Chapter 1
Fundamentals of Intellectual Technologies

Intellectual technologies which are used to do the tasks of identification and decision making in this book represent a combination of three independent theories:

- *of fuzzy sets* - as a means of natural language expressions and logic evidence formalization;
- *of neural nets* - artificial analogs of the human brain simulating the capability to learn;
- *of genetic algorithms* - as a means of optimal decision synthesis from a multiplicity of initial variants on which the operations of crossing, mutation and selection are performed.

The concept of the linguistic variable underlies natural language expressions formalization [1, 2]. According to Zadeh [1], such a variable whose values are words or sentences of the natural language, that is the qualitative terms, is called the linguistic variable. Using the notion of membership function, each of the terms estimating a linguistic variable can be formulated in the form of a fuzzy set defined on a corresponding universal set [2]. Fuzzy logic apparatus does not contain learning mechanisms. That is why the results of fuzzy logic evidence strongly depend on the membership functions type used to formalize fuzzy terms: “small”, “large”, “cool”, “hot” and alike.

The main feature of neural networks is their learning ability. This is realized by special algorithms among which the back-propagation algorithm is the most popular [3, 4]. There is no need for prior information about the structure of the sought functional dependence to train the neural network. Only the training data in the form of experimental “input – output” pairs are needed, and the price for it is the fact that a trained neural network – a graph with weighted edges – doesn’t yield to semantic interpretation.

Optimization is the most important stage in solving identification problems [5 – 7]. A task of nonlinear optimization can be solved by various methods among which the gradient descent [8] is the most universal. However, when there is a great number of input variables the gradient descent method requires finding the minimum from various initial points that substantially increases computer time expenses. Genetic algorithms represent the powerful apparatus of optimal decision synthesis [9, 10]. These algorithms are analogues of random search [8], which is carried out simultaneously from various initial points, cutting the time of search for optimal solutions.

A.P. Rotshtein et al.: Fuzzy Evidence in Identif., Forecast. and Diagn., STUDFUZZ 275, pp. 1–37.
springerlink.com

1.1 Fuzzy Sets

This section is written on the basis of the works [1, 2, 11, 12]. The additional information relative to fuzzy sets and decision making under uncertainty can be found in the works [13 – 21].

The concept of a *set*, and set theory, are powerful tools in mathematics. Unfortunately, a *sin qua non* condition underlying set theory, i.e. that an element can either belong to a set or not, is often not applicable in real life where many vague terms as "large profit", "high pressure", "moderate temperature", "reliable tools", "safe conditions", etc. are extensively used. Unfortunately, such imprecise descriptions cannot be adequately handled by conventional mathematical tools.

If we wish to maintain the very meaning of imprecise (vague) terms, a crisp differentiation between elements (e.g., pressure values) that are either high or not high may be artificial, and some values may be perceived high to some extent, not fully high and not fully not high.

An attempt to develop a formal apparatus to involve a partial membership in a set was undertaken in the mid-1960's by Zadeh [1]. He introduced the concept of a *fuzzy set* as a collection of objects which might "belong" to it to a degree, from 1 for full belongingness to 0 for full nonbelongingness, through all intermediate values. This was done by employing the concept of a *membership function*, assigning to each element of a *universe of discourse* a number from the unit interval to indicate the *intensity (grade) of belongingness*. The concept of a membership function was evidently an extension of that of a *characteristic function* of a conventional set assigning to the universe of discourse either 0 (nonbelongingness) or 1 (belongingness). Then, basic properties and operations on fuzzy sets were defined by Zadeh (and later by his numerous followers) being essentially extensions (in the above spirit) of their conventional counterparts.

Since its inception, fuzzy sets theory has experienced an unprecedented growth of interest in virtually all fields of science and technology.

1.1.1 Fundamentals of Fuzzy Set Theory

Suppose that $X = \{x\}$ is a *universe of discourse*, i.e. the set of all possible (feasible, relevant, ...) elements to be considered with respect to a fuzzy (vague) concept (property). Then a *fuzzy subset* (or a *fuzzy set*, for short) A in X is defined as a set of ordered pairs $\{(x, \mu^A(x))\}$, where $x \in X$ and $\mu^A : X \to [0, 1]$ is the *membership function* of A; $\mu^A(x) \in [0, 1]$ is the *grade of membership* of x in A, from 0 for full nonbelongingness to 1 for full belongingness, through all intermediate values. In some contexts it may be expedient to view the grade of membership of a particular element as its degree of compatibility with the (vague) concept represented by the fuzzy set. Notice that the degrees of membership are clearly subjective.

Many authors denote $\mu^A(x)$ by $A(x)$. Moreover, a fuzzy set is often equated with its membership function so that both A and $\mu^A(x)$ are often used interchangeably.

Notice that if $[0, 1]$ is replaced by $\{0, 1\}$, this definition coincides with the characteristic function based description of an ordinary (nonfuzzy) set. Moreover, the original Zadeh's unit interval is chosen for simplicity, and a similar role may be played by an ordered set, e.g., a lattice.

It is convenient to denote a fuzzy set defined in a finite universe of discourse, say A in $X = \{x_1, x_2, ..., x_n\}$ as

$$A = \mu^A(x_1)/x_1 + \mu^A(x_2)/x_2 + ... + \mu^A(x_n)/x_n = \sum_{i=1}^{n} \mu^A(x_i)/x_i ,$$

where "$\mu^A(x_i)/x_i$" (called a singleton) is a pair "grade of membership – element" and "+" is meant in the set-theoretic sense.

Example 1.1. If $X = \{1, 2, ..., 10\}$, then a fuzzy set "large number" may be given as

$$A = \text{" large number"} = 0.2/6 + 0.5/7 + 0.8/8 + 1/9 + 1/10$$

to be meant as: 9 and 10 are surely (to degree 1) "large numbers", 8 is a "large number" to degree 0.8, etc. and 1,2,...,5 are surely not "large numbers". Notice that the above degrees of membership are subjective (a "large number" is a subjective concept!) and context-dependent, and - by convention - the singletons with $\mu^A(\bullet) = 0$ are omitted.

In practice it is usually convenient to use a piecewise linear representation of the membership function of a fuzzy set as shown in Fig. 1.1 since only two values, $\overline{a}$ and $\underline{a}$, are needed.

