. Tiňo, and A. Kráľ. Úvod do teórie neurónových sietí (Introduction to the Theory of Neural Networks), IRIS, Bratislava, 1997, ISBN 80-88778-30-1.

[4] Volná, E.: Neuronové sítě 1 (Neural Networks 1). Lecture notes. Ostrava University in Ostrava, Ostrava, 2008.

[5] Volná, E.: Neuronové sítě 2 (Neural Networks 2). Lecture notes. Ostrava University in Ostrava, Ostrava, 2008.

[6] Osička P.: Jednoduché modely neuronu (Simple Neuron Models), [online], 2012, phoenix.inf.upol.cz/~osicka/courses/uns/uvod.pdf.

5 NEURAL NETWORKS – PERCEPTRONS

5.1 BASIC INFORMATION

The following text is part of the course “Artificial Intelligence” and is intended primarily for students of Mathematical Biology study programme. This chapter deals with the most common concept of feed-forward artificial neural networks, usually not precisely acknowledged as single-layer or multilayer perceptron(s). The chapter describes the organization, adaptation and active dynamics of these networks, and analyses and illustrates their classification capabilities. More specifically, it deals with the basic adaptation algorithm for multilayer feed-forward networks, the algorithm of backward propagation of errors (error backpropagation).

5.2 LEARNING OUTCOMES

Mastering the learning text will enable students to:

- *Become familiar with the basic concepts of feed-forward neural networks, and with the principles of adaptation, active and organization dynamics of the network.*
- *Define the organization dynamics of single-layer and multilayer perceptrons (MLPs).*
- *Generalize the findings from previous chapter for expressing classification possibilities of the single-layer perceptron.*
- *Derive the classification possibilities of multilayer perceptrons in 2D space and demonstrate their classification capabilities via implementation by the XOR multilayer perceptron.*
- *Understand the consequence of Kolmogorov’s theorem for MLPs.*
- *Understand MLP adaptation dynamics and explain the formal mathematical expression of backpropagation algorithm, and describe the importance of its individual parameters.*
- *Identify the application fields for MLPs.*

5.3 FEED-FORWARD NEURAL NETWORKS

5.3.1 Introduction

Artificial neural network is a mathematical model comprising simple interconnected elementary processing units, the neurons.

We can trace some typical general features of artificial neural networks; usually, the following applies:

- *Procedural elements are very simple; however, they work simultaneously in large numbers*
- *Network activity is parallel*
- *Individual procedural elements operate independently, locally and often asynchronously*
- *Neurons are mutually, more or less densely, interconnected with oriented connections, evaluated by numerical weights*
- *The weights are adjusted through controlled network learning and their setting represents the internal memory of the network knowledge*
- *They represent a robust solution of partial damage to the network or incompleteness of the input data*

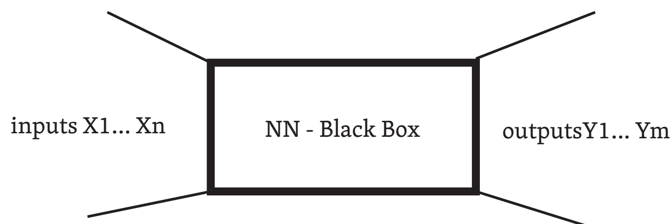


***Fig. 5.1** General scheme of a neural network*

Artificial neural networks implement projection (transformation) from the input space (input vectors x) into the output space (output vectors y). From the external point of view, the artificial neural network can be considered as a "black box" because the found transformation is difficult to interpret, except for elementary cases; there is no formalized analysis of artificial neural networks.

Artificial neural networks can be divided according to a number of criteria:

- *According to the type of used computational elements (formal neuron)*
- *According to the network topology, i.e. the arrangement of elements (recurrent, forward, reciprocal links)*
- *According to the training algorithm*
- *According to structuring*
    - *Unstructured – all neurons are equivalent (Hopfield)*
    - *Structured – typically contain an arrangement of unstructured subnets*
        - *Hierarchical*
        - *Competitive*

The scheme below presents one of the possible divisions of artificial neural networks according to Lippmann, including the typical representatives of the given class.

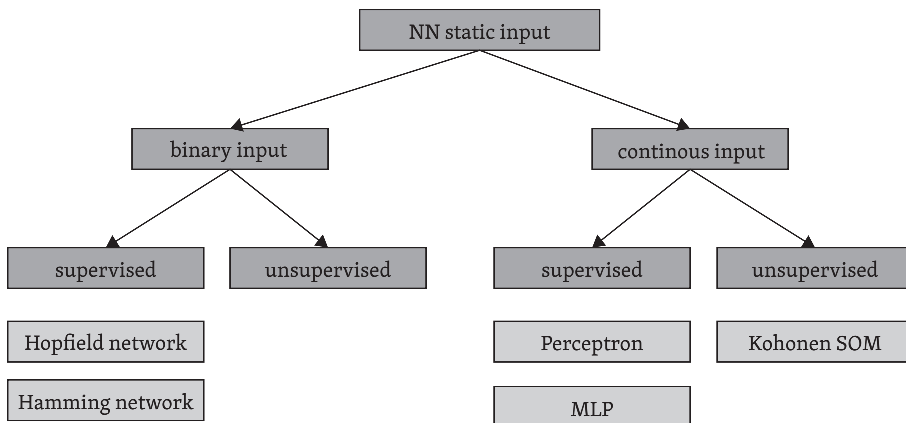


Fig. 5.2 possible division of neural networks according to Lippmann

5.3.2 Dynamics of neural networks

As in the case of the single neuron, we can distinguish three phases of the establishment and operation of artificial neural networks, three types of dynamics.

Organization dynamics specifies the network topology, establishment of the network and the possible changes in its architecture. It is mostly constant, but not necessarily. We can distinguish two basic organization dynamics – cyclic and acyclic. Acyclic dynamics can always be divided into separate layers.

Within the active dynamics, network neurons occur in a defined steady state, with setting the network weights. Generally, it is possible to consider continuous function of the network; practically, however, it is a clocked system with discrete time when the neuron responses are calculated in individual discrete steps. The status of output neurons represents the output of the neural network, the result of the calculation. After stabilizing the active network dynamics, we dispose of a steady output per given input, determined by the state of output neurons, representing the function of network transformation.

Adaptation dynamics represent the process of neural network learning that leads to setting the weights of individual network neurons. Analogous to the above, we can consider a continuous function; practically, however, this process is again divided into individual discrete adaptation steps. The process results in set weights of the neural network, entering into active network dynamics.

5.3.3 Historical Rosenblatt perceptron

It represents the model of classical neural network inspired by the human eye. The network modelled perception, hence the name "perceptron", was tasked with recognizing different recorded traits using optical sensors arranged in an array of 20x20 elements.

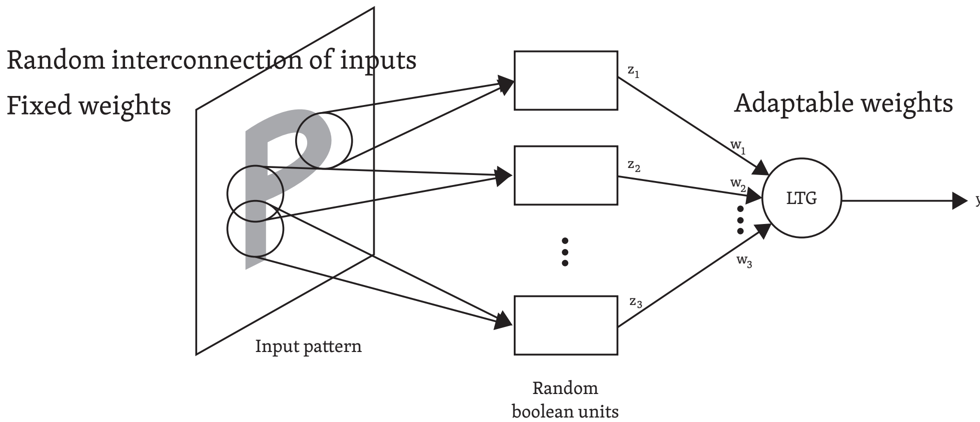


Fig. 5.3 Rosenblatt perceptron

The concept of perceptron has its roots in the field of artificial intelligence and now, regardless of its original meaning, it is used for all feed-forward neural networks, i.e. networks with layered arrangement of neurons and unidirectional signal propagation from input to output. Neurons in one layer are independent and can work in parallel; neurons of the next layer are activated upon calculation of all neurons in the previous layer.

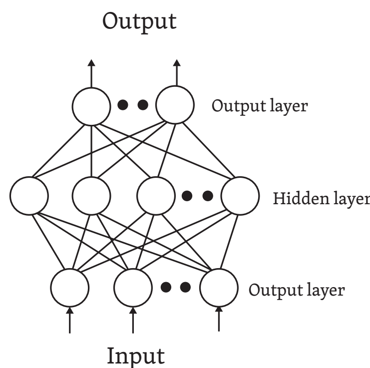


Fig. 5.4 General structure of the multilayer feed-forward network

The feed-forward neural network is, thus, represented by a directed graph where each node is connected by valued oriented (generally on both sides) edges. Evaluations of these edges (weights) are signal processing parameters. In feed-forward networks, the signal is spread through individual layers from input towards output. We distinguish input, output and hidden nodes; edges represent signal flow. A connection between two neurons (nodes) is oriented; the direction of information flow (from which neuron and to which neurons) is, therefore,

given. A property of the connection is the weight that determines the ability and the intensity of information transfer. The weight can be valued both positively and negatively; neurons may stimulate or suppress each other.

5.4 SINGLE-LAYER PERCEPTRON

5.4.1 Organization, active and adaptation dynamics

Conceptually, the single-layer perceptron is the simplest layered network. It represents M independent neurons operating in parallel. Each of these neurons, thus, implements the transformation of the input vector to an output value independent of other neurons.

From the perspective of organization dynamics, the network consists of N neurons in the input layer and the layer with M output neurons. The input layer is not composed of neurons within the meaning of the established definition; it only represents nodes implementing an identical copy of the values of the input vector to the inputs of all neurons. Both layers are completely interlinked; each j-th output neuron is connected to all input neurons.

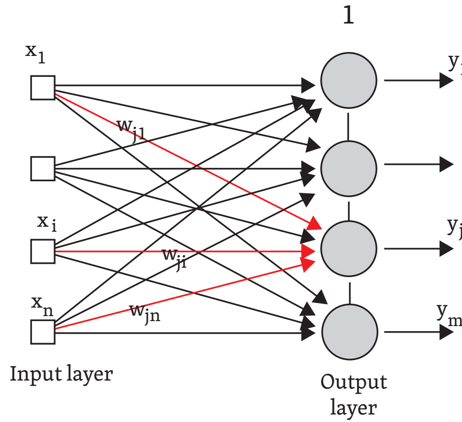


Fig. 5.5 Single-layer perceptron

During active dynamics, the network implements $R^n \rightarrow R^m$ transformation set during adaptation dynamics, generally not considering the characteristics of output functions (identity functions). Specific field of output values is given by activation transfer functions of the output neurons. For example, it is the implementation of $R^n \rightarrow (0,1)^m$ transformation in the case of sigmoid activation functions approximating sharp nonlinearity.

If introducing the following rule for marking the neuron weights,

$$w_{j,i} = w_{\text{where, from where}} = {}_{\text{where}}w_{\text{from}} \text{ where } = {}_{\text{j-th neuron}}w_{\text{ i-th weight}} \quad (1)$$

we can write for the response of j-th neuron in active dynamics

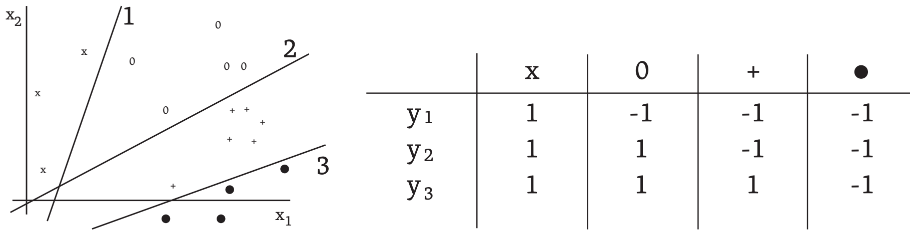
$$y_j = \sigma(\xi) = \sigma\left(\sum_{i=0}^{n} w_{ji} x_i\right) \quad (2)$$

The independence of neurons in the output layer results in the fact that each behaves completely independent of other neurons, during active as well as adaptation dynamics. In this manner, active and adaptation dynamics of the single-layer perceptron completely corresponds to the single neuron.

5.4.2 Classification possibilities

For simplicity, consider a single-layer perceptron implementing $R^2 \rightarrow \{0,1\}^m$ transformation, i.e. the classification of elements of two-dimensional space into two classes. Recall that the single neuron with binary output can distinguish two linearly separable classes in space, where the dividing line is a straight line with normal vector corresponding to the vector of neuron weights.

In the case of single-layer perceptron, there can be m dividing lines passing through the space, one for each neuron of the output layer. The actual multiplication of neurons in the output layer, however, does not bring any shift in the perceptron classification possibilities compared to the single neuron because the neural network lacks the option of combining the outputs of individual neurons and thus allow classification into more classes.



	x	0	+	●
y_1	1	-1	-1	-1
y_2	1	1	-1	-1
y_3	1	1	1	-1

Fig. 5.6 Division of Euclidean space by a single-layer perceptron with three neurons

A way out of this situation is the addition of another layer of neurons and the introduction of the concept of multilayer perceptrons.

5.5 MULTILAYER PERCEPTRON

5.5.1 Organization and active dynamics

General organization dynamics of the multilayer perceptron is shown in the Figure 5.7.

Fig. 5.7 Organization dynamics of a multilayer perceptron

Compared to one-layer perceptron, in addition to the input and output layers, the network topology was extended by at least one other layer, the so-called hidden layer. Individual neurons of adjacent layers are completely interlinked while neurons within a layer are not interconnected. The number of neurons in the hidden layers can be different; it is chosen according to the nature of the problem being solved, usually in the range between the number of input and output neurons.

The input and output layers are defined identically as in the case of single-layer perceptron; therefore, in the mode of active dynamics, the network again implements $R^n \rightarrow R^m$ transformation, depending on the nature of output activation functions of neurons in the output layer. However, the neurons of hidden layers are involved in the calculation of output values.

Active dynamics of the multilayer perceptron proceeds in single clocked discrete time steps (k) where outputs of j-th neurons in the layer (k) are always parallelly and locally computed, again according to the relationship

$$ {}^{k}y_{j} = \sigma(\xi) = \sigma\Big(\sum_{i=0}^{n} {}^{k}w_{ji} x_{i}\Big) \qquad (3) $$

The only difference is that n means not only the size of the input vector (the number of neurons in the input layer) as in the case of single-layer perceptron; more generally, it expresses the number of neurons in the previous (k-1) layer for the k-th layer.

5.5.2 Illustration of classification possibilities of multilayer perceptrons

For simplicity, assume again the classical Euclidean two-dimensional input space in which we want to classify individual points into two classes. The input layer will contain the number of neurons equal to the dimension of the input vector, i.e. two neurons. In the case of classification into two classes, the output layer will be formed by one neuron with the output function in the form of sharp nonlinearity, i.e. providing {0,1} outputs. Specific number of neurons in the hidden layers will not be substantial to this illustration.

The input space and classified sets are schematically shown in the Figure 5.8.

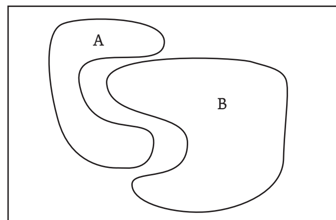


Fig. 5.8 Schematic representation of classified sets

We have already noted that the single-layer perceptron will provide lines through the space but does not have the ability to combine individual subspaces. That brings up another layer in a two-layer perceptron.

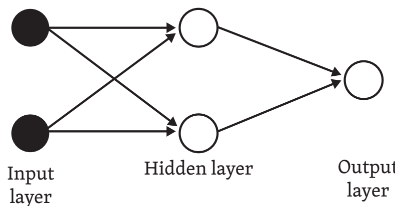


Fig. 5.9 The hidden layer in a multilayer perceptron

A two-layer perceptron with two neurons of the hidden layer could, therefore, divide the input space into 4 areas, for example as follows:

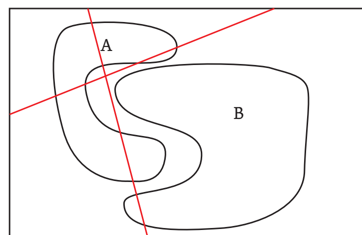
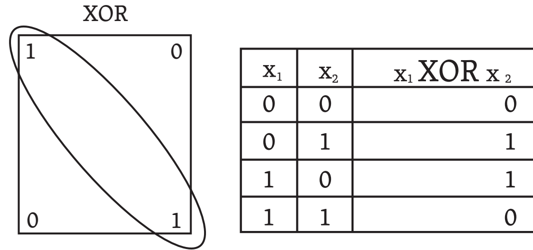


Fig. 5.10 Division of the space by a multilayer perceptron

In this case, classification into classes A and B would not be completely perfect because the left lower area contains predominantly patterns from class A as well as some patterns of class B. The concept of active dynamics in a two-layer perceptron is such that the first layer divides the space into half-planes and the second layer performs their logic conjunction. It thus allows the creation of arbitrary convex region. Of course, the network can learn quite differently; the illustration only shows that this is possible.

Let’s now return to the XOR logic function where the impossibility of its quantification by single neurons and the absence of adaptation algorithm for multilayer networks led to stagnation of their development. Is a two-layer perceptron able to implement the XOR logic function?



x_1	x_2	x_1 XOR x_2
0	0	0
0	1	1
1	0	1
1	1	0

Fig. 5.11 XOR logic function

The XOR function can be expressed using elementary logic functions such as XOR (X1, X2) = (X1 OR X2) NOT (X1 AND X2). For example, the implementation by the neural network might look as shown in the Figure 5.12.

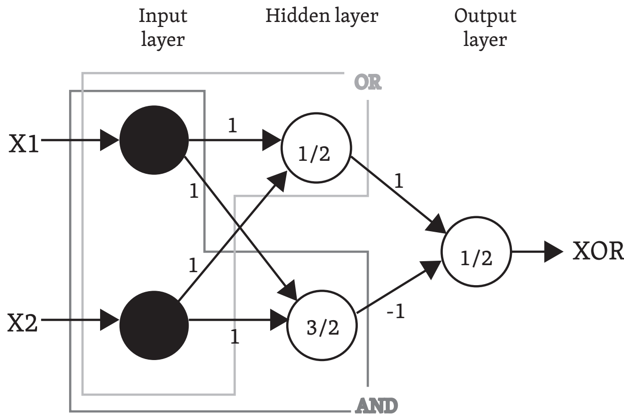


Fig. 5.12 Possible implementation of XOR logic functions by a multilayer perceptron

Briefly, consider a model of three-layer perceptron with three neurons in the first hidden layer and two neurons in the next hidden layer.

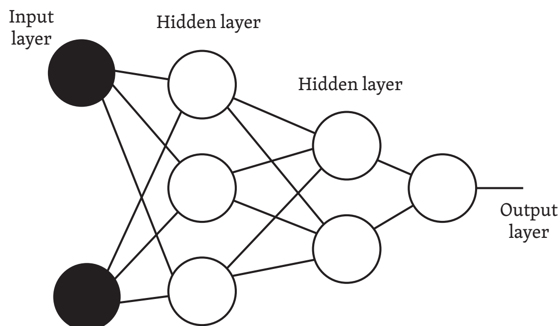


Fig. 5.13 Three-layer perceptron

The idea of active dynamics is such that the first layer divides the space into half-planes, the second layer divides the space into convex subregions, and the third layer performs the union of these regions. Of course, the network can learn quite differently; the illustration only shows that this is possible. Multilayer perceptron with two hidden layers and binary neurons is, thus, a universal approximator of any Boolean function of N variables.

5.5.3 Classification possibilities of multilayer perceptrons – Kolmogorov's theorem

In expressing the classification capability and computing power of multilayer perceptrons, we can refer to the Kolmogorov's theorem, which says:

"Each multi-dimensional real-valued continuous function $f(x_1,\dots, x_n)$ of N variables can be accurately expressed as a linear combination of a finite number of continuous non-linear functions of one variable."

$$f(x_1,\dots,x_n) = \sum_{q=1}^{2n+1} \Psi_q \left(\sum_{p=1}^{n} \Phi_{pq}(x_p) \right) \tag{4}$$

where ψ, ϕ are nonlinear functions of one variable.

If perceiving the above theorem in the realization of artificial neural networks and looking for an analogy, we will find that the multilayer perceptron provides sufficient means for implementing the theorem. Individual non-linear functions required by the Kolmogorov's theorem can be implemented using non-linear functions such as sigmoid functions of single neurons. Linear combinations and, therefore, the summation relations of the Kolmogorov's theorem then correspond to the calculation of activation functions of single neurons in hidden layers.

The presented relationship actually corresponds to the calculation implemented by a multilayer perceptron with n input neurons, n neurons of the first hidden layer with functions Φ_{pq} (4), 2n + 1 neurons of the second hidden layer with functions Ψ_q and one output neuron. The condition is to use neurons with continuous, monotonically increasing activation functions in hidden layers.

The Kolmogorov's theorem can be formulated more generally, according to [1].

$$f(x_1\dots x_n) = \sum_{q=1}^{2n+1} \Psi \left(\sum_{p=1}^{n} \lambda^{pq} \Phi_q(x_p) \right) \tag{5}$$

where the requirements for generally different activation transfer functions of neurons in each layer were reduced to multiplication of one function ϕ_q with coefficient λ_{pq}.

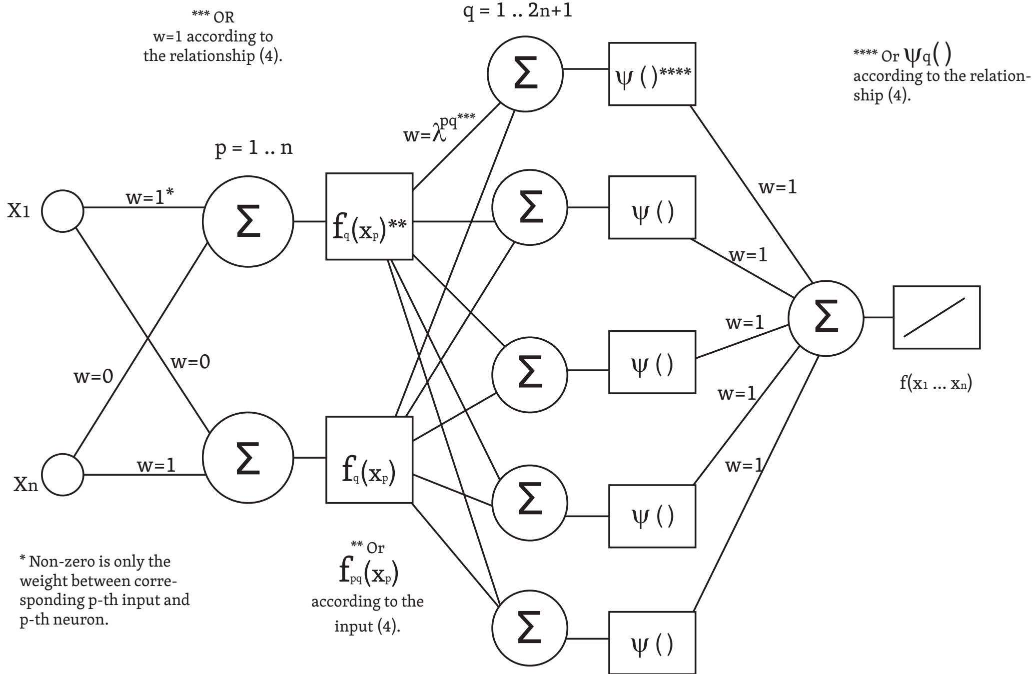


Fig. 5.14 Kolmogorov's theorem for a multilayer perceptron

According to [1], the choice of Ψ a Φ functions may be problematic due to the following reasons: for an accurate approximation of the function f(x), these sought functions may have properties that do not allow their use as neuron transition output functions because Ψ functions are not generally smooth.

Despite the relativization of the above conclusion of Kolmogorov theorem for the multilayer perceptron (inapplicability of general functions such as neuron transfer functions), it has been demonstrated [3] that the multiplication of the number of neurons in hidden layers can result in the universal approximation ability of a multilayer perceptron provided that we are satisfied only with an arbitrarily accurate approximation of the function f(**x**), not its absolutely exact representation.

The consequence of Kolmogorov's theorem for multilayer perceptron can be summarized as follows: the multilayer perceptron with two hidden layers, a sufficient number of neurons with continuous monotonically increasing activation functions, can be a universal approximator of any function of N variables. This represents significant computational power. The power is even so great that the classification into linearly separable sets is sufficient for approximate solutions to many real problems.

In the case of a double-layer perceptron, i.e. the perceptron with one hidden layer, we can refer to the results of Hornik and Leshno published in [1].

Their findings show that the multilayer perceptron with one hidden layer, a sufficient number of neurons (N*(2N+1), as it seems) and non-polynomial (sigmoidal) activation functions of neurons in hidden layers is a universal approximator for any function of N variables.

5.5.4 Adaptation dynamics

Recall that the adaptation dynamics of the single neuron, as well as single-layer perceptron, is based on supervised learning using the training and test sets of gradually presented vectors. Suppose the training set with p elements in the following form

$$\mathbf{T} = \{[\mathbf{x}_1, \mathbf{y}_{d1}], [\mathbf{x}_2, \mathbf{y}_{d2}] \ldots [\mathbf{x}_p, \mathbf{y}_{dp}]\} \qquad (6)$$

The algorithm of adaptation dynamics involves gradual presentation of the network training patterns and iterative adjustment of weights based on a deviation detected at the output from the network. This deviation for the p-th training set pattern is determined by error vector ep.

$$\mathbf{e}_p = \mathbf{y}_{dp} - \mathbf{y}_p \qquad (7)$$

One learning step (step t) can be represented by the following sequence of operations:

1. Presentation of input $\mathbf{x}_p$ to the network.
2. Finding the network response $\mathbf{y}_p$.
3. Calculation of deviation $\mathbf{e}_p$ for the last layer of the network.
4. Calculating the deviation per neurons of previous layers.
5. Local adjustment of network neuron weights on the basis of calculated deviation for each neuron.

These steps are repeated until meeting the conditions for stopping the adaptation.

In the case of a single neuron and single-layer perceptron, determining the deviation and subsequent adjustment of weights is easy; however, the situation is more complicated in multilayer perceptrons. Regarding the multilayer perceptron, the adaptation algorithm is able to directly determine only a deviation at the network output layer, and yet established algorithms could only adapt the last weights in the output layer. Therefore, the essential question is how to determine the error of neurons in hidden layers of the multilayer perceptron. This situation is addressed by the so-called algorithm of error backpropagation, the discovery of which resulted in rapid development of interest in neural networks after a long downturn period.

5.5.5 Algorithm of error backpropagation (BP algorithm)

The error backpropagation algorithm is a basic learning algorithm for multilayer perceptrons. It will be derived in the following sections. Vectors are marked as with a line above the letter, like $\overline{x}$.

Network learning can be considered as an optimization process when the network adjusts its parameters to minimize the error function resulting as a difference between the actual and desired outputs. For one input vector, the network implements the following transformation

$$\overline{y}_p = \varphi(\overline{x}_p) \qquad (8)$$

The error vector obtained after presenting the p-th vector of the training set can be written as

$$\overline{e}_p = \overline{y}_{dp} - \overline{y}_p \qquad (9)$$

Instant square deviation of the network classification for the p-th vector of the training set is

$$\varepsilon_p = \sum_{j=1}^{N} \left({}_j e_p\right)^2 \qquad (10)$$

where N is the number of output elements, or the output vector size.

However, the stated square deviation is only a deviation calculated based on the presentation of a single input. Of course, the optimal situation would be if we were able to express the mean square deviation over all possible inputs of the network. These inputs, however, are not available; we only dispose of the training set containing p training pairs. The mean square deviation over all inputs contained in the training set can then be expressed as

$$\varepsilon = \mathrm{E}\{\varepsilon_p\} \ pro \forall \varepsilon_p \qquad (11)$$

When minimizing this function for example using the gradient method of the steepest descent, we would obtain the best possible network classification in terms of the mean square deviation over the entire training set. This mean square deviation is obviously a function of network weights **w**. Their change, therefore, implements a procedure to minimize the mean square deviation. For the change in neuron weights **w**, we can write

where t denotes the learning step and μ is a constant determining the speed of adaptation.

$$\overline{w}(t+1) = \overline{w}(t) - \mu \nabla \varepsilon \qquad (12)$$

Gradient ∇ε can be written as

$$\nabla \varepsilon = \frac{\partial \varepsilon}{\partial w}(t) \qquad (13)$$

In practice, however, the calculation of the gradient is unfortunately difficult even for small networks. Therefore, it is usually replaced by calculating the sequence of partial gradients where each sub-gradient obtained in one step of network learning approximates the gradient value over the whole training set.

$$\frac{\partial \varepsilon}{\partial w}(t) \approx \frac{\partial \varepsilon_p}{\partial w}(t) \qquad (14)$$

Recall now that the perceptron output layer is composed of individual neurons, each of which contributes to the overall perceptron error. Consider now a single neuron in the output layer.