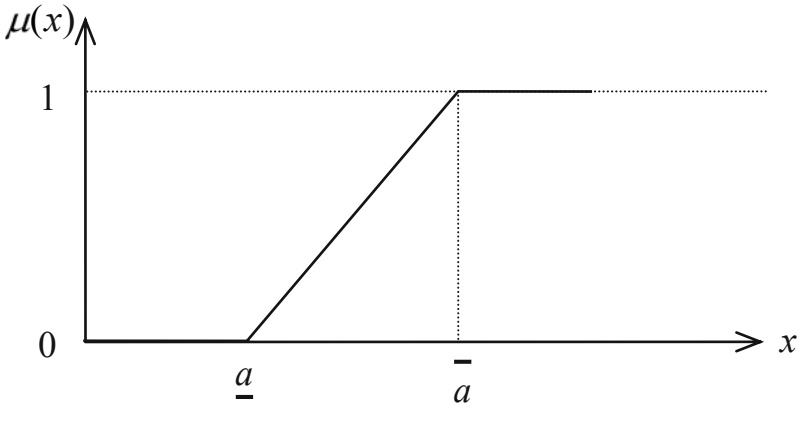


Fig. 1.1. Membership function of a fuzzy set

1.1.2 Basic Properties of Fuzzy Sets

A fuzzy set A in X is *empty*, $A = \varnothing$, if and only if $\mu^A(x) = 0$, $\forall x \in X$.

Two fuzzy sets A, B in X are *equal*, $A = B$, if and only if $\mu^A(x) = \mu^B(x)$, $\forall x \in X$.

A fuzzy set A in X is *included* in (is a *subset* of) a fuzzy set B in X, $A \subseteq B$, if and only if $\mu^A(x) \le \mu^B(x)$, $\forall x \in X$.

Example 1.2. Suppose $X = \{1, 2, 3\}$ and $A = 0.3/1 + 0.5/2 + 1/3$ and $B = 0.4/1 + 0.6/2 + 1/3$; then $A \subseteq B$.

An important concept is the *cardinality* of a fuzzy set. If $X = \{x_1, x_2, ..., x_n\}$, and

$$A = \mu^A(x_1)/x_1 + \mu^A(x_2)/x_2 + ... + \mu^A(x_n)/x_n = \sum_{i=1}^{n} \mu^A(x_i)/x_i \ ,$$

then the (nonfuzzy) cardinality of A is defined as

$$\text{card } A = | A | = \sum_{i=1}^{n} \mu^A(x_i) \ .$$

Example 1.3. If $X = \{1, 2, 3, 4\}$ and $A = 0.1/1 + 0.4/2 + 0.7/3 + 1/4$, then $\text{card } A = 2.2$.

1.1.3 Basic Operations on Fuzzy Sets

The basic operations here are naturally the complement, intersection and union, as in the conventional set theory.

The *complement* of a fuzzy set A in X, $\neg A$, is defined as

$$\mu^{\neg A}(x) = 1 - \mu^A(x), \ \forall x \in X \ .$$

and it corresponds to the negation «not».

The *intersection* of two fuzzy sets A, B in X, $A \cap B$, is defined as

$$\mu^{A \cap B}(x) = \mu^A(x) \wedge \mu^B(x), \ \forall x \in X \ ,$$

where «$\wedge$» is the minimum, and it corresponds to the connective «and».

The *union* of two fuzzy sets, A, B in X, $A \cup B$, is defined as

$$\mu^{A \cup B}(x) = \mu^A(x) \vee \mu^B(x), \ \forall x \in X \ ,$$

where «$\vee$» is the maximum, and it corresponds to the connective «or».

Example 1.4. If $X = \{1, 2, ..., 10\}$,

A = "small number" = 1/1 + 1/2 + 0.8/3 + 0.5/4 + 0.3/5 + 0.1/6,
B = "large number" = 0.1/5 + 0.2/6 + 0.5/7 + 0.8/8 + 1/9 + 1/10,
then $\neg A$ = "not small number" = 0.2/3 + 0.5/4 + 0.7/5 + 0.9/6 + 1/7 + 1/8 + 1/9 + 1/10
$A \cap B$ = "small number" and "large number" = 0.1/5 + 0.1/6
$A \cup B$ = "small number" or "large number" = 1/1 + 1/2 + 0.8/3 + 0.5/4 + 0.3/5 + 0.2/6 + 0.5/7 + 0.8/8 + 1/9 + 1/10.

The above definitions are classic, and have been commonly employed though they are evidently by no means the only ones. For instance, the use of a t-norm for the intersection and an s-norm for the union has often been advocated. They are defined as follows:

a t-norm is defined as $t : [0,1] \times [0,1] \to [0,1]$ such that:

a) a t 1 = a
b) a t b = b t a
c) a t b ≥ c t d, if a ≥ c, b ≥ d
d) a t b t c = a t (b t c).

Some more relevant examples of t-norms are:

$a \wedge b = \min(a,b)$ - this is the most popular t-norm,

$a \cdot b$,

$1 - [1 \wedge ((1-a)^p + (1-b)^p)^{1/p}]$, $p \geq 1$.

an s-norm (t-conorm) is defined as $s : [0,1] \times [0,1] \to [0,1]$ such that:

a) a s 0 = a
b) a s b = b s a
c) a s b ≥ c s d, if a ≥ c, b ≥ d
d) a s b s c = a s (b s c).

Some examples of more popular s-norms are:

$a \vee b = \max(a,b)$ - this is the most popular s-norm,

$a + b - a \cdot b$

$1 \wedge \left(a^p + b^p\right)^{1/p}$, $p \geq 1$.

1.1.4 Further Properties and Related Concepts

An α*-cut* (α*-level set*) of a fuzzy set A in X is defined as the ordinary set $A_\alpha \subseteq X$ such that

$$A_\alpha = \{ x \in X : \mu^A(x) \geq \alpha \}, \quad \forall \alpha \in [0,1].$$

Example 1.5. If $A = 1/1 + 0.8/2 + 0.5/3 + 0.1/4$, then $A_{0.1} = \{1, 2, 3, 4\}$, $A_{0.5} = \{1, 2, 3\}$, $A_{0.8} = \{1, 2\}$ $A_1 = \{1\}$.

The concept of an α-cut of a fuzzy set is crucial for the so-called *decomposition theorem* which states that any fuzzy set A in X may be represented as some (equivalent) operation on conventional sets (subsets of X).

Of fundamental importance here is the so-called *extension principle* [1] which gives a formal apparatus to carry over operations (e.g., arithmetic or algebraic) from sets to fuzzy sets. Namely, if $f: X \to Y$ is a function (operation) and A is a fuzzy set in X, then A induces via f a fuzzy set B in Y given by

$$\mu^B(y) = \begin{cases} \sup\limits_{y=f(x)} \mu^A(x) & , \ f^{-1}(y) \neq \varnothing \\ 0 & , \ f^{-1}(y) = \varnothing \end{cases} . \tag{1.1}$$

Example 1.6. Let $X = \{1, 2, 3, 4\}$, $Y = \{1, 2, 3, 4, 5, 6\}$ and $y = x + 2$. If now $A = 0.1/1 + 0.2/2 + 0.7/3 + 1/4$, then $B = 0.1/3 + 0.2/4 + 0.7/5 + 1/6$.