Regarding the k0-th output network layer, the gradient value for the i-th weight of the w_i j-th neuron of this layer can be expressed according to the rule of composition differentiation

$$\frac{\partial \varepsilon_p}{\partial {}^{k_0}_{j} w_i} = \frac{\partial \varepsilon_p}{\partial \sigma\left({}^{k_0}_{j}\xi_p\right)} \frac{\partial \sigma\left({}^{k_0}_{j}\xi_p\right)}{\partial {}^{k_0}_{j}\xi_p} \frac{\partial {}^{k_0}_{j}\xi_p}{\partial {}^{k_0}_{j} w_i} \qquad (15)$$

If we analyse the formula by individual parts, then the third part of the right side of the formula is a differentiation of the inner potential of the j-th neuron with respect to i-th weight. The inner potential of the j-th neuron

is expressed as a sum of input values multiplied by the weights w. In this case, however, the neuron input values are represented by outputs y_p from the previous (k_0-1 layer); the result of differentiation can be written as follows

$$\frac{\partial\, {}^{k_0}_{j}\xi_p}{\partial\, {}^{k_0}_{j}w_i} = \frac{\partial \sum_{\forall i} {}^{k_0-1}_{i}y_p\, {}^{k_0}_{j}w_i}{\partial\, {}^{k_0}_{j}w_i} = {}^{k_0-1}_{i}y_p \quad (16)$$

The central part of the right side of the formula is a differentiation of the activation output function with respect to the neuron inner potential. Specific activation transfer function is not currently defined; therefore, leave the relationship without modification, only rewrite the differentiation

$$\frac{\partial \sigma\left({}^{k_0}_{j}\xi_p\right)}{\partial\, {}^{k_0}_{j}\xi_p} = \sigma'\left({}^{k_0}_{j}\xi_p\right) \quad (17)$$

After considering the expressions (9, 10), the introductory part of the formula (16) expressing error differentiation with respect to the neuron output y can be written as

$$\frac{\partial \varepsilon_p}{\partial \sigma\left({}^{k_0}_{j}\xi_p\right)} = \frac{\partial\left({}_{j}e_p\right)^2}{\partial \sigma\left({}^{k_0}_{j}\xi_p\right)} = \frac{\partial\left({}_{j}y_{dp} - {}_{j}y_p\right)^2}{\partial\, {}_{j}y_p} = -2\left({}_{j}y_{dp} - \sigma\left({}^{k_0}_{j}\xi_p\right)\right) = -2\left({}^{k_0}_{j}e_p\right) \quad (18)$$

After adjustment, we get the following relation

$$\frac{\partial \varepsilon_p}{\partial\, {}^{k_0}_{j}w_i} = \frac{\partial \varepsilon_p}{\partial \sigma\left({}^{k_0}_{j}\xi_p\right)} \frac{\partial \sigma\left({}^{k_0}_{j}\xi_p\right)}{\partial\, {}^{k_0}_{j}\xi_p} \frac{\partial\, {}^{k_0}_{j}\xi_p}{\partial\, {}^{k_0}_{j}w_i} = -2\left({}_{j}y_{dp} - \sigma\left({}^{k_0}_{j}\xi_p\right)\right)\sigma'\left({}^{k_0}_{j}\xi_p\right){}^{k_0-1}_{i}y_p = -2\left({}^{k_0}_{j}e_p\right)\sigma'\left({}^{k_0}_{j}\xi_p\right){}^{k_0-1}_{i}y_p \quad (19)$$

When substituting in (12) and using vector expression, we get the following relation for adjusting the weights of a neuron of the last layer

$${}^{k_0}_{j}\overline{w}(t+1) = {}^{k_0}_{j}\overline{w}(t) + 2\mu\left({}^{k_0}_{j}e_p\right)\sigma'\left({}^{k_0}_{j}\xi_p\right){}^{k_0-1}\overline{y}_p \quad (20)$$

In the relation, it remains to determine the value of the error e_p and possibly differentiation σ'. For neurons of the output layer, error e_p is already evident from the relation (9) and is equal to

$${}^{k}_{j}e_p = {}_{j}y_{dp} - {}^{k}_{j}y_p \quad (21)$$

Of course, the differentiation σ' depends on the particular activation output function of the neuron. Favourites are sigmoidal activation functions that are smooth and monotonic, and which have a simple expression of differentiation.

$$y = \frac{1}{1+e^{-\alpha}} \quad (22)$$

$$y' = \frac{1}{\left(1+e^{-\alpha}\right)^2} = \frac{1}{1+e^{-\alpha}}\,\frac{1+e^{-\alpha}+1}{1+e^{-\alpha}} = \frac{1}{1+e^{-\alpha}}\left(1-\frac{1}{1+e^{-\alpha}}\right) = y(1-y) \quad (23)$$

The choice of monotonic functions is more appropriate given the limitations of local minima of the error function. However, any differentiable functions can be used as activation transfer functions.

At this point, we are able to determine the error and perform adaptation of the output layer neurons using a formally derived relationship.

The procedure of deriving formulas for the output layer can be similarly used in case of neurons of the hidden layers:

$$\frac{\partial \varepsilon_p}{\partial\, {}^k_j w_i} = \frac{\partial \varepsilon_p}{\partial \sigma\left({}^k_j\xi_p\right)} \frac{\partial \sigma\left({}^k_j\xi_p\right)}{\partial\, {}^k_j\xi_p} \frac{\partial\, {}^k_j\xi_p}{\partial\, {}^k_j w_i} = \frac{\partial \varepsilon_p}{\partial \sigma\left({}^k_j\xi_p\right)} \sigma'\left({}^k_j\xi_p\right) {}^{k-1}_{\;i}y_p \tag{24}$$

where

$$\frac{\partial \varepsilon_p}{\partial \sigma\left({}^k_j\xi_p\right)} = \frac{\partial \varepsilon_p}{\partial\, {}^k_j y_p} = \left({}^k_j e_p\right) = \sum_{i=1}^{M_{(k+1)}} \frac{\partial \varepsilon_p}{\partial\, {}^{k+1}_{\;i}y_p} \frac{\partial\, {}^{k+1}_{\;i}y_p}{\partial\, {}^{k+1}_{\;i}\xi_p} \frac{\partial\, {}^{k+1}_{\;i}\xi_p}{\partial\, {}^k_j y_p} = -2\sum_{i=1}^{M_{(k+1)}} {}^{k+1}_{\;i}e_p\, \sigma'\left({}^{k+1}_{\;i}\xi_p\right) {}^{k+1}_{\;i}w_j \tag{25}$$

The above relations (18, 24) clearly show that the calculation of error function gradient for hidden neurons differs from the definition for neurons of the output layer only in the part for calculating error e_p at the neuron output.

Adjustment of weights for any network neuron can then be expressed as

$${}^k_j\overline{w}(t+1) = {}^k_j\overline{w}(t) - \mu\left({}^k_j e_p\right)\sigma'\left({}^k_j\xi_p\right) {}^{k-1}\overline{y}_p \tag{26}$$

where the following applies to neurons of the output layer:

$${}^k_j e_p = -2\left({}_j y_{dp} - \sigma\left({}^k_j\xi_p\right)\right) \tag{27}$$

The below relation applies to neurons of inner hidden layers:

$${}^k_j e_p = -2\sum_{i=1}^{M_{(k+1)}} \sigma'\left({}^{k+1}_{\;i}\xi_p\right) {}^{k+1}_{\;i}e_p\; {}^{k+1}_{\;i}w_j \tag{28}$$

The above derived expressions (26, 27, 28) are very often adjusted to expressions in the following forms:

$${}^k_j\overline{w}(t+1) = {}^k_j\overline{w}(t) + 2\mu\left({}^k_j e_p\right)\sigma'\left({}^k_j\xi_p\right) {}^{k-1}\overline{y}_p \tag{29}$$

$${}^k_j e_p = {}_j y_{dp} - \sigma\left({}^k_j\xi_p\right) \tag{30}$$

$${}^k_j e_p = \sum_{i=1}^{M_{(k+1)}} \sigma'\left({}^{k+1}_{\;i}\xi_p\right) {}^{k+1}_{\;i}e_p\; {}^{k+1}_{\;i}w_j \tag{31}$$

Here, the relationship with heuristically derived δ-rule for a single neuron is straightforward.

If we observe the formula (31), it is apparent that error ${}^{k}_{j}e_p$ at the output of the j-th neuron in the k-th hidden layer is obtained by summing the errors ${}^{k+1}_{i}e_p$ of all neurons at outputs from the following (k+1) layer, where this error ${}^{k+1}_{i}e_p$ is multiplied by the differentiation of the transfer output function σ' $\left({}^{k+1}_{i}\xi_p\right)$ and also by the value of weight ${}^{k}_{j}w$ of the synapse between the neurons of both layers.

The principle of error backpropagation algorithm consists in determining the error at the outputs of all neurons in the output layer, which can be easily determined according to the equation (30). Subsequently, this error is also used for adaptation of the hidden layer: the error from the output layer actually spreads through the network structure back from the output layer to the previous layer, where the output error is modified by differentiation of the transfer function for the output layer neuron, and consequently also by the weight leading to the output layer neuron from the neuron of the previous layer. Each neuron of hidden layer, thus, receives an error that corresponds to its contribution to the overall error of the output layer. In this way, the error is gradually propagated backward by the network structure, from the output to all previous layers. Therefore, this is called error backpropagation algorithm.

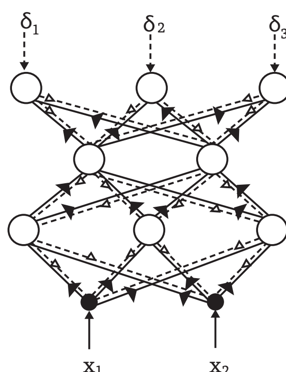


Fig. 5.15 Illustration of the error backpropagation algorithm

In points, the error backpropagation algorithm for a created multilayer perceptron can be summarized as follows:

1. Initialize the network weights
2. Choose the p-th pattern from the training set
3. Present the training pattern $\overline{x}_p$ to the network
4. Determine the network response $\overline{y}_p$ to $\overline{x}_p$
5. Determine the error at network output according to (30)
6. Determine the error in previous layers using error backpropagation to previous layers
7. Adjust weights locally according to (29)
8. Repeat the procedure from point 2 until the network performance is sufficient

5.5.6 Minimization of the error function using adaptation algorithm

Adaptation algorithm, for example the mentioned BP algorithm, performs minimization of the error function by searching for the minimum on the error hypersurface through the use of a gradient method. During the optimization process, each network weight can be seen as one dimension of the n-dimensional error space. For two weights, this hypersurface can have the form as shown in the Figure 5.16, where the horizontal axes represent the weight values and the vertical axis represents the sizes of the error function, which is a function of the network weights. The task of the adaptation process is to get as low as possible on this hypersurface, ideally into the global minimum. Nevertheless, there is also the risk of getting stuck in the local minima.

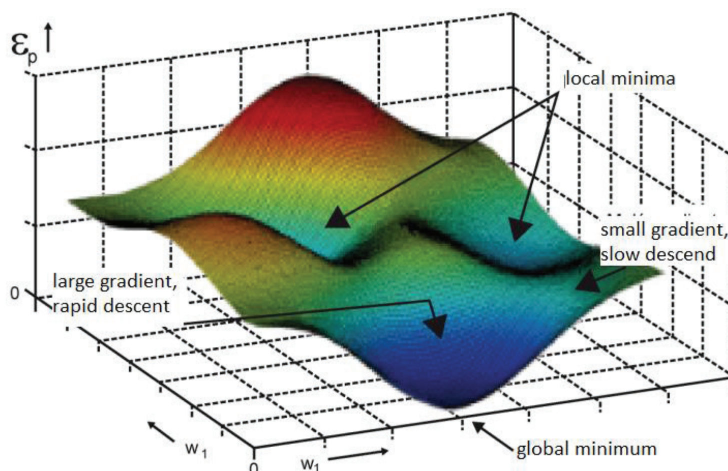


Fig. 5.16 Minimization of error function during adaptation

The risk of getting stuck in a local minimum can be reduced by the choice of initial parameters and the training set, and obviously also by modification of the adaptation algorithm.

Starting initialization of neuron weights

The starting initialization of weights defines a point on the error surface from where the algorithm starts when searching for the minimum. If the initial setting of weights defines the starting point on a sloping track aiming at a local minimum, it is obvious that the global minimum of the error function will not be found and the minimization will end in this local minimum. Unfortunately, we are not able to practically estimate the correct starting initialization; therefore, the starting initialization of network weights uses random numbers and is repeated if it fails.

Selection of the rate constant μ

The rate constant μ defines the size of the step with which the algorithm moves on the surface of error function. Its selection is also not straightforward. Too large step could lead to unstable movement across the surface and consequently to failure to find the minimum; on the contrary, small steps securely direct to the nearest minimum but the algorithm may converge slowly. A compromise solution is gradual reduction of the step size during the minimization process.

Complexity of the error function

Obviously, the shape of the error function surface depends not only on the network parameters (weights, transfer functions) but also on the input vectors. If the surface is too rugged with plenty of local minima, the activity of minimization algorithm is very difficult. Therefore, the network inputs are often preprocessed. The goal of preprocessing is to reduce the input dimension of vectors and especially suppress the irrelevant features of input vectors and emphasize the properties that are essential. Elimination of non-essential characteristics of the inputs reduces the incidence of local minima on the surface of error function and simplifies its shape. Unfortunately, we are often not able to decide about the substantiality or irrelevance of some characteristics of the inputs other than intuitively.

The method of presenting inputs of the training set

It is preferable to present inputs from the training set to the network randomly. Movement around the error surface is also random and consequently allows examination of a wider area of the error surface.

Modification of the BP algorithm

Modification of BP algorithm aims to reduce the risk of getting stuck in a local minimum. The modification of BP algorithm consists in the introduction of inertia η in the adaptation process; initially, we choose η close to 1 and reduce its value to zero during adaptation. In case the error is increased, it is recommended to omit inertia.

Relation (12) can be simplified to

$$\overline{w}(t+1) = \overline{w}(t) + \Delta\overline{w}(t) \quad (32)$$

When introducing inertia η into the relation, we get

$$\overline{w}(t+1) = \overline{w}(t) + (1-\eta)\Delta\overline{w}(t) + \eta\Delta\overline{w}(t-1) \quad (33)$$

For η = 0, the algorithm behaves like a pure BP algorithm.

5.5.7 Multilayer perceptron and overlearning syndrome

Multilayer perceptron is a robust universal approximator that represents a very versatile solution to the multilayer perceptron, due to the validity of the Kolmogorov's theorem. Its use is particularly practical in cases where we only dispose of examples of the inputs and outputs of a certain process. However, the great expressive power of the multilayer perceptron is redeemed by the relatively computational-intensive optimization of the network using the BP algorithm. Disadvantages also include the fact that the found transformation is hidden in the network structure; it cannot be directly interpreted.

Finding the global minimum of error function over the training set is also not always efficient. We may encounter the so-called effect of network overlearning when the network effectively minimizes the error function over the training set but at the expense of the generality of the found transformation for patterns outside the training set. The network is adapted to irrelevant details contained in the training set. Such overlearned network is then unable to generalize, as seen in decrease in network performance with regard to the test set.

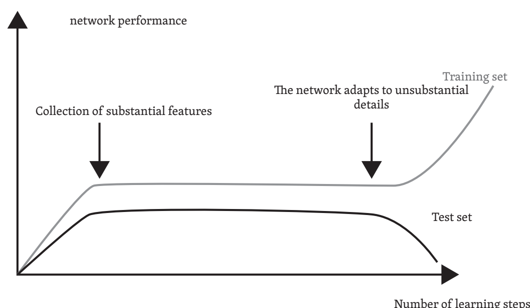


Fig. 5.17 Overlearning syndrome

Network overlearning can be prevented using several procedures. It is preferable not to seek achieving the absolute minimum of error function over the entire training set but to stop network learning at a certain earlier moment. This moment can often be estimated as sudden fall in the value of error function over the training set. An even more efficient procedure preventing overlearning consists in introducing not only the training and test sets but also the validation set that stops adaptation in time.

When designing a perceptron, if possible, we also try to choose a small number of neurons in the hidden layers; this will force the network to search for simpler and more general transformation. In order to ensure that the network does not focus on collecting irrelevant details of the training set, it is always possible to slightly distort the presented inputs, for example by random noise.

5.6 LITERATURE

[1] Šíma, J., Neruda, R.: Teoretické otázky neuronových sítí (Theoretical Questions of Neural Networks), MATFYZPRESS, 1996, ISBN 80-85863-18-9.

[2] Jan, J.: Číslicová filtrace, analýza a restaurace signal (Digital Filtering, Analysis and Restoration of Signals).University of Technology in Brno, VUTIUM, 2002. ISBN 80-214-1558-4.

[3] Kůrková V.: Kolmogorov's Theorem and Multilayer Neural Network. Neural Networks, Volume 5, Issue 3, Pages 501-506, 1992

6. NETWORKS WITH MUTUAL RELATIONS

6.1 BASIC INFORMATION

The following text forms part of the course "Artificial Intelligence" and is intended primarily for students of Mathematical Biology study programme. This chapter deals with the concept of neural networks with mutual relations (interrelationships), also called as associative networks. The topology of these networks significantly differs from that of feed-forward neural networks; biological analogy of behaviour of these networks mostly corresponds to recollecting (retrieving) memorized patterns. The basic theoretical model of Hopfield network is discussed in more detail.

6.2 LEARNING OUTCOMES

Mastering the learning text will enable students to:

- *Become familiar with the principles of associative neural networks.*
- *Understand the organization and active dynamics of Hopfield networks.*
- *Understand the introduction of the concepts of status, power and attractors of Hopfield networks.*
- *Set the network attractors using inequalities, including the procedures preventing false attractors.*
- *Understand the differences between stochastic neuron and its deterministic variant, and describe the Boltzmann machine as a stochastic extension of the Hopfield network.*
- *Understand the basic principle of heteroassociative networks on the example of bidirectional associative memory*

6.3 GENERAL CHARACTERISTICS OF ARTIFICIAL NEURAL NETWORKS WITH MUTUAL RELATIONS

Networks with mutual relations are also often called associative or attractor networks because the principle of their active dynamics consists in recollection or association of previously learned patterns – attractors. Due to their nature (the networks remember certain learned patterns), the models of associative networks are also known as associative memories.

Networks with interrelationships between neurons are characterized by feedbacks between neurons; the signals, therefore, do not propagate in one direction through individual layers as in the case of perceptrons. Connections between neurons can be complete (Hopfield network). It is also symmetrical, i.e. synapses are evaluated as single weight value, which is used for both directions of information propagation via the synapse. Most frequently, the network uses neurons with binary or bipolar characteristics that have the ability to express two states.

In active dynamics, the interconnection of neurons results in individual neurons being affected by neighbouring neurons with which they have a common bond. Active dynamics involves the gradual selection of one of the neurons (either deterministically or randomly) and adjustment of its status based on the influence of surrounding neurons. Due to changes in the state of neurons, the network gradually moves into a stable state preventing further changes; the network reaches the attractor.

In networks with mutual relations, we thus observe a difference compared to active dynamics of perceptrons. Perceptrons, in active dynamics, provide immediate response to the presented input by "computation" while the active dynamics of networks with mutual relations represents an iterative process aimed at finding one of the attractors. In comparison with perceptrons, the active dynamics has, therefore, an opposite character.

The iterative process of retrieving patterns and system transition to a stable state has again its analogy in the biological field where it corresponds to cognitive functions of the brain, such as object recognition. The analogy can also be found in the physical area; the Hopfield network originated as a model of spin glasses describing the initially random arrangement of directions of the magnetic moments of atoms in the crystalline grid, but which leads to an ordered stable state.

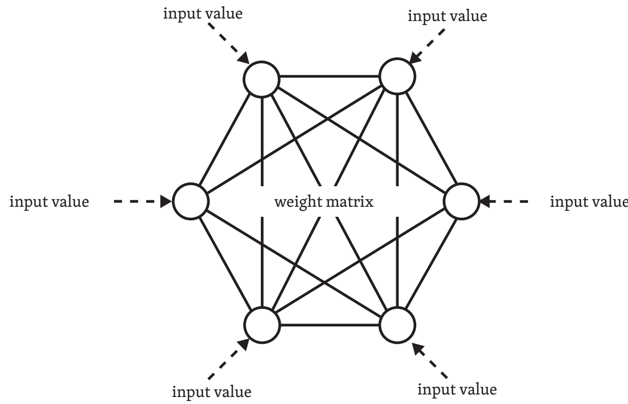


Fig. 6.1 General organizational dynamics of Hopfield networks

Associative neural networks can be divided into two classes, autoassociative and heteroassociative networks. Based on the presented input, autoassociative networks seek to retrieve the corresponding pattern in its complete form. An example might be the presentation of a distorted or incomplete picture of a person's face to the network. The result of network retrieving would then be an undamaged and complete picture previously stored in the network weights as one of the attractors.

In contrast, heteroassociative networks do not recall directly the presented pattern but complete this pattern with some related information of other nature. An example might be a network that identifies the given person and assigns previously memorized personal data to that person after a photograph of the person's face is presented.

6.4 HOPFIELD NETWORK

6.4.1 Organization dynamics

The Hopfield network topology consists of a fixed arrangement of N neurons as shown in Figure 6.1. Neurons are interconnected by bidirectional links to each other, i.e. inputs of each j-th neuron are connected with the outputs of all other i-th neurons. The weights are symmetric:

$$w_{ij} = w_{ji} \qquad (1)$$

and

$$w_{ii} = 0. \qquad (2)$$

The neuron output is represented by the internal state of the neuron as shown below. In the Hopfield network, there are usually neurons with binary (0,1) or bipolar (-1,1) characteristics. Each neuron contains an explicitly expressed threshold. Initial adjustment of the neuron state after presenting an input can be made by a one-time setup of the neuron state. It is good to realize what applies to the basic model of Hopfield networks with no hidden nodes: the number of elements of the input vector **x** is equal to the number of samples of the output vector **V**, and this is equal to the number of neurons in the network **N**.

6.4.1.1 States of the Hopfield network

An example of the Hopfield network with three nodes is shown in the Figure 6.2. Symmetrical weights are denoted as w, explicit thresholds of neurons are denoted as ϑ, and outputs, i.e. the states of neurons, are denoted as **V**.

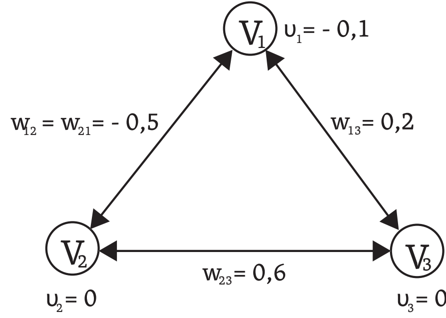


Fig. 6.2 Hopfield network with three nodes

Output V_i of each single i-th neuron is defined in accordance with the previously established definition of the single neuron as

$$V_i = y_i = \sigma(\xi_i) = \sigma(\sum_j w_{ij} V_j - \vartheta_i) \tag{3}$$

where ϑ_i represents the explicit threshold, and the activation output function $\sigma(\xi_i)$ has bipolar or binary character.

Variant, where the activation output function is defined as a continuous, e.g. sigmoidal, function,

$$\sigma(\xi_i) = \frac{1}{1 + e^{-\xi_i}} \tag{4}$$

is called the continuous Hopfield network, unlike its discrete implementation with binary or bipolar neurons.

In each discrete moment, the Hopfield network with N neurons is characterized by the output vector **V** of individual neurons. The output of each i-th neuron is also called the state of neuron V_i. Network state is then given by the vector of N elements where each element represents the state of one of the neurons. A particular importance relates to the network at steady state when this network state represents the end of active network dynamics, i.e. the searched attractor

6.4.2 Active dynamics of Hopfield networks

Suppose a Hopfield network with binary neurons with explicit threshold, with set values of weights of synapses between the neurons.

Initiation of active network dynamics corresponds to applying the input vector **x** to individual neurons. In this way, we set the initial state V_i of all neurons, i.e. also of the whole network. The state of each i-th neuron is given by the corresponding i-th value of the input vector element. In our case concerning binary neurons, the initial state of each neuron is, therefore, set to 0 or 1.

Then, one neuron is randomly selected and its status is updated according to relation (3).

After the state update, the state of a binary neuron can be changed to opposite or may remain unchanged. The change of its state depends on whether the neuron is sufficiently excited by its neighbours (inner potential exceeds the threshold of excited neuron) with which it is interconnected by rated connections. The strength and character

of the connection between neurons determines the degree of their mutual influence and may have inhibitory or excitatory effects. After changing the state of each neuron, the network switches to the next state also as a whole.

The next step of active dynamics is an option of another (randomly, i.e. also the same) neuron and the situation is repeated.

The algorithm thus gradually selects individual neurons and the network goes through various states until it reaches a steady state where transitions are no longer feasible. In other words, it is not possible to select any neuron that would change its status to the opposite.

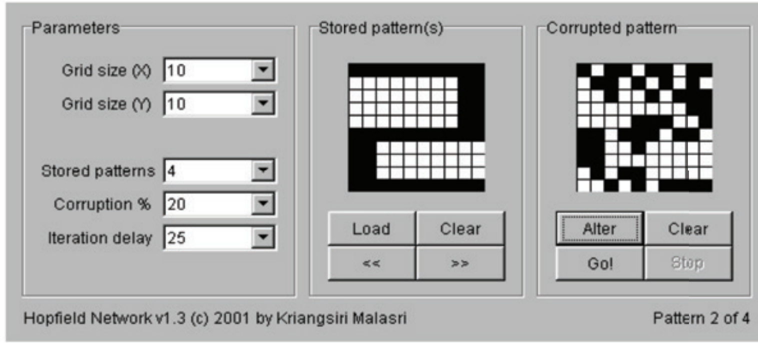


Fig. 6.3 Applet to demonstrate the capabilities of Hopfield networks. (http://www.cbu.edu/~pong/ai/hopfield/hopfield.html)

6.4.2.1 The energy function

For better understanding of the Hopfield network, we define the notion of energy of the i-th neuron.

$$E_i = -V_i \xi_i = -V_i \left(\sum_j w_{ij} V_j - \vartheta_i \right) \quad (5)$$

Similar to the state of the network, the total energy of the network at a given discrete time is again expressed by summation of partial energies of all neurons as

$$E = -\sum_i V_i \xi_i \quad (6)$$

Obviously, the energy function depends on the network state. In the process of active dynamics, i.e. when the state of any individual neuron of the network is changed, the energy function changes as well, and this change can be expressed as

$$\Delta E_i = -\Delta V_i \xi_i \quad (7)$$

It depends only on the change in the neuron state, ΔV_i, because the inner potential ξ_i of the neuron selected by the active dynamics remains constant in its given step.

In the binary neuron, there can be only two changes to the state V_i. Either from the state $V_i = 0$ to the state $V_i = 1$, or from the state $V_i = 1$ to the state $V_i = 0$. In these two cases, the energy change is as follows:

- *Change of V_i from 0 -> 1. Here applies that $\Delta V_i > 0$. However, this situation can occur only when $\xi_i > 0$. According to equation (7), the result is $\Delta E_i < 0$.*
- *Change of V_i from 1 -> 0. Here applies that $\Delta V_i < 0$. However, this situation can occur only when $\xi_i \leq 0$. According to equation (7), the result is $\Delta E_i \leq 0$.*

When activated, the energy of each neuron can therefore only decrease or remain the same.

The previous relationships provide important conclusions to the energy function. The total energy of the network, which is the sum of the energies of individual neurons, can only decrease or remain unchanged. The network states with low energy are more stable than the states with high energy. The network, thus, gradually moves into states with lower energy in individual (neuron-discrete) steps until it reaches a state where no neuron can switch to a lower energy state. The network stabilizes its state to that of low energy from which no more transition is feasible - it sticks in a local minimum of the energy function, in one of the attractors. This state represents the retrieved network pattern, i.e. the final network response to the presented input.