1.1.5 Fuzzy Relations

Fuzzy relations - exemplified by «much larger than», «more or less equal», etc. - are clearly omnipresent in human discourse. Formally, if $X = \{x\}$ and $Y = \{y\}$ are two universes of discourse, then a *fuzzy relation* R is defined as a fuzzy set in the Cartesian product $X \times Y$, characterized by its membership function $\mu^R: X \times Y \to [0, 1]$; $\mu^R(x, y) \in [0, 1]$ reflects the *strength of relation* between $x \in X$ and $y \in Y$.

Example 1.7. Suppose that X = {horse, donkey} and Y = {mule, cow}. The fuzzy relation «similar» may then be defined as

$$R = \text{«similar»} = 0.8/(\text{horse, mule}) + 0.4/(\text{horse, cow}) + \\ + 0.9/(\text{donkey, mule}) + 0.5/(\text{donkey, cow})$$

to be read that, e.g., a horse and a mule are similar to degree 0.8, a horse and a cow to degree 0.4, etc.

Notice that for finite, small enough X and Y, a fuzzy relation may be evidently shown in the matrix form.

A crucial concept related to fuzzy relations is their *composition*. If we have two fuzzy relations R in $X \times Y$ and S in $Y \times Z$, then their (*max-min*) *composition* is a fuzzy relation $R \circ S$ in $X \times Z$ defined by

$$\mu^{R \circ S}(x, z) = \sup_{y \in Y} [\mu^R(x, y) \wedge \mu^S(y, z)] .$$

Fuzzy relations play a crucial role in virtually all applications, notably in decision making and control.

1.1.6 Fuzzy Numbers

The extension principle defined by (1.1) is a very powerful tool for extending non-fuzzy relationships to their fuzzy counterparts. It can also be used, e.g., to devise *fuzzy arithmetic*.

A *fuzzy number* is defined as a fuzzy set in the real line, A in R, which is normal (i.e. $\sup_{x\in R}\mu^A(x)=1$) and bounded convex (i.e. whose all α-cuts are convex and bounded). A fuzzy number may be exemplified by «about five», «a little more than 7», «more or less between 5 and 8», etc.

Notice that function f in (1.1) may be, say, the sum, product, difference and quotient, and we can extend via (1.1) the four main arithmetic operations: addition, multiplication, subtraction and division to fuzzy sets, hence obtaining fuzzy arithmetic.

Namely, for the basic four operations we obtain:

* addition

$$\mu^{A+B}(z)=\max_{z=x+y}[\mu^A(x)\wedge\mu^B(y)]\ ,\ \forall x,y,z\in R\ ;$$

* subtraction

$$\mu^{A-B}(z)=\max_{z=x-y}[\mu^A(x)\wedge\mu^B(y)]\ ,\ \forall x,y,z\in R\ ;$$

* multiplication

$$\mu^{A*B}(z)=\max_{z=x*y}[\mu^A(x)\wedge\mu^B(y)]\ ,\ \forall x,y,z\in R\ ;$$

* division

$$\mu^{A/B}(z)=\max_{z=x/y,\,y\neq 0}[\mu^A(x)\wedge\mu^B(y)]\ ,\ \forall x,y,z\in R\ .$$

Unfortunately, the use of the extension principle to define the arithmetic operations on fuzzy numbers is in general numerically inefficient, hence it is usually assumed that a fuzzy number is given in the so-called $L-R$ *representation* whose essence is that the membership function of a fuzzy number is

$$\mu^A(x)=\begin{cases}L\left(\dfrac{m-x}{\alpha}\right)\ ,\ \alpha>0,\ \forall x\le m\\ R\left(\dfrac{m-x}{\beta}\right)\ ,\ \beta>0,\ \forall x\ge m\end{cases}\ ,$$

where function L is such that:

a) $L(-x) = L(x)$,
b) $L(0) = 1$,
c) L is increasing in $[0, +\infty]$,

and similarly function R .

Here m is the *mean value* of the fuzzy number A , α is its *left spread*, and β is its *right spread*; notice that when α , β =0, then the fuzzy number A boils down to a real number m .

A fuzzy number A can now be written as $A = (m_A, \alpha_A, \beta_A)$, and the arithmetic operations may be defined in terms of the m 's, α 's and β 's. For instance, in the case of *addition*:

$$A + B = (m_A, \alpha_A, \beta_A) + (m_B, \alpha_B, \beta_B) = (m_A + m_B, \alpha_A + \alpha_B, \beta_A + \beta_B) ,$$

and similarly for the other arithmetic operations.

In practice, however, the $L-R$ representation is further simplified in that the functions L and R are assumed to be linear which leads to *triangular fuzzy numbers* exemplified by the one shown in Fig. 1.2a, and whose membership function is generally given by

$$\mu^A(x) = \begin{cases} (x - a^-)/(a - a^-) , & a^- \le x \le a \\ (a^+ - x)/(a^+ - a) , & a \le x \le a^+ \end{cases} .$$

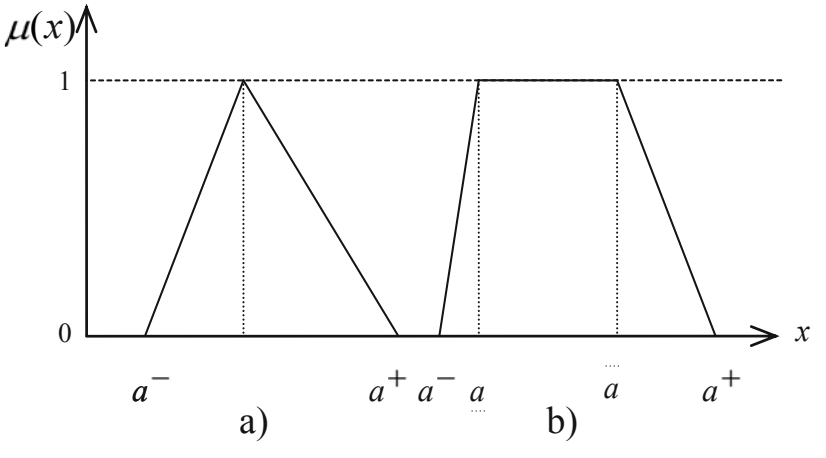


Fig. 1.2. Membership functions of triangular and trapezoid fuzzy numbers

Notice that a triangular fuzzy number may be adequate for the formalization of such terms as, say, *around 5* or *much more than 10* (in this case, evidently, a^+ must be a very large number). For the representation of such fuzzy numbers as, e.g., *more or less between 5 and 7* the *trapezoid fuzzy numbers* may be used which are exemplified in Fig. 1.2b and whose membership function is generally written as

$$\mu^A(x) = \begin{cases} (x - a^-)/(\underline{a} - a^-) \ , & a^- \le x \le \underline{a} \\ 1 \qquad\qquad\qquad\quad , & \underline{a} \le x \le \overline{a} \\ (a^+ - x)/(a^+ - \overline{a}) \ , & \overline{a} \le x \le a^+ \end{cases}.$$

1.1.7 Fuzziness and Probability

Novices at fuzzy set theory very often try to compare it with theory of probability. However, both theories are hardly comparable because they treat uncertainty differently. Some statistical uncertainty is considered in theory of probability, e.g., *a probability of hitting the target is equal to* 0.9. Fuzzy set theory allows us to operate with linguistic uncertainty, e.g., *good shot*. These types of uncertainty can be formalized with the help of:

- distribution functions – for theory of probability,
- membership functions – for fuzzy set theory.