6.4.2.2 The state map of transitions

The process of active dynamics for a specific Hopfield network can be displayed using the state maps of transitions. Each node in this map contains information about the current state of the network as well as about the energy that corresponds to this state.

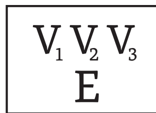


Fig. 6.4 A node in the state map of transitions and its content

In the diagram, the individual nodes can be arranged from nodes with the highest energy E to nodes with the lowest energy. Applying an input to the network corresponds to adjusting the network to a state equal to this input. When assuming a network where individual neurons are chosen separately and randomly with equal probability, the Hopfield network and its map of transitions can be visualized as follows:

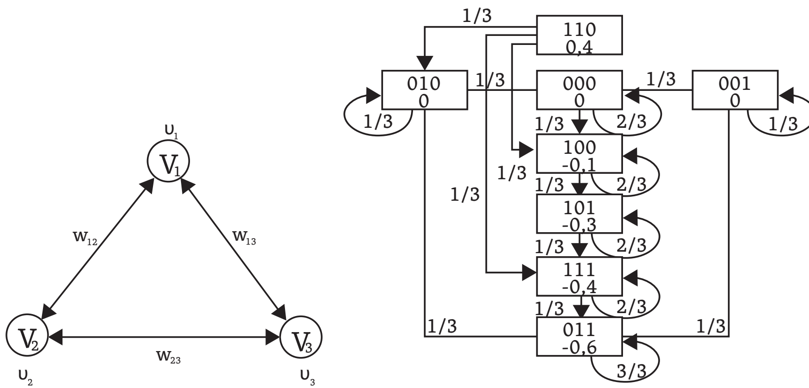


Fig. 6.5 Hopfield network with three nodes and the corresponding map of transitions

The edges between individual nodes correspond to the probability of executing different transitions. In our case, the selection of each of the three neurons is equally likely. After a certain number of steps, the network gets into the state expressed as 011, which corresponds to the minimum value of the network energy, and is even a global attractor (in this case). In this state, if any neuron is selected / activated, it does not change its status, and the network remains stable.

6.4.3 Adaptation dynamics

Adaptation dynamics is based on the use of the training set that contains pmax of patterns of size N. However, these patterns do not have any assigned expected output as in the case of feed-forward networks; given the concept of autoassociative networks, each pattern is one of the desired network attractors.

$$M = \{x^1, x^2, \ldots x^{pmax}\} = \{V^1, V^2, \ldots V^{pmax}\} \qquad (8)$$

At a fixed network topology, the aim of adaptation dynamics is to find such a setting of network parameters so that the presented patterns represent the local minima of energy function. Typically, this only relates to adaptation of weights connecting individual neurons; the threshold values are considered to be zero. In the phase of active dynamics, all presented patterns are attracted to the "nearest" memorized ("remembered") pattern. It is clear that the network of a given topology is able to "remember" only a limited number of patterns. The probability p(s) of network stable state can be expressed as

$$p(s) = \frac{1}{2} - \frac{1}{\sqrt{\pi}} \int_0^{\sqrt{\frac{N}{2P}}} e^{-x^2} dx \qquad (9)$$

Hopfield experimentally deduced that the number of binary patterns P, that the network is able to remember and then also retrieve, is approximately equal to

$$P \approx 0{,}15N \qquad (10)$$

where N is the number of neurons.

According to the mentioned literature, this number is still overvalued; for $P \leq 0{,}138N$, the required attractors correspond to the local minima of energy function.

For bipolar neurons, the approximate formula for the number of patterns is as follows:

$$P \approx \frac{N}{2\log_2 N} \qquad (11)$$

For stable storing of patterns, the Hopfield network indeed requires a relatively high number of nodes. Network memory can be expanded by adding the so-called hidden nodes incorporated into the network structure; however, they do not provide any input or output.

6.4.3.1 Hebb rule

The Hebb rule can be used to adjust weights between neurons, both in the case of binary and bipolar neuron outputs. Different patterns of the training set are gradually applied to the network. The weights w_{ij} are then positively reinforced with consistent positive activation of neurons to the presented input. In case of inconsistent activation of neighbouring neurons interconnected by bonds with weight w, their mutual relations are weakened. The resulting weight then represents the difference between the consistent and inconsistent states (outputs V_i) of neurons for all inputs from the training set.

Specific value of the weight w_{ij} between neurons can be set for binary neurons according to the relation

$$w_{ij} = \sum_{pmax} [x_i - 1][x_j - 1] = \sum_{pmax} [V_i - 1][V_j - 1], \text{ for } i \neq j \qquad (12)$$

For bipolar neurons, the relationship is simpler.

$$w_{ij} = \sum_{pmax} x\, x_j = \sum_{pmax} V_i V_j \text{ for } i \neq j \qquad (13)$$

The individual patterns are stored in the network indirectly in the form of relations between neurons and their states evaluated by weights. This method of learning, however, may lead to phantoms, false attractors, which represent the local minima of energy function but do not represent any of the attractors presented and desired.

6.4.3.2 Adjustment by inequalities

The procedure for setting the weights by inequalities will be explained on the implementation of a specific example. It is a procedure after which the network has the desired attractors.

Again, consider a Hopfield network with three interconnected nodes.

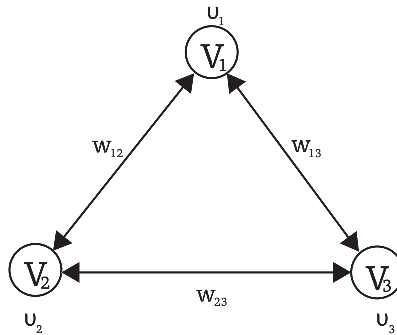


Fig. 6.6 Example of the Hopfield network

We demand that the attractors are elements A and B from set M = {A=010, B=111}. Therefore, we are looking for such adjustment of weights w and thresholds ϑ for these attractors, which would ensure that the network is stable and maintains its state. For the desired attractor A = 010, we can build a system of inequalities (see Table 6.1).

$V_1 = 0$	$\xi_1 \leq 0 \rightarrow w_{12} V_2 + w_{13} V_3 - \vartheta_1 \leq 0$	after substituting for V: $w_{12} - \vartheta_1 \leq 0$
$V_2 = 1$	$\xi_2 > 0 \rightarrow w_{12} V_1 + w_{23} V_3 - \vartheta_2 \leq 0$	after substituting for V: $\vartheta_2 < 0$
$V_3 = 0$	$\xi_3 \leq 0 \rightarrow w_{13} V_1 + w_{23} V_2 - \vartheta_3 \leq 0$	after substituting for V: $w_{23} - \vartheta_3 \leq 0$

***Tab. 6.1** The system of inequalities for the attractor A*

Similarly for the attractor B = 111 (Table 6.2).

$V_1 = 1$	$\xi_1 \geq 0 \rightarrow w_{12} V_2 + w_{13} V_3 - \vartheta_1 > 0$	after substituting for V: $w_{12} + w_{13} - \vartheta_1 > 0$
$V_2 = 1$	$\xi_2 \geq 0 \rightarrow w_{12} V_1 + w_{23} V_3 - \vartheta_2 > 0$	after substituting for V: $w_{12} + w_{23} - \vartheta_2 > 0$
$V_3 = 1$	$\xi_3 \geq 0 \rightarrow w_{13} V_1 + w_{23} V_2 - \vartheta_3 > 0$	after substituting for V: $w_{13} + w_{23} - \vartheta_3 > 0$

***Tab. 6.2** The system of inequalities for the attractor B*

These relations represent a system of six linear inequations with six unknowns. The system has infinite solutions satisfying the given conditions, for example w_{12} = 0.5; w_{13} = 0.4; w_{23} = 0.1; and $\vartheta1$ = 0.7; and $\vartheta2$ = -0.2; and $\vartheta3$ = 0.4.

Nevertheless, this procedure does not prevent false attractors representing unwanted local minima of energy function, bounded by higher values of the energy function in all directions (i.e. possible state changes). Active network dynamics then can constantly get stuck in these unwanted minima. The solution, which avoids false attractors, may be completion of the inequalities for each state which must not be an attractor. In practice, however, this is hardly solvable. An alternative is the network learning through the use of delta rule that can be summarized into the following steps:

1. Select the set of desired attractors M = $\{\mathbf{x}^1, \mathbf{x}^2, ... \mathbf{x}^{pmax}\}$
2. Apply one of the patterns **x** to the network and compute the activation functions and neuron outputs (states)
3. If the states resulting from the activation function coincide with the desired (presented) states, do nothing.
4. Otherwise, modify the weights of excited inputs and threshold of the actual neuron by selected ±Δ depending on the neuron response
   a. If V_i = 0 and it should be 1 (x_i = 1)=> reduce the threshold and strengthen excited weights by Δ
   b. If V_i = 1 and it should be 0 (x_i = 0)=> increase the threshold and strengthen excited weights by Δ

6.5 BOLTZMANN MACHINE

6.5.1 Organization dynamics

Organization dynamics of the Boltzmann machine is exactly the same as in Hopfield networks, most commonly Hopfield networks with hidden neurons. The difference is in the type of used neurons. They do not behave deterministically as in the case of Hopfield networks; the change of the neuron state to opposite state (0 -> 1 and 1 -> 0) depends not only on the value of the activation function but also on the quantity, called the network temperature T ≥ 0. This temperature usually changes during the course of the network operation and is responsible for the stochastic behaviour of neurons.

The basis of Boltzmann machine is a stochastic neuron for which we know the probability P of its being in the given state based on the value of neuron inner potential. The function expressing the dependence of the probability of state change is often a sigmoidal function modified by parameter T. Depending on the temperature, the stochastic neuron may change its state with certain probability, regardless of the value of the inner potential ξ_i.

$$P = \sigma(\xi_i) = \frac{1}{1 + e^{\frac{-\xi_i}{1+T}}} \qquad (14)$$

If T = 0, we get a standard sigmoidal function for the probability of the state change; it is therefore highly likely that the neuron will behave according to the value of its inner potential ξ_i. Furthermore, the equation (14) indicates that the probability of selecting both states equals with increasing value of parameter T, regardless of the value of the inner potential ξ_i , and that the neuron behaves completely randomly for T -> ∞; both states are equally likely.

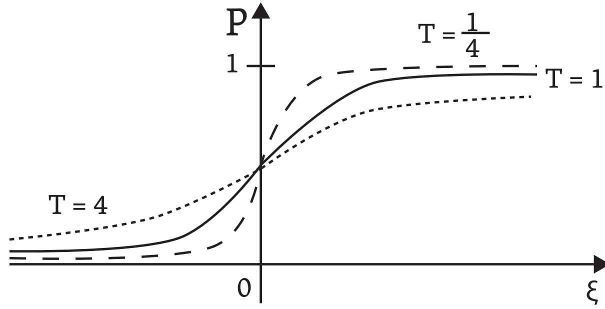


Fig. 6.7 The course of the probability activation function depending on T and , according to [1]

6.5.2 Active dynamics

Active dynamics of the Boltzmann machine is again analogous to the Hopfield network. During the process of active dynamics, one of the neurons is again (randomly) selected in a discrete step t, and its real inner potential ξ_i is evaluated in accordance with (3).

$$\xi_i(t) = \sum_j w_{ij} y_j(t) - \vartheta_i \tag{15}$$

However, the change in the neuron state does not depend only on the value of this potential, i.e. its comparison with zero in case of binary neurons; the state change is determined stochastically and the neuron is active / 1 or passive / 0 with the following probabilities:

$$P\{y_i(t+1) = 1\} = \sigma(\xi_i(t)) \tag{16}$$

or

$$P\{y_i(t+1) = 0\} = 1 - P\{y_i(t+1) = 1\} = \sigma(-\xi_i(t)). \tag{17}$$

Consider the network as shown in Figure 6.8. The probabilities of network transitions between states change depending on the temperature T. For example, let's focus on the probability of transition from the state 011 to other states.

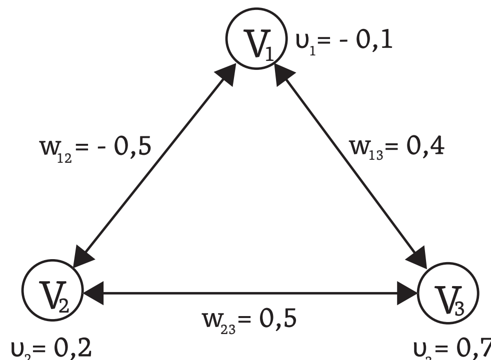


Fig. 6.8 Example of a network

For T = 0, we get a Hopfield network with deterministic behaviour. The probability of the states never achievable in the Hopfield network increases with increasing temperature T.

The process of network active dynamics, working with the temperature T, is known as simulated annealing. Network activity takes place as follows: First, we set a high temperature T. If the temperature is high, the individual neurons and thus the network behave very non-deterministically and switch between various states rather randomly. After some period of network activity, the network state stabilizes. Then we slightly reduce the temperature T and the process repeats. In this way, we proceed until receiving the value T = 0, inclusive. More likely, the stochastic behaviour of neurons enables overcoming the local minima of energy function, which may constitute false attractors. For the Hopfield network, any sticking to the local minimum is final, unlike the Boltzmann machine. This procedure provides higher probability of reaching the global minimum of the energy function and thus the desired attractor. In this respect, the behaviour of Boltzmann machine is more advantageous than the behaviour of Hopfield networks.

6.6 BIDIRECTIONAL ASSOCIATIVE MEMORY (BAM)

Bidirectional associative memory was introduced in 1987 by Bart Kosko. BAM is a representative of heteroassociative networks; after presenting the input vector, these networks retrieve another vector, associated with the vector presented. The resulting vector typically contains data that describes the presented vector in more detail or attach some information related to this vector.

6.6.1 Organization dynamics

It is a network that consists of two layers of completely interconnected neurons. Each of these layers is simultaneously the input and output layers. After its presentation to the network, the input vector **x** associates the output vector y on the first layer; on the contrary, the vector **x** is associated on the inputs of the second layer after presenting the vector **y**.

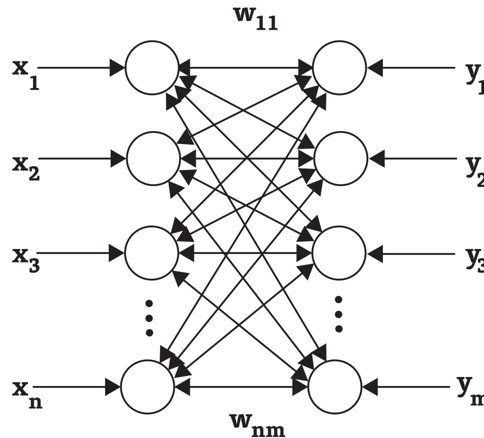


Fig. 6.9 BAM topology according to [5]

6.6.2 Adaptation dynamics

Consider a bipolar implementation of neurons, i.e. neurons with outputs (-1,1). Adjustment of the network weights is implemented based on the training set represented by couples of related pairs of vectors **x** and **y**. The maximum number of pairs of associations that can be remembered by the network is min (m, n).

$$M = \{[x^1,y^1], [x^2,y^2]\dots. [x^{pmax},y^{pmax}]\} \qquad (18)$$

Weights can be set according to the relation

$$w_{ij} = \sum_{p=1}^{p\max} w_{ij} = \sum_{p=1}^{p\max} x_i^p y_j^p \qquad (19)$$

which can be modified to matrix notation as follows:

$$W = \sum_{p=1}^{p\max} (X^p)^T Y^p \qquad (20)$$

The entire network configuration is then represented by a matrix of weights **W**.
Suppose we have a training set M consisting of two pairs.
M = {[$\mathbf{x}^1$ = (1;-1; 1; -1; 1; -1), y^1 = (1; 1; -1; -1)], [$\mathbf{x}^2$ = (1; 1; 1; -1; -1; -1), y^2 = (1; -1; 1; -1)] }

The matrix of weights W can be calculated according to equation (20); then we get the resulting matrix of weights.

$$W = \begin{bmatrix} 2 & 0 & 0 & -2 \\ 0 & -2 & 2 & 0 \\ 2 & 0 & 0 & -2 \\ -2 & 0 & 0 & 2 \\ 0 & 2 & -2 & 0 \\ -2 & 0 & 0 & 2 \end{bmatrix}$$

6.6.3 Active dynamics

Active network dynamics takes place at discrete steps. The network operates symmetrically; after presenting the vector to the input layer, the neurons of this layer calculate their response (synchronously, in a single step) and this signal, multiplied by the weights, propagates to neurons of the opposite layer, which again synchronously calculate their outputs and thus create the output vector.

In the next step, the role of layers rotates and the calculated output vector is propagated by the network structure back to the opposite, initially input layer. This mutual signal exchange takes place until the network state stabilizes, i.e. until the moment when the outputs on either network layer do not change. After reaching this steady state, the input layer contains the vector from the training set that most closely matches the presented input vector, and the output layer contains the vector associated with this pattern in the training set.

For bipolar BAM, its response at the steady shape can be easily expressed according to the relations

$$Y = \mathrm{sgn}(XW) \qquad (21)$$

$$X = \mathrm{sgn}\,(YW^T) \qquad (22)$$

For example, after presenting the input x1=(1;-1; 1; -1; 1; -1) and when using weights matrix W from the previous paragraph, we get Y = sgn(XW) = sgn(8, 4, -4, -8) => (+,+,-,-) => (1,1,-1,-1).

6.7 LITERATURE

[1] Šíma, J., Neruda, R.: Teoretické otázky neuronových sítí (Theoretical Questions of Neural Networks), MATFYZPRESS, 1996, ISBN 80-85863-18-9.

[2] Jan, J.: Číslicová filtrace, analýza a restaurace signal (Digital Filtering, Analysis and Restoration of Signals). University of Technology in Brno, VUTIUM, 2002. ISBN 80-214-1558-4.

[3] Volná, E.: Neuronové sítě 1. (Neural Networks 1). Lecture notes. Ostrava University in Ostrava, Ostrava, 2008.

[4] Kosko B.: Bidirectional Associative Memories, IEEE Trans. on Systems, Man, and Cybernetics, 1987.

[5] http://moon.felk.cvut.cz/~pjv/Jak/_neur/n566/bam.html

[6] V. Kvasnička, L. Beňušková, J. Pospíchal, I. Farkaš, P. Tiňo, and A. Kráľ. Úvod do teórie neurónových sietí (Introduction to the Theory of Neural Networks), IRIS, Bratislava, 1997, ISBN 80-88778-30-1.

[7] http://www.cbu.edu/~pong/ai/hopfield/hopfield.html

[8] http://en.wikipedia.org/wiki/Bidirectional_associative_memory

7. COMPETITIVE NETWORKS

7.1 BASIC INFORMATION

The following text forms part of the course "Artificial Intelligence" and is intended primarily for students of Mathematical Biology study programme. This chapter deals with competitive networks that are characterized by the existence of a competitive, interconnected layer of neurons. The neurons of this network are activated by applying the input; in the steady state, after mutual competition, the "strongest" neuron usually wins. The Kohonen model of self-organizing maps will be discussed in detail.

7.2 LEARNING OUTCOMES

Mastering the learning text will enable students to:

- *Describe the principle of competitive networks and demonstrate this principle on the dynamics of MAXNET network*
- *Become familiar with the dynamics of Hamming network and demonstrate its active dynamics on a model example*
- *Understand the organization and active dynamics of Kohonen self-organizing maps*
- *Formally define the adaptation dynamics of Kohonen maps, to describe the pitfalls of Kohonen learning and carry out its expansion by LVQ1, LVQ2, LVQ3 algorithms*

7.3 A SIMPLE COMPETITIVE NETWORK MAXNET

The basic concept of competitive networks will be illustrated on the example of the simplest network, the network for maximum selection, MAXNET. As in the case of other competitive networks, an essential component of this network is the so-called competitive layer in which the individual interconnected neurons are excited by applying an input signal – an input classified vector. In the next phase, which already takes place in individual discrete time steps k, the state of the entire layer is always recursively and synchronously updated. Finally, in the steady state of the network, only one neuron in this competitive layer is active (has non-zero output). This winning neuron is a representative of the cluster assigned to the presented input. In the case of MAXNET network, the winner is always a neuron with the highest initial input value; due to the equality of weights between neurons of the competitive layer, it is the neuron with the highest inner potential in step 0. The competitive layer thus corresponds to the block for maximum selection.

7.3.1 Organization dynamics

MAXNET consists of the competitive layer with M classical binary neurons, as introduced in the chapter on perceptrons. In this layer, neurons are connected to each other.

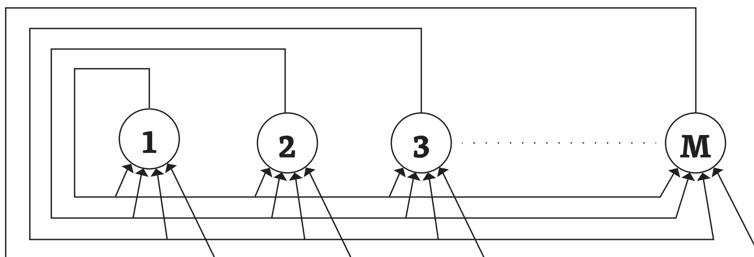


Fig. 7.1 Competitive layer of MAXNET network

Thresholds of all neurons are zero

$$\vartheta_i = 0 \ for \forall i \quad (1)$$

Weight value of the feedback bond of each neuron is

$$w_{ij} = 1 \ for \ i = j \qquad (2)$$

All bonds between neurons are set to the same value (-ε)

$$w_{ij} = -\varepsilon \ for \ i \neq j \qquad (3)$$

The value ε is selected in the interval from zero to 1/M, where M is the number of neurons in the competi tive layer

$$0 < \varepsilon < \frac{1}{M} \qquad (4)$$

These weights remain constant. The number of neurons of the competitive layer is usually chosen according to the expected number of classes into which we want to classify.

After applying the signal (input) to the competitive layer, the network is complemented by an input layer fully connected to the competitive layer by weighted bonds that are subject to adaptation in the learning phase.

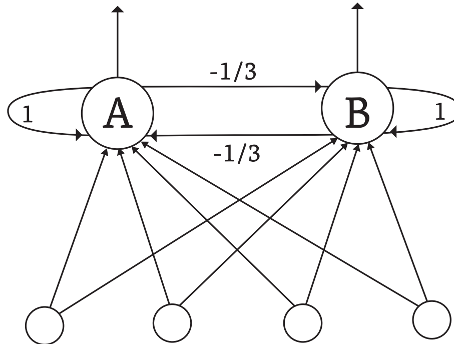


Fig. 7.2 Simple competitive network MAXNET with two neurons in the competitive layer

7.3.2 Adaptation dynamics

The weights of the competitive layer remain constant throughout the adaptation and active dynamics. Adaptation relates only to weights which value bonds from the input layer to the competitive layer. Learning takes place on the basis of inputs from the training set.

$$M = \{\mathbf{x}^1, \mathbf{x}^2, ... \mathbf{x}^{pmax}\} \qquad (5)$$

Typical steps of competitive learning in MAXNET networks are as follows:

- *Adjustment of fixed values of weights in the competitive layer*
- *Adjustment of the input layer weights to random values or according to priori knowledge of the expected clusters (typical representative). A single weight between the i-th element of the input vector and the j-th neuron of the competitive layer is usually chosen in the interval (0,1), so that* $\sum_{\forall i} w_{ij} = 1$

- *Presentation of input vector* $\mathbf{x}$ *to the network*
- *Calculation of primary outputs of neurons of the competitive layer*
- *Competition of neurons during single steps in the competitive layer (input is disconnected)*
- *Determination of the winning neuron, the representative with non-zero output*
- *Adjustment of the weights of the winning neuron* $w_{ij} = w_{ij} + \Delta w_{ij}$ *by* Δw_{ij} *(6) and their normalization to* $\sum_{\forall i} w_{ij} = 1$

$$\Delta w_{ij} = \eta x_i \frac{1}{M} - \eta w_{ij}, \forall i \tag{7}$$

- *Repetition of the process for the next input vector* $\mathbf{x}$*. During network learning, the coefficient* η *usually gradually decreases. This procedure speeds up network convergence.*

Consider the network in Figure 7.2. Consider the initial setting of weights of neurons A and B as w_A=(0.2;0.2;0.3;0.3) and w_B=(0.2;0.3;0.3;0.2). For example, before starting lateral inhibition and for vector $\mathbf{x}$ = (1; 1; 0; 0), the output of neuron A will be y_A = 0.4 and the output of neuron B will be y_B = 0.5. After completion of the competition, the winning neuron will be neuron B ($y_A < y_B$); therefore, we will adjust the vector of its weight w_B according to (7). Select η =1/2. The number of neurons M = 2.

After presenting the first input, the weights will be

w_{1B}=0.2+0.5*0.5-0.5*0.2=0.35

w_{2B}=0.3+0.5*0.5-0.5*0.3=0.4

w_{3B}=0.3+0-0.5*0.3=0.15

w_{4B}=0.2+0-0.5*0.2=0.1

After adjusting the weights in this manner, the network would classify into the cluster represented by neuron A for cluster of inputs (0;0;0;1), (0;0;1;0), (0;0;1;1), (0;1;1;1), (1;0;0;1), (1;0;1;1), and into the class represented by neuron B for cluster of inputs (0;1;0;0), (0;1;1;0), (1;0;0;0), (1;1;0;0), (1;1;0;1), (1;1;1;0). Other vectors would not be classified into any of the classes.

Similarly, we would proceed in the adaptation of weights also for other presented inputs. An important feature is that the mentioned algorithm does not explicitly determine the class into which the inputs should be included; therefore, faults in network classification are not and cannot be assessed. The network forms clusters independently, only on the basis of presented inputs. It is, therefore, unsupervised learning.

7.3.3 Active dynamics

In the process of active dynamics, individual inputs (input vectors) are presented to the network. After implementing the calculation, the network responds by activating a single neuron of the competitive layer, the representative of the cluster. The input is then included in the cluster represented by the selected representative. In case of MAXNET networks, it is the neuron which was most excited after applying the input, i.e. the neuron with the maximum value of the activation function. In other words, it is the neuron whose weights maximally correlate with the input vector presented.

Consider again the network from Figure 7.2 and vectors of input weights to neurons A and B w_A=(0.2;0.2;0.3;0.3) and w_B=(0.2;0.3;0.3;0.2).

This network selects representative A for inputs (0;0;0;1), (0;0;1;1), (1;0;0;1), (1;0;1;1), and representative B for inputs (0;1;0;0), (1;1;1;0), (1;1;0;0), (1;1;1;0). Other vectors will not be classified into any of the classes because none of the neurons will win (both neurons are excited equally and the competitive layer mutually voids their effect).

The following should be noted: in the category of self-organizing networks with interrelationships, the active and adaptation dynamics may not always be strictly separated by phases; due to the self-learning network algorithm, adaptation can theoretically run again and again based on continuously presented new inputs.

7.4 HAMMING NETWORK

7.4.1 Organizational dynamics

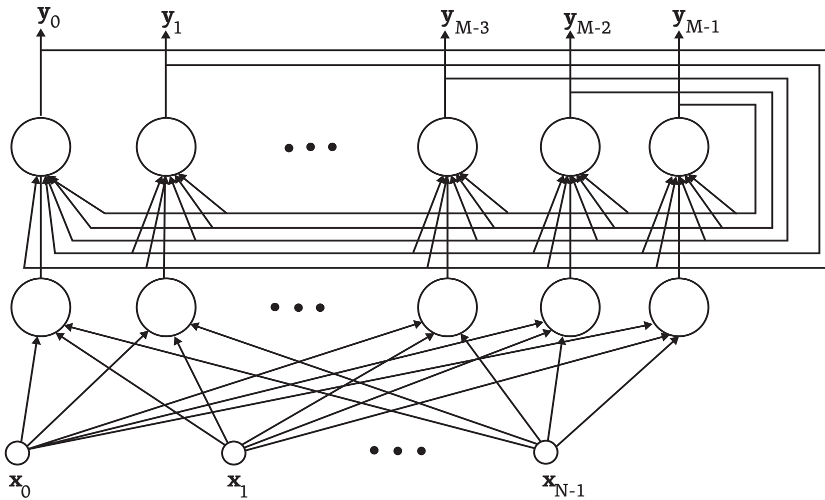


Fig. 7.3 Hamming network

The organization dynamics of the Hamming network is evident from the Figure 7.3. It consists of three layers; the first layer is the input layer and transmits N inputs from the vector x to the second layer, Hamming layer with M neurons, via full interconnection. This layer (each neuron in this layer) counts complement of the Hamming distance of each neuron (the vector of its weights) from the presented input. The outputs of the second layer are directly transmitted (with their weight = 1) to the last, competitive layer, selecting the maximum from M neurons.