The founder of fuzzy set theory L. Zadeh gives the following example to illustrate the crucial difference between the two distributions [17].

Example 1.8. Let us consider the assertion «The author eats X eggs at breakfast»,

$$X = \{1, 2, 3, \dots\}.$$

Some possibility and probability distributions correspond to the value of X, which can be considered as an uncertain parameter.

The possibility distribution $\pi_X(u)$ can be interpreted as a degree (a subjective measure) of easiness, with which the author eats u eggs. To define the probability distribution $P_X(u)$, it is necessary to observe the author over a period of 100 days.

Both distributions are presented below.

Possibility and probability distribution

u	1	2	3	4	5	6	7	8
$\pi_X(u)$	1	1	1	1	0.8	0.6	0.4	0.2
$P_X(u)$	0.1	0.8	0.1	0	0	0	0	0

It is seen, that the high degree of possibility $\pi_X(u)$ does not mean in any case the same high degree of probability $P_X(u)$. There is no doubt: *if an event is impossible, then it is also improbable.*

The crucial difference between theory of probability and theory of possibility is apparent wherein the axiom of complement is treated differently in these two theories:

$$P(A)+P(\overline{A})=1 \quad \text{- for theory of probability,}$$
$$\pi(A)+\pi(\overline{A})\neq 1 \quad \text{- for theory of possibility.}$$

1.2 Genetic Algorithms

As mentioned in the preface, optimization is the most important stage in solving identification problems [5 – 7]. The main difficulties in the application of the classical methods of nonlinear functions optimization [8] are related to the problems of finding a local extremum (Fig. 1.3) and overcoming of the "dimension curse" (Fig. 1.4).

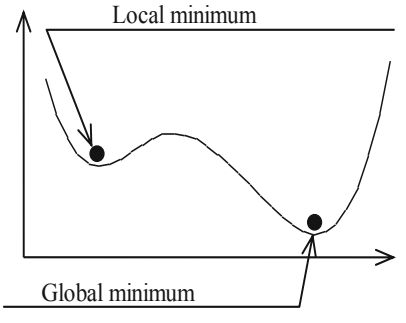


Fig. 1.3. Problem of local extremum

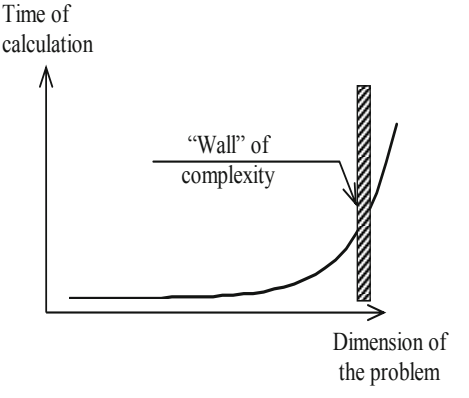


Fig. 1.4. Problem of "dimension curse"

The attempts to overcome these problems resulted in the creation of a special theory of genetic algorithms, which grow the optimal solution by crossing-over the initial variants with consequent selection using some criterion (Fig. 1.5). The general information about genetic algorithms presented in this chapter is based on the works [9, 10, 22, 23].

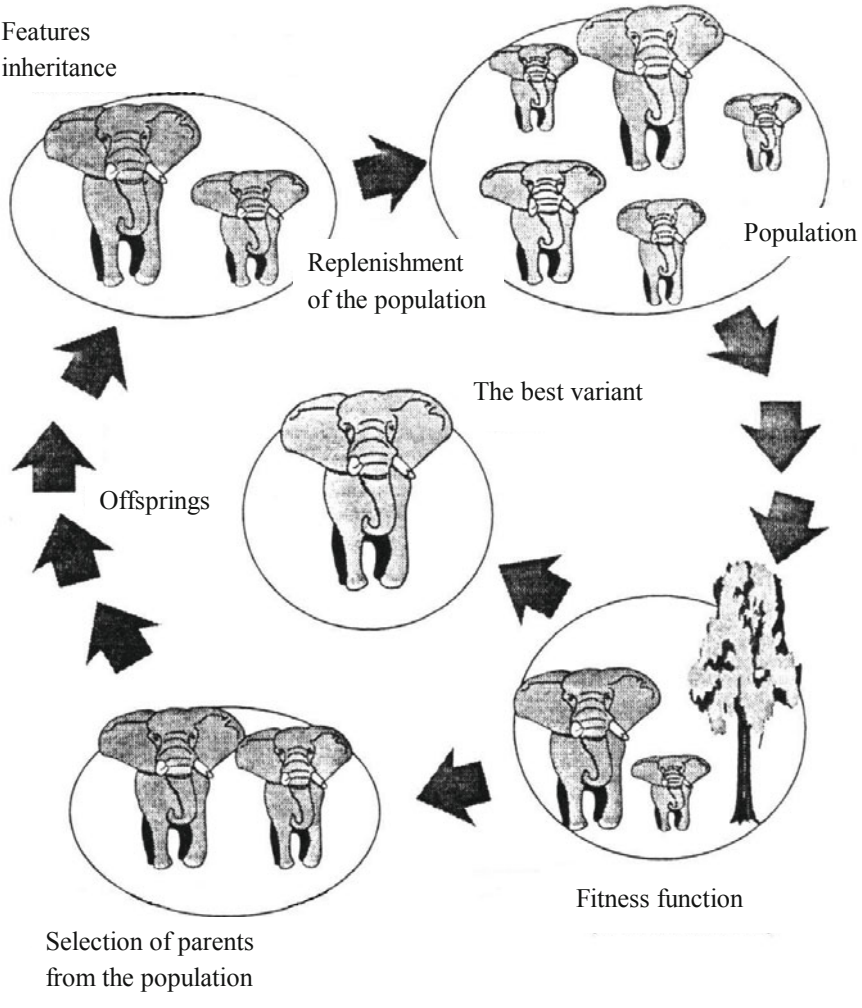


Fig. 1.5. Idea of genetic algorithm
(In: Goldberg D. Genetic Algorithms in Search, Optimization and Machine Learning, Addison Wesley, 1989)