7.4.2 Adaptation dynamics

Further, assume N-dimensional bipolar input vectors, i.e. vectors with N elements acquiring the values 1 or (-1). The training set contains M pattern bipolar vectors **x**.

$$\mathbf{M} = \{\mathbf{x}^1, \mathbf{x}^2, \ldots \mathbf{X}^M\} \qquad (8)$$

The parameters of the Hamming layer, i.e. weights and thresholds of i-th neuron, are set by direct calculation based on the representatives of the training set.

Thresholds of i-th neuron are set to

$$\vartheta_i = \frac{N}{2} \, pro \forall i \qquad (9)$$

The vector of weights wi of i-th neuron is set according to values of i-th representative **x**, in compliance with the relation

$$\vec{w}_i = \frac{1}{2}\vec{x}_i \ pro \forall i \tag{10}$$

Each neuron of the Hamming layer thus corresponds to one input from the set of patterns, it is its representative.

The competitive layer is adjusted according to the rules described in the chapter on MAXNET.

7.4.3 Active dynamics

The active dynamics of the Hamming network is obvious. After presenting the input vector, the Hamming layer calculates complement of the Hamming distance of the input vector from all neurons, representatives. The highest output value appertains to the neuron that is closest to the input in terms of the Hamming distance. The last layer then ensures that this neuron will be selected as the winning representative after the competition is over.

For example, assume the following set of patterns

$M = \{x^1(1;-1;-1;1;1), x^2(-1;1;-1;1;-1), x^3(1;-1;1;-1;1)\}$

Suppose the input vector **x** = (1;1;1;-1;-1).

The weights and thresholds of neurons in the Hamming layer are set according to equations (9), (10). For the response of i-th neuron of the Hamming layer, we can write

$y_1 = ((x^1/2)*x)-5/2 = ½(x^1*x-5) = ½(-3-5) = -4$
$y_2 = ((x^2/2)*x)-5/2 = ½(x^2*x-5) = ½(-1-5) = -3$
$y_3 = ((x^3/2)*x)-5/2 = ½(x^3*x-5) = ½(1-5) = -2$

The winning neuron is the third neuron with i = 3, which is closest to the vector **x** in terms of the Hamming distance.

The Hamming network is actually a model whose parameters serve to store a specific set of representatives; explanation of its action regarding the interpretation of outputs is obvious, in contrast to the multilayer feed-forward neural networks, for example. Network parameters can be easily set by calculation, without the need for iterations. On the other hand, these advantageous properties are redeemed by a smaller ability of the network to generalize. Regarding practical implementations working with multi-dimensional input vectors, a real-working Hamming network would lead to a very large set of patterns, i.e. a large number of neurons in the network.

7.5 SELF-ORGANIZING MAPS

7.5.1 A simple self-organizing map – organization and active dynamics

In case of self-organizing maps, the neural network operates only as a notion; usually, it is actually implemented using simpler data structures such as arrays (vectors) or matrices.

The organizational structure of simple self-organizing maps is identical to the network described in the section devoted to MAXNET. It is a two-layer network where the input layer is completely interconnected with the competitive layer. The network usually implements transformation from the space of real numbers into the space of binary values.

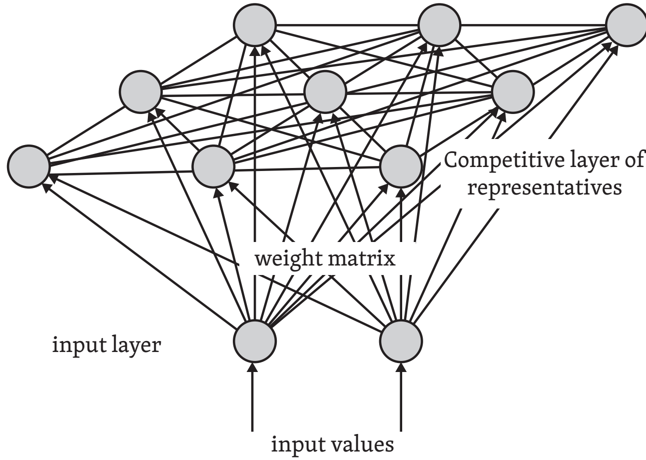


Fig. 7.4 General topology of a self-organizing map

As in the case of Hamming networks, the weight vectors between neurons of the input layer and competition layer constitute positions of representatives in the input space. In active dynamics, similar to the Hamming network, the network calculates the distance of presented input from the position of all representatives given by the neuron weights. The competitive layer then again ensures the win of the strongest representative.

7.5.2 Adaptation dynamic of simple self-organizing maps – Kohonen learning

The basic method of adaptive dynamics is unsupervised learning. Therefore, the network itself organizes their weights, only on the basis of inputs presented from the training set.

$$\mathbf{M} = \{\mathbf{x}^1, \mathbf{x}^2, \dots \mathbf{x}^{pmax}\} \qquad (11)$$

During its adaptation, the network creates representatives for clusters found by the network in the training set and subsequently classifies the presented inputs into these clusters in active dynamics.

The process of adaptation takes place using the method of Kohonen learning. In this adaptation process, the weights are initially set to random values. Subsequently, to the input in the k-th step, one of the inputs from the training set is applied. The network determines the winning i-th neuron. Weights of the winning neuron are modified according to the following relation

$$\mathbf{w}_i(k+1) = \mathbf{w}_i(k) + \eta\,(\mathbf{x}(k) - \mathbf{w}_i(k)).$$

where η is a parameter from the interval (0,1>. (12)

Parameter η determines the degree of vector **w** moving toward vector **x** and the speed of learning. At the beginning of learning, its value is usually close to 1, and is gradually reduced to 0 in the process of learning. The idea of adaptation is as follows: through weight modification, we shift the position of the winning representative in the input space closer to the currently classified input. In this way, we gradually browse the entire training set until the moment when the positions of representatives are changed minimally.

However, this kind of adaptation also has its substantial disadvantages. It is oversensitive to the appropriate initial initialization of neuron weights so that it covers the portion of the input space where we assume classified inputs, i.e. the sought clusters. Imagine a situation where we want to identify clusters in the classical two-dimensional Euclidean space. Assume that the initial representatives, after the starting random initialization of the neuron weights, are distributed unevenly and are concentrated only in one region of the input space. Further-

more, assume that the training set, in which we want to identify clusters, comprises vectors from the same Euclidean space but located in an area other than the initialized weights. The area of representatives given by the initial setting of weights and the area of inputs from the training are thus distant from each other in the space. After presenting the first input from the training set, the nearest representative will win. Its weights are adjusted according to (12) and this representative is so moved much closer to the area of inputs from the training set. But, what will happen after presenting another input from the same training set? Since the input vectors are relatively close to each other, the winner will be the same representative, the one that was corrected in the last step. In this way, the algorithm further continues and cyclically modifies only one representative. Others remain unchanged. In practice, this basic algorithm often fails. The illustrated problem can be addressed by the extended self-organizing neural networks called Kohonen maps.

7.5.3 Kohonen self-organizing maps – organization and active dynamics

The organization dynamics of Kohonen maps is completely the same as in the case of the simple self-organizing map shown in Figure 7.4. The difference consists in the competitive layer. Its organization is again completely consistent with the previous cases of competitive networks; in addition, however, the neurons (detectors) are logically arranged in geometric structure. Most often, it is a line or a grid, but it may be basically arbitrary. Usually, its size has much lower dimension than the original input space.

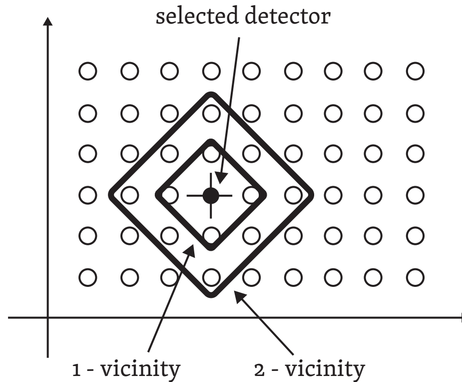


Fig. 7.5 Definition of the detector neighbourhood according to [2]

The geometric arrangement of detectors allows us to define the term “1 - n vicinity” of each detector. In the process of active dynamics, this geometric arrangement does not apply; after the input is presented, active dynamics is entirely consistent with the simple self-organizing map defined in the previous chapter.

7.5.4 Kohonen self-organizing maps – adaptation dynamics

The difference of Kohonen maps from simple self-organizing maps consists in the process of adaptation dynamics that takes into account the geometrical arrangement of detectors. The adaptation is again based on the presentation of inputs from the training set. After presenting the vector, the winning neuron is again determined, as in the previous case. Afterwards, however, not only the weights of the winning neuron are adjusted but also the weights that are located in the geometric vicinity of the winner, according to equation (12).

The usual procedure is as follows: in the initial phase of adaptation, a broad vicinity of the winner is modified, usually involving the entire network. The modified vicinity then gradually decreases. The value of parameter η, determining the speed of learning, usually decreases together with the vicinity. This procedure attracts representatives to the places with the highest concentration of inputs from the training set. This leads to the minimization of classification error over the training set in terms of minimizing the mean square deviation of input vectors from the representatives.

Parameter η may not be constant, not in terms of the distance from the winning representative, and its value may fall with the distance from the winner, e.g. according to Gaussian function with its centre in the winning neuron. This procedure also implicates that the representatives, which are close to each other in terms of topological arrangement (e.g. on a grid), are close to each other after adaptation also in the original input space.

7.5.5 Kohonen self-organizing maps – supervised learning

The mentioned procedures of adaptation of Kohonen maps took place as unsupervised (without any teacher). Distribution of inputs to individual clusters was given only by the network adaptation dynamics. Nevertheless, we often want to give some meaning to the found clusters, to name the found clusters, i.e. to classify the presented inputs into a pre-known number of named categories. For this purpose, we use the learning algorithms of supervised Kohonen maps, known as LVQ1, LVQ2 and LVQ3 (LVQ = Linear Vector Quantization), according to [1].

The first learning phase is identical for all three algorithms and consists in applying the standard algorithm for adaptation of Kohonen maps, as presented in the previous chapter. In this phase, only input vectors **x** of the training set are used; the corresponding output categories **y** are not used.

$$\mathbf{M} = \{[\mathbf{x}^1,\mathbf{y}_d^1], [\mathbf{x}^2,\mathbf{y}_d^2] \ldots . [\mathbf{x}^{pmax},\mathbf{y}_d^{pmax}]\} \qquad (13)$$

After network learning, we identify how the network classifies across the whole training set. For each input vector **x** from the training set, we determine the representative assigned to the given input **x**. At the same time, we dispose of the correct category y_d for each input. For each representative, we can then determine a set of named categories y_d and the frequency of including inputs **x** for each of them. Based on data on the frequency of assigning categories to individual representatives, we will name the representatives by the category which they most often represented over the training set.

The last step is the correction of weights, i.e. the positions of representatives, by one of the mentioned LVQ algorithms.

LVQ1 algorithm works as follows: we present inputs from the training set to the network and determine the winner. Then we evaluate whether the classification of the presented input is correct, i.e. whether the correct cluster is determined. Subsequently, we adjust weights only in that representative, according to the relationship identical to (12) for correctly classified input vectors

$$\mathbf{w}_i(k+1) = \mathbf{w}_i(k) + \eta\,(\mathbf{x}(k) - \mathbf{w}_i(k))$$

and

$$\mathbf{w}_i(k+1) = \mathbf{w}_i(k) - (\mathbf{x}(k) - \mathbf{w}_i(k)) \qquad (14)$$

for incorrectly classified inputs.

In case of correct classification, the representative approaches the classified pattern; otherwise, the representative shifts away from the erroneously classified pattern. LVQ1 only corrects the positions found for the representatives, the value η should be very small, between 0.01–0.02 according to [1], and should gradually approach zero. This procedure leads to emptying the border areas between adjacent clusters. The determined boundary approximates the Bayesian decision boundary and lies in the middle of the join between representatives of different classes.

LVQ2 algorithm is used only for a few (thousand) iterations; it has been experimentally found that it initially improves the placement of decision boundaries but begins to deteriorate the classification after a large number of iterations. We apply the input and determine two neurons closest to the input, i.e. representatives. It must be true that one of them classifies correctly and the other incorrectly. Furthermore, it must true that the input is not too close to any of the two representatives, is located “in the middle between them” in a window of usual wid-

th of about 10-30% of the neuron distance. When these conditions are met, we adjust both neurons pursuant to the usual relations, i.e. we bring the given neuron of the pair of neurons closer to the presented input (12), (14) or put it off, according to the classification correctness.

LVQ3 algorithm is a stable extension of LVQ2. It works in the same way as LVQ2 algorithm, i.e. it corrects the position of the two closest representatives according to LVQ2, (12) (14). In addition, it also adjusts the weights of a correctly classifying representative from the selected pair according to

$$\mathbf{w}_i(k+1) = \mathbf{w}_i(k) + \varepsilon \eta (\mathbf{x}(k) - \mathbf{w}_i(k)). \qquad (15)$$

In this manner, it moves it closer to a correctly classified input. Parameter ε remains constant during the adaptation and is chosen in the range of 0.1 – 0.5 [1].

7.6 LITERATURE

[1] Šíma, J., Neruda, R.: Teoretické otázky neuronových sítí (Theoretical Questions of Neural Networks), MATFYZPRESS, 1996, ISBN 80-85863-18-9.

[2] Jan, J.: Číslicová filtrace, analýza a restaurace signal (Digital Filtering, Analysis and Restoration of Signals). University of Technology in Brno, VUTIUM, 2002. ISBN 80-214-1558-4.

[3] Volná, E.: Neuronové sítě 1. (Neural Networks 1). Lecture notes. Ostrava University in Ostrava, Ostrava, 2008.

[4] Kvasnička, V., Beňušková, L., Pospíchal, J., Farkaš, I., Tiňo, P., Kráľ A.: Úvod do teórie neurónových sietí (Introduction to the Theory of Neural Networks), IRIS, Bratislava, 1997, ISBN 80-88778-30-1.

8. INTRODUCTION TO GENETIC ALGORITHMS (GA)

8.1 BASIC INFORMATION

The following text is part of the course "Artificial Intelligence" and is intended primarily for students of Mathematical Biology study programme. This chapter deals with the basic concepts of genetic algorithms (GA) and explains the procedures generally used in their implementation.