1.2.1 General Structure of Genetic Algorithms

Genetic algorithms are stochastic search techniques based on the mechanism of natural selection and natural genetics. Genetic algorithms, differing from conventional search techniques, start with an initial set of random solutions called a *population*. Each individual in the population is called a *chromosome*, representing a

solution to the problem at hand. A chromosome is a string of symbols; it is usually, but not necessarily, a binary bit string. The chromosomes *evolve* through successive iterations, called *generations*. During each generation, the chromosomes are *evaluated*, using some measures of fitness. To create the next generation, new chromosomes, called *offsprings*, are formed by either (a) merging two chromosomes from the current generation using a *crossover* operator or (b) modifying a chromosome using a *mutation* operator. A new generation is formed by (a) selecting, according to the fitness values, some of the parents and offsprings and (b) rejecting others so as to keep the population size constant. Fitter chromosomes have higher probabilities of being selected. After several generations, the algorithms converge to the best chromosome, which hopefully represents the optimum or suboptimal solution to the problem. Let $P(t)$ and $C(t)$ be parents and offsprings in current generation t; the general structure of genetic algorithms (see Fig. 1.6) is described as follows:

Procedure: Genetic Algorithm

begin
  $t := 0$;
  initialize $P(t)$;
  evaluate $P(t)$ by using a fitness function;
  while (not termination condition) **do**
    recombine $P(t)$ to yield $C(t)$;
    evaluate $C(t)$ by using a fitness function;
    select $P(t+1)$ from $P(t)$ and $C(t)$;
    $t := t+1$;
  end
end.

Usually, initialization is assumed to be random. Recombination typically involves crossover and mutation to yield offspring. In fact, there are only two kinds of operations in genetic algorithms:

1. Genetic operations: crossover and mutation.
2. Evolution operation: selection.

The genetic operations mimic the process of heredity of genes to create new offspring at each generation. The evolution operation mimics the process of *Darwinian evolution* to create populations from generation to generation.

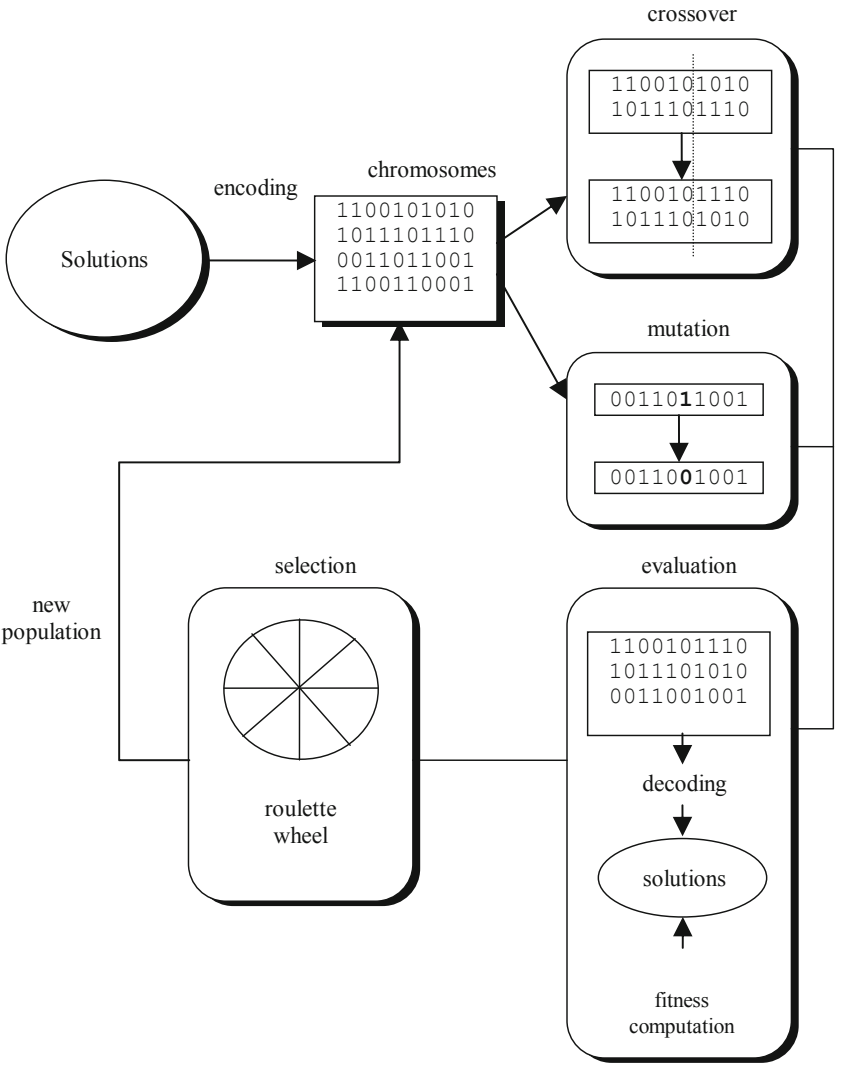


Fig. 1.6. The general structure of genetic algorithms

1.2.2 Genetic Operators

Crossover operator. Crossover is the main genetic operator. It operates on two chromosomes at a time and generates offspring by combining both chromosomes' features. A simple way to achieve crossover would be to choose a random cut-point and generate the offspring by combining the segment of one parent to the left of the cut-point with the segment of the other parent to the right of the cut-point (Fig. 1.7).

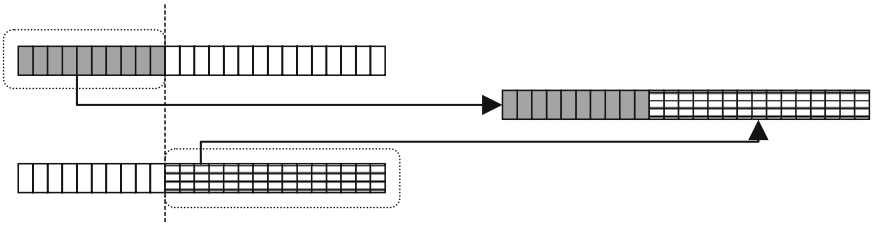


Fig. 1.7. Crossover operator

This method works well with the bit string representation. The performance of genetic algorithms depends, to a great extent, on the performance of the crossover operator used.

The *crossover rate* (denoted by p_c) is defined as the ratio of the number of offspring produced in each generation to the population size (usually denoted by *pop_size*). This ratio controls the expected number $p_c \times pop\_size$ of chromosomes to undergo the crossover operation. A higher crossover rate allows exploration of more of the solution space and reduces the chances of settling for a false optimum. However, if this rate is too high, it results in the wastage of a lot of computation time in exploring unpromising regions of the solution space.

Mutation operator. Mutation is a background operator which produces spontaneous random changes in various chromosomes. A simple way to achieve mutation would be to alter one or more genes. In genetic algorithms, mutation serves the crucial role of either (a) replacing the genes lost from the population during the selection process so that they can be tried in a new context or (b) providing the genes that were not present in the initial population.

The *mutation rate* (denoted by p_m) is defined as the percentage of the total number of genes in the population. The mutation rate controls the rate at which new genes are introduced into the population for trial. If it is too low, many genes that would have been useful are never tried out; if it is too high, there will be much random perturbation, the offspring will start losing their resemblance to the parents, and the algorithm will lose the ability to learn from the history of the search.