8.2 LEARNING OUTCOMES

Mastering the learning text will enable students to:

- *Understand the basic concepts such as the individual, population, objective function, the fitness of an individual*
- *Understand the operations used in the implementation of genetic algorithms – selection, crossover, mutation*
- *Become familiar with some selection methods and compare them with each other*
- *Understand the relationship between GA and algorithms used in NN*

8.3 BASIC CONCEPTS OF GENETIC ALGORITHMS

8.3.1 Introduction

Genetic algorithms are not very clearly defined and belong to a wide class of algorithms of artificial intelligence, inspired by biology. GA usually serve for solving optimization tasks. Like NN, they are applied in areas where finding a solution to a defined problem is analytically difficult or unavailable. We can note, very briefly and simply, that the properties of all higher living organisms are encoded in the genetic information stored in a set of cellular structures – nuclear chromosomes. Chromosomes contain genetic information expressed in the DNA chain. For this chain, we can identify individual sections, logical functional units, which are called genes. Again, with some simplification, we can say that specific forms of genes (alleles) and their combinations are responsible for properties that a specific individual will have. According to the Darwin's theory of evolution, surviving and reproducing individuals are those whose properties are best adapted to their environment (i.e. individuals having a suitable phenotype). Over many generations, the evolutionary pressure leads to the selection of the most successful population in the given environment.

GA form a relatively novel family of algorithms used approximately since the 1960s and utilizes the above mentioned knowledge of biology. Specifically, as results from their name, it is an area of genetics and evolutionary theory.

8.3.2 Basic concepts

The basic GA concepts include the notion of the **individual**, α. In GA, in compliance with the biological analogy, this term is described by a set of "chromosomes" and the specific content of individual genes. In GA, a particular representation of an individual can be expressed by a single (haploid individual) string (array) of numeric values where a specific position in the string corresponds to a specific gene and the value in this array then indirectly determines the characteristics of the individual. The numerical quantities are most commonly implemented as binary values due to the simplicity of working with them; generally, however, they may take real values. In GA, when using binary values $a_1..a_k$, any individual α can be described as a string

$$P = \{\alpha_1, \alpha_2, \ldots, \alpha_N\} \quad (1)$$

Population is then formed by a set of N individuals

$$P = \{\alpha_1, \alpha_2, \ldots, \alpha_N\} \quad (2)$$

The actual population of individuals is called **generation**.

The objective function $f(\alpha)$ performs mapping from the binary genotype space into the space of real numbers. Each individual is thus reflected by the value of its objective function that models the environment. Its value is analogous to the phenotype and performs transformation.

$$\{0,1\}^k \rightarrow R \tag{3}$$

It holds that the lower the objective function value, the more optimal is the given individual in the context of the specific problem being solved. An individual with a minimum value of the objective function thus represents the best currently known solution to the problem. The objective function is, therefore, a criterion that estimates the closeness of the given individual to the optimal solution (individual), which we try to minimize. In NN, the analogous minimization process is finding the optimal values of network weights during the adaptation dynamics – minimization of the network error function.

Each individual in the population can, therefore, be evaluated according to its potential, fitness for survival against others, according to its advantages with respect to the environment. Fitness of an individual is the representation F(α)

$$\forall \alpha_n \in P : f(\alpha_1) \leq f(\alpha_2) \Rightarrow F(\alpha_1) \geq F(\alpha_2) \geq 0 \tag{4}$$

For practical reasons, it is appropriate to introduce normalization so that the fitness value was in the interval (0,1); therefore, we introduce normalized fitness where the following applies:

$$F'(\alpha_i) = \frac{F(\alpha_i)}{\sum_{\alpha_j \in P} F(\alpha_j)} \tag{5}$$

$$\sum_{\alpha \in P} F'(\alpha) = 1 \tag{6}$$

The easiest way to quantify an individual's fitness is to perform linear interpolation of individual fitness points from the objective function f -> F

$$F(\alpha) = \frac{1}{f_{min} - f_{max}} \left[(1-\varepsilon) f(\alpha) + f_{min}\varepsilon - f_{max} \right]$$

where ε is a value close to zero. (7)

The solution of the problem using GA seems to be, generally, as follows: we dispose of (or generate) an initial population of individuals representing several elements of the solution space. In order to explore the broadest possible field of solutions, we must be able to create new individuals; therefore, we allow the current population to reproduce. In this way, we generate a new population representing other elements of the solution space. However, we are mostly unable to generate and evaluate the entire space of possible solutions due to its large size. Therefore, we do not enable all individuals to reproduce but only those who are most promising in terms of individual fitness. Other individuals, with lower fitness value, usually do not reproduce and die. This leads to

creating a new population represented by the descendants of the previous population. The total size of the population remains generally constant.

The solution or the group of possible solutions to a task using GA is, therefore, represented by the search for such individuals who have optimum properties from the perspective of the given problem. It is obvious that the GA must be able to generate new individuals and choose the best ones of them. For this purpose, we use operators (crossover, mutation, selection) that are described in detail in the following subsections.

8.3.3 Selection operator

Selection operator is used to select individuals in the population for further reproduction. There are many algorithms that usually work with individual fitness. The main methods of selecting individuals are shown below.

8.3.3.1 The roulette selection

The probability of selecting i-th individual from a population of N individuals is

$$p(i) = \frac{F_i}{\sum_N F_j} \tag{8}$$

Every individual obtains a part of an imaginary roulette corresponding to the value of its fitness function.

The expected value for the selection of an individual is equal to the proportion of the given individual quality toward the average individual quality in the population (arithmetic mean)

$$EV(i) = \frac{F_i}{\frac{\sum_N F_j}{N}} = \frac{N * F_i}{\sum_N F_j} \tag{9}$$

The expected value for the selection of an individual is thus proportional to its qualities expressed by the fitness function toward the rest of the population. In the selection of individuals, from the very beginning, this algorithm strongly prefers more elite individuals. The algorithm quickly converges to a solution; due to its preference for elite individuals, however, it can also easily get stuck in a local minimum during its "direct" path through the space of possible solutions.

8.3.3.2 Marshalling method

Individuals in the population are arranged in ascending order according to the value of their fitness functions.

$$F(\alpha_1) \le F(\alpha_i) \le F(\alpha_N) \tag{10}$$

Expected values of other N individuals in the population are determined according to the following equation

$$EV(i) = EV(1) + (EV(N) - EV(1))\frac{i-1}{N-1} \tag{11}$$

The expected selection value thus depends only on the individual's position in the implemented order (arrangement). The aim is to avoid the previous case (roulette selection) with a high probability of selecting only a few

elite individuals, which may lead to insufficient exploration of the solution space. Conversely, a disadvantage may be that the mentioned mapping, only on the row position, results in losing information about the actual quality of the individual.

8.3.3.3 Linear and exponential selection

These selection methods require a population of N individuals being ordered from the worst to the best individual according to fitness values. The probability of selecting i-th individual then depends only on the position (index i) of the individual in the given order. This dependence is either linear

$$p(i) = \frac{1}{N}\left(n^{-} + \left(n^{+} - n^{-}\right)\right)\frac{i-1}{N-1}$$

, where n+ is an arbitrary selected evaluation of the best individual, typically in the range of 1 < n+ <2, n- is the evaluation of the worst ranking individual n- = 2- n+ in the population. (12)

The value of n^+ may be higher than 2, and even more prefer elite individuals, but it is not recommended for calculation, since the worst individuals would receive a negative value of n^-. Other individuals in the population receives evaluation in interval (n^+ - n^-)based on a linear dependence of its index position i in the population.
The ratio n^-/N or n^+/N is the probability of selecting the worst or the best individual.

Exponential dependence of probability of selecting i-th individual can be determined according to the following equation

$$p(i) = \frac{c^{N-i}}{\sum_{N} c^{N-j}} \tag{13}$$

The basis of the exponent c is selected from the interval (0,1), usually close to 1. Exponential function value is growing up to a value equal to 1 for N-th individual. Changes in this parameter c can increase or decrease the selectivity of the algorithm and thus create different population pressure on the selection of individuals. The selection algorithm with exponential selection is one of the algorithms often used in practice.

8.3.3.4 Boltzmann selection

The Boltzmann selection is an analogy with simulated annealing, which we came across in the Boltzmann machine. Therefore, it works with a variable called temperature T. The variable T alters over time, i.e. during different generations of populations of individuals. Initially, the temperature is again set as high and gradually decreases to zero. This gradually increases the selection pressure; at low temperature, i.e. at the end of the algorithm, only the best individuals reproduce. The sense of this algorithm is the same as in the case of Boltzmann machine, i.e. to reduce the likelihood of getting stuck in a local minimum and achieve the greater likelihood of reaching the global minimum, here in the sense of the objective function.

$$EV(i) = \frac{N e^{\frac{F_i}{T}}}{\sum_{N} F_j} \tag{14}$$

8.3.3.5 Other types of selection

Previous selection methods assume that a new population is created in each new generation and the parent population becomes extinct. However, this may not always be the case. At least, let's briefly mention some of the other methods. **Tournament selection** method has several variants; for example, we randomly select two individuals and the better of them advances to reproduction. It is, therefore, not necessary to evaluate and compare the characteristics of all individuals simultaneously. Another selection option is **elitism** that preserves the best individuals. The most successful individuals of a given generation do not die but are transferred to the following generation together with the descendants. Another method of selection is **trimming**; the population is divided into two or more parts, and only individuals from the more successful group advance to reproduction. An additional alternative is **random selection** in which individuals are chosen completely at random.

In the current generation, the selection operator is able to find individuals who should reproduce and create a new population. Therefore, we dispose of individuals representing parents. Crossover operator generates their descendants. The most common case is a single-point crossover of two parents where the parents interchange the remaining part of the string from a certain random gene (position in the string), (1110011|1011) X (1111111|1000) => (1110011|1000) and (1111111|1011).

Furthermore, we usually decide whether both descendants advance to the next generation or just the better one (for example). The crossover operator exists in a number of variants; there can be a multi-point crossover among more than two parents, or some parents may advance without reproduction directly to the next generation, as mentioned in the elitism selection.

8.3.5 Mutation

As in nature, the mutation operator introduces a random element to the reproduction and thus allows escaping from the local minimum if the mutation proves to be advantageous. Mutation is usually implemented as a single -point mutation when the content of one gene, with a certain probability, is randomly changed, (1110011|1011) X (1111111|1000) => (1110011|1000) and (1111111|1**1**11).

In fact, the mutation represents an attempt to step aside where the mutated individual probably occurs outside the area of the solution space explored so far. This allows us to examine new opportunities and the solution space is explored to a greater extent with regard to the effort of finding the global minimum.

8.4 GA TASKS

8.4.1 GA summary

Typical steps of GA can be summarized as follows:

1. Initialization of the population, coding and setting of parameters of individuals. The first generation.
2. Evaluation of the population, definition of criteria for individuals. Fitness calculation.
3. Selection of individuals for reproduction.
4. Crossover of parental individuals, creating descendants. Selection of descendants for the next generation.
5. Mutations of some individuals in the new population.
6. Evaluation of individuals of the new population, one of them can be a solution. Yes = end.
7. If not, proceed to step 2 for the new population.

Genetic algorithms are suitable for optimization problems that are difficult to solve by mathematical methods by other means. Their implementation is relatively simple but they often require a lot of computing time and memory. The great implementation advantage of GA is the possibility of their easy parallelization on multiple computing units. Genetic algorithms always reach a solution; the question is whether the solution will be good enough, there is no guarantee of optimality. We can observe a certain difference from classical algorithms using gradient methods in the search for solutions, basically proceeding in one, probably the most correct direction,

according to the gradient value of the criterial function. On the contrary, GA explore the wider area of solutions, do not require continuous objective criterial function and the mutation operator allows us to explore also solutions completely outside the current local minimum. Usually, GA quickly converge to finding a specific sub-optimal solution; however, further refinement to find the optimal solution in the search area is slow. Therefore, we often use the so-called hybrid algorithms; first, a GA is applied, and the optimal solution is then searched locally using the gradient method.

8.4.2 Relation between GA and NN

GA can be also successfully used for optimizing neural networks. GA are mostly applied in the process of organization and adaptation dynamics where they optimize topology or weights of neural networks.

In the phase of NN organization dynamics, the genetic algorithms may be used in finding the optimal number of neurons, their parameters, arrangement into layers and connections. Usually, the algorithms are based either on a minimal network structure and gradually add more neurons, or the procedure is reversed and the initial complex network is gradually simplified while maintaining the performance parameters. In the phase of organization dynamics, the implementation of GA is computationally very demanding with regard to the necessity of checking a large number of complex individuals and requires considerable system resources. Here, each individual is represented by a complex topological structure of NN, and also its examination and fitness evaluation is not trivial.

In the stage of NN adaptation dynamics, the network topology is fixed; the GA finds an optimal setting for values of weights of individual neurons. The weights can be presented as real values but they are usually coded as binary values, which leads to simpler GA implementation. In the given length of the chain describing an individual, the disadvantage is that we are able to capture only a limited number of values of weights, which may not be sufficient. Chain lengthening can refine their values but their extension significantly slows the given GA.

GA can also be used simultaneously, in the phase of adaptation as well as organization dynamics. Despite their demandingness in the phase of organization dynamics, the use of GA appears to provide benefits because they are able to find a network with the simplest and most optimum structure while maintaining the performance parameters. For the adaptation of network weights, in addition to GA, we can possibly use the gradient backpropagation algorithm that is much less computationally demanding and brings mostly satisfactory results despite the danger of getting stuck in local minima.

8.5 LITERATURE

[1] Šíma, J., Neruda, R.: Teoretické otázky neuronových sítí (Theoretical Questions of Neural Networks), MATFYZPRESS, 1996, ISBN 80-85863-18-9.

[2] Volná, E.: Neuronové sítě 2 (Neural Networks 2). Lecture notes. Ostrava University in Ostrava, Ostrava, 2008.

[3] Kvasnička, V., Beňušková, L., Pospíchal, J., Farkaš, I., Tiňo, P., Kráľ A.: Úvod do teórie neurónových sietí (Introduction to the Theory of Neural Networks), IRIS, Bratislava, 1997, ISBN 80-88778-30-1.

[4] Pavlovič, J.: Multikriteriální hybridní evoluční algoritmy pro výběr a optimalizaci dekontaminačních technologií (Multi-criteria Hybrid Evolutionary Algorithms for the Selection and Optimization of Decontamination Technologies), 2008. Dissertation, Masaryk University, http://is.muni.cz/th/4035/fi_r/.

Published by Masaryk University, Brno 2015
1st edition

ISBN 978-80-210-7728-7

muni
PRESS