1.2.3 Search Techniques

Search is one of the more universal problem-solving methods for such problems where one cannot determine *a priori* the sequence of steps leading to a solution. Search can be performed with either blind strategies or heuristic strategies. Blind search strategies do not use information about the problem domain. Heuristic search strategies use additional information to guide the search along with the best search directions. There are two important issues in search strategies: exploiting the best solution and exploring the search space. Hill-climbing is an example of a strategy which exploits the best solution for possible improvement while ignoring

the exploration of the search space. Random search is an example of a strategy which explores the search space while ignoring the exploitation of the promising regions of the search space. Genetic algorithms are a class of general-purpose search methods combining elements of directed and stochastic search which can make a remarkable balance between exploration and exploitation of the search space. At the beginning of genetic search, there is a widely random and diverse population and the crossover operator tends to perform a widespread search for exploring the complete solution space. As the high fitness solutions develop, the crossover operator provides exploration in the neighbourhood of each of them. In other words, the types of searches (exploration or exploitation) a crossover performs would be determined by the environment of the genetic system (the diversity of population), but not by the operator itself. In addition, simple genetic operators are designed as general-purpose search methods (the domain-independent search methods); they perform essentially a blind search and could not guarantee to yield an improved offspring.

1.2.4 Comparison of Conventional and Genetic Approaches

Generally, the algorithm for solving optimization problems is a sequence of computational steps which asymptotically converge to an optimal solution. Most classical optimization methods generate a deterministic sequence of computation based on the gradient or higher-order derivatives of the objective function. The methods are applied to a single point in the search space. The point is then improved along the deepest descending/ascending direction gradually through iterations. This point-to-point approach has the danger of falling in local optima. Genetic algorithms perform a multiple directional search by maintaining a population of potential solutions. The population-to-population approach attempts to make the search escape from local optima. Population undergoes a simulated evolution: at each generation the relatively good solutions are reproduced, while the relatively bad solutions die. Genetic algorithms use probabilistic transition rules to select someone to be reproduced and someone to die so as to guide their search toward regions of the search space with likely improvement.

1.2.5 Advantages of Genetic Algorithms

Genetic algorithms have received considerable attention regarding their potential as a novel optimization technique. There are three major advantages when applying genetic algorithms to optimization problems:

1. Genetic algorithms do not have much mathematical requirements about the optimization problems. Due to their evolutionary nature, genetic algorithms will search for solutions without regard to the specific inner workings of the problem. Genetic algorithms can handle any kind of objective functions and any kind of constraints (i.e., linear or nonlinear) defined on discrete, continuous, or mixed search spaces.

2. The ergodicity of evolution operators makes genetic algorithms very effective at performing a global search (in probability). The traditional approaches perform a local search by a convergent stepwise procedure, which compares the values of nearby points and moves to the relative optimal points. Global optima can be found only if the problem possesses certain convexity properties that essentially guarantee that any local optima is a global optima.

3. Genetic algorithms provide us with a great flexibility to hybridize with domain-dependent heuristics to make an efficient implementation for a specific problem.

1.2.6 Genetic Algorithm Vocabulary

Because genetic algorithms are rooted in both natural genetics and computer sciences, the terminology used in genetic algorithm literature is a mixture of the natural and the artificial.

In a biological organism, the structure that encodes the prescription specifying how the organism is to be constructed is called a *chromosome*. One or more chromosomes may be required to specify the complete organism. The complete set of chromosomes is called a *genotype*, and the resulting organism is called a *phenotype*. Each chromosome comprises a number of individual structures called *genes*. Each gene encodes a particular feature of the organism, and the location, or *locus*, of the gene within the chromosome structure determines what particular characteristic the gene represents. At a particular locus, a gene may encode any of several different values of the particular characteristic it represents. The different values of a gene are called *alleles*.

The correspondence of genetic algorithm terms and optimization terms is summarized in Table. 1.1.

Table 1.1. Explanation of genetic algorithm terms

Genetic algorithms	Explanation
1. Chromosome	Solution (coding)
2. Gene (bits)	Part of solution
3. Locus	Position of gene
4. Alleles	Values of gene
5. Phenotype	Decoded solution
6. Genotype	Encoded solution

1.2.7 Examples with Genetic Algorithms

In this section we explain in detail about how a genetic algorithm actually works, using two simple examples.

Example 1.9. Optimization problem. The numerical example of optimization problem is given as follows:

$$f(x_1, x_2) = (-2x_2^3 + 6x_2^2 + 6x_2 + 10) \cdot \sin(\ln(x_1) \cdot e^{x_2})$$
$$0.5 \le x_1 \le 1.1, \qquad 1.0 \le x_2 \le 4.6$$

It is necessary to find: $\max_{x_1, x_2} f(x_1, x_2)$.

A three-dimensional plot of the objective function is shown in Fig. 1.8.

Representation. First, we need to encode decision variables into binary strings. The length of the string depends on the required precision. For example, the domain of variable x_j is $\left[a_j, b_j\right]$ and the required precision is five places after the decimal point. The precision requirements imply that the range of the domain of each variable should be divided into at least $\left(b_j - a_j\right) \times 10^5$ size ranges. The required bits (denoted with m_j) for a variable is calculated as follows:

$$2^{m_j - 1} < \left(b_j - a_j\right) \times 10^5 \le 2^{m_j} - 1$$

The mapping from a binary string to a real number for variable x_j is straightforward and completed as follows:

$$x_j = a_j + \mathit{decimal}(\mathit{substring}_j) \times \frac{b_j - a_j}{2^{m_j} - 1} \ ,$$

where $\mathit{decimal}(\mathit{substring}_j)$ represents the decimal value of $\mathit{substring}_j$ for decision variable x_j .

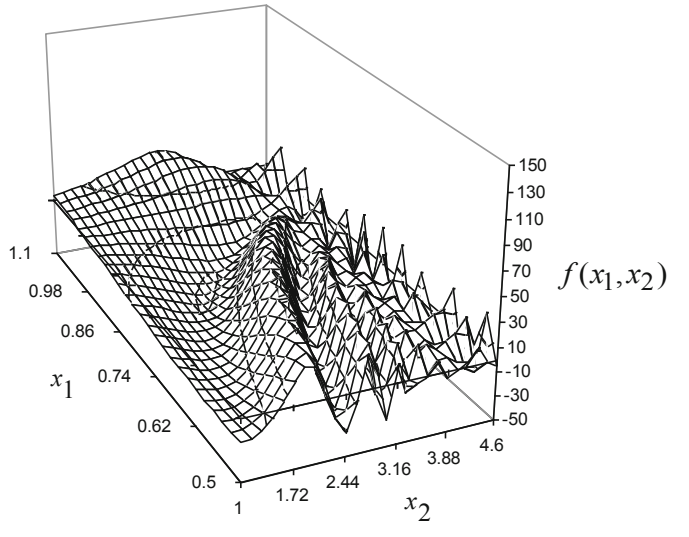


Fig. 1.8. Objective function

Suppose that the precision is set as five places after the decimal point. The required bits for variables x_1 and x_2 is calculated as follows:

$$(1.1 - 0.5) \times 100{,}000 = 60{,}000$$

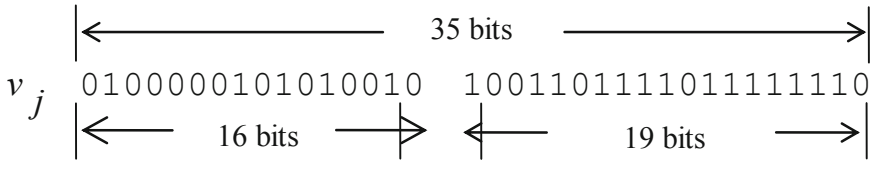
$$2^{15} < 60{,}000 \le 2^{16} - 1, \qquad m_1 = 16$$

$$(4.6 - 1.0) \times 100{,}000 = 360{,}000$$

$$2^{18} < 360{,}000 \le 2^{19} - 1, \qquad m_2 = 19$$

$$m = m_1 + m_2 = 16 + 19 = 35 \ .$$

The total length of a chromosome is 35 bits which can be represented as follows:



The corresponding values for variables x_1 and x_2 are given below:

	Binary number	Decimal number
x_1	0100000101010010	16722
x_2	10011011111011111110	319230

$$x_1 = 0.5 + 16722 \times \frac{1.1 - 0.6}{2^{16} - 1} = 0.65310 \ ,$$

$$x_2 = 1.0 + 319230 \times \frac{4.6 - 1.0}{2^{19} - 1} = 3.19198 \ .$$

Initial population. Initial population is randomly generated as follows:

v_1 = [01000001010100101001101111011111110]

v_2 = [10001110101110011000000010101001000]

v_3 = [11111000111000001000010101001000110]

v_4 = [01100110110100101101000000010111001]

v_5 = [00000010111101100010001110001101000]

v_6 = [10111110101011011000000010110011001]

v_7 = [00110100010011111000100110011101101]

v_8 = [11001011010100001100010110011001100]

v_9 = [01111110001011101100011101000111101]

v_{10} = [01111101001110101010000010101101010]

The corresponding decimal values are:

v_1 = $[x_1, x_2]$ = [0.653097, 3.191983]

v_2 = $[x_1, x_2]$ = [0.834511,2.809287]

v_3 = $[x_1, x_2]$ = [1.083310,2.874312]

v_4 = $[x_1, x_2]$ = [0.740989,3.926276]

v_5 = $[x_1, x_2]$ = [0.506940,1.499934]

v_6 = $[x_1, x_2]$ = [0.946903,2.809843]

v_7 = $[x_1, x_2]$ = [0.622600,2.935225]

v_8 = $[x_1, x_2]$ = [0.976521,3.778750]

v_9 = $[x_1, x_2]$ = [0.795738,3.802377]

v_{10} = $[x_1, x_2]$ = [0.793504,3.259521]

Evaluation. The process of evaluating the fitness of a chromosome consists of the following three steps:

1°. Convert the chromosome's genotype to its phenotype. Here, this means converting the binary string into relative real values $\mathbf{x}^k = (x_1^k, x_2^k)$, $k = 1, 2, ..., pop\_size$.

2°. Evaluate the objective function $f(\mathbf{x}^k)$.

3°. Convert the value of the objective function into fitness. For the maximization problem, the fitness is simply equal to the value of the objective function $eval(v_k) = f(\mathbf{x}^k)$, $k = 1, 2, ..., pop\_size$.

An evaluation function plays the role of the environment, and it rates chromosomes in terms of their fitness.

The fitness function values of the above chromosomes are as follows:

$eval(v_1)$ = $f(0.653097,3.191983)$ = 20.432394

$eval(v_2)$ = $f(0.834511,2.809287)$ = -4.133627

$eval(v_3)$ = $f(1.083310,2.874312)$ = 28.978472

$eval(v_4)$ = $f(0.740989,3.926276)$ = -2.415740

$eval(v_5)$ = $f(0.506940,1.499934)$ = -2.496340

$eval(v_6)$ = $f(0.946903,2.809843)$ = -23.503709

$eval(v_7)$ = $f(0.622600,2.935225)$ = -13.878172

$eval(v_8)$ = $f(0.976521,3.778750)$ = -8.996062

$eval(v_9)$ = $f(0.795738,3.802377)$ = 6.982708

$eval(v_{10})$ = $f(0.793504,3.259521)$ = 6.201905

It is clear that chromosome v_3 is the strongest one and that chromosome v_6 is the weakest one.

Selection. In most practices, a *roulette wheel* approach is adopted as the selection procedure [22]; it belongs to the fitness-proportional selection and can select a new population with respect to the probability distribution based on fitness values. The roulette wheel can be constructed as follows:

1. Calculate the fitness value $eval(v_k)$ for each chromosome v_k:

$$eval\ (v_k) = f\ (\mathbf{x}^k)\ ,\ k = 1,2,...,pop\_size\ .$$

2. Calculate the total fitness for the population:

$$F = \sum_{k=1}^{pop\_size} \left(eval\ (v_k) - \min_{j=1,pop\_size} \{eval\ (v_j)\} \right)$$

3. Calculate selection probability p_k for each chromosome v_k:

$$p_k = \frac{eval(v_k) - \min_{j=1,pop\_size} \{eval\ (v_j)\}}{F}\ ,\quad k = 1,2,...,pop\_size\ .$$

4. Calculate cumulative probability q_k for each chromosome v_k:

$$q_k = \sum_{j=1}^{k} p_j \ , \qquad k = 1, 2, ..., pop\_size \ .$$

The selection process begins by spinning the roulette wheel pop_size times; each time, a single chromosome is selected for a new population in the following way:

1°. Generate a random number r from the range $[0,1]$.

2°. If $r \le q_1$, then select the first chromosome v_1; otherwise, select the k th chromosome v_k ($2 \le k \le pop\_size$) such that $q_{k-1} < r \le q_k$.

The total fitness F of the population is:

$$F = \sum_{k=1}^{10} \left(eval\ (v_k) - \min_{j=1,10} \{ eval\ (v_j) \} \right) = 242.208919 \ .$$

The probability of a selection p_k for each chromosome v_k ($k = 1, 2, ..., 10$) is as follows:

$p_1 = 0.181398$, $p_2 = 0.079973$, $p_3 = 0.216681$,
$p_4 = 0.087065$, $p_5 = 0.086732$, $p_6 = 0.000000$,
$p_7 = 0.039741$, $p_8 = 0.059897$, $p_9 = 0.125868$,
$p_{10} = 0.122645$.

The cumulative probability q_k for each chromosome v_k ($k = 1, 2, ..., 10$) is as follows:

$q_1 = 0.181398$, $q_2 = 0.261370$, $q_3 = 0.478052$,
$q_4 = 0.565117$, $q_5 = 0.651849$, $q_6 = 0.651849$,
$q_7 = 0.691590$, $q_8 = 0.751487$, $q_9 = 0.877355$,
$q_{10} = 1.000000$.

Now we are ready to spin the roulette wheel 10 times, and each time we select a single chromosome for a new population. Let us assume that a random sequence of 10 numbers from the range $[0,1]$ is as follows:

0.301431	0.322062	0.766503	0.881893
0.350871	0.583392	0.177618	0.343242
0.032685	0.197577 .		

The first number $r_1 = 0.301431$ is greater than q_2 and smaller than q_3, meaning that the chromosome v_3 is selected for the new population; the second number $r_2 = 0.322062$ is greater than q_2 and smaller than q_3, meaning that the chromosome v_3 is again selected for the new population; and so on. Finally, the new population consists of the following chromosomes:

$$v'_1 = [11111000111000001000010101001000110] \quad (v_3)$$
$$v'_2 = [11111000111000001000010101001000110] \quad (v_3)$$
$$v'_3 = [11001011010100001100010110011001100] \quad (v_8)$$
$$v'_4 = [01111110001011101100011101000111101] \quad (v_9)$$
$$v'_5 = [11111000111000001000010101001000110] \quad (v_3)$$
$$v'_6 = [01100110110100101101000000010111001] \quad (v_4)$$
$$v'_7 = [01000001010100101001101111011111110] \quad (v_1)$$
$$v'_8 = [11111000111000001000010101001000110] \quad (v_3)$$
$$v'_9 = [01000001010100101001101111011111110] \quad (v_1)$$
$$v'_{10} = [10001110101110011000000010101001000] \quad (v_2)$$

Crossover. Crossover used here is one-cut-point method, which randomly selects one cut-point and exchanges the right parts of two parents to generate offspring. Consider two chromosomes as follows, and the cut-point is randomly selected after the 17th gene:

$$v_1 = [11111000111000001000010101001000110]$$
$$v_2 = [10001110101110011000000010101001000]$$

The resulting offspring by exchanging the right parts of their parents would be as follows:

$$v_1 = [11111000111000001000000010101001000]$$
$$v_2 = [10001110101110011000010101001000110]$$

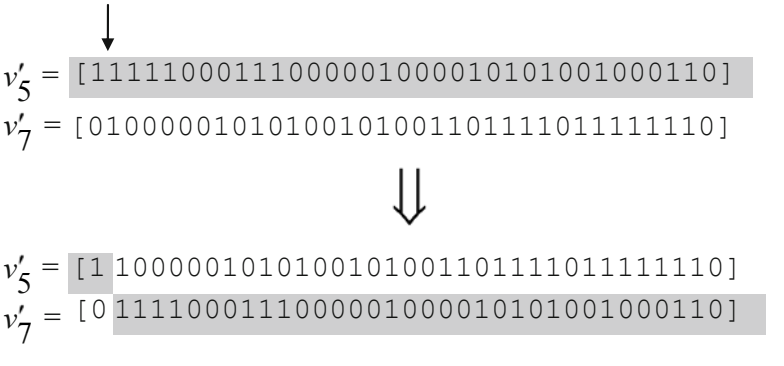
The probability of crossover is set as $p_c = 0.25$, so we expect that, on average, 25% of chromosomes undergo crossover. Crossover is performed in the following way:

```
Procedure: Crossover
begin
  k :=0 ;
  while ( k ≤ 10 ) do
      r_k := random number from [0,1] ;
      if ( r_k < 0.25 ) then
          select v_k as one parent for crossover;
      end ;
      k: = k +1 ;
   end ;
end.
```

Assume that the sequence of random numbers is:

0.625721	0.266823	0.288644	0.295114
0.163274	0.567461	0.085940	0.392865
0.770714	0.548656 .		

This means that the chromosomes v_5' and v_7' were selected for crossover. We generate a random integer number pos from the range [1, 34] (because 35 is the total length of a chromosome) as cutting point or in other words, the position of the crossover point. Assume that the generated number pos equals 1, the two chromosomes are cut after the first bit, and offspring are generated by exchanging the right parts of them as follows:



Mutation. Mutation alters one or more genes with a probability equal to the mutation rate. Assume that the 18th gene of the chromosome v_1' is selected for a mutation. Since the gene is 1, it would be flipped into 0. Thus the chromosome after mutation would be:

$$v_1' = [111110001110000010\,00010101001000110]$$

$$\Downarrow$$

$$v_1' = [111110001110000011\,00010101001000110]$$

The probability of mutation is set as $p_m = 0.01$, so we expect that, on average, 1% of the total bit of the population would undergo mutation. There are $m \times pop\_size = = 35 \times 10 = 350$ bits in the whole population; we expect 3.5 mutations per generation. Every bit has an equal chance to be mutated. Thus we need to generate a sequence of random numbers r_k $(k = 1..350)$ from the range [0,1]. Suppose that the following genes will go through mutation:

Position of gene in population	Number of chromosome	Position of gene in population	Random number | r_k
111	4	6	0.009857
172	5	32	0.003113
211	7	1	0.000946
347	10	32	0.001282

After mutation, we get the final population as follows:

$v_1' =$ [11111000111000001000010101001000110]
$v_2' =$ [11111000111000001000010101001000110]
$v_3' =$ [11001011010100001100010110011001100]
$v_4' =$ [01111010001011101100011101000111101]
$v_5' =$ [11000001010100101001101111011110110]
$v_6' =$ [01100110110100101101000000010111001]
$v_7' =$ [11111000111000001000010101001000110]
$v_8' =$ [11111000111000001000010101001000110]
$v_9' =$ [01000001010100101001101111011111110]
$v_{10}' =$ [10001110101110011000000010101000000] .

The corresponding decimal values of variables x_1 and x_2 and fitness are as follows:

f(1.083310,2.874312)=28.978472
f(1.083310,2.874312)=28.978472
f(0.976521,3.778750)=-8.996062
f(0.786363,3.802377)=9.366723
f(0.953101,3.191928)=-23.229745
f(0.740989,3.926276)=-2.415740
f(1.083310,2.874312)=28.978472
f(1.083310,2.874312)=28.978472
f(0.653097,3.191983)=20.432394
f(0.834511,2.809232)=-4.138564

Now we just completed one iteration of the genetic algorithm. The test run is terminated after 1000 generations. We have obtained the best chromosome in the 419th generation:

$$v^* = \texttt{[01000011000100110110010011011101001]}$$
$$eval\ (v^*) = f(0.657208,2.418399) = 31.313555$$
$$x_1^* = 0.657208 \qquad x_2^* = 2.418399$$
$$f(x_1^*, x_2^*) = 31.313555.$$

Example 1.10. Word matching problem. Another nice example to show the power of genetic algorithms, the *word matching problem* tries to evolve an expression of «live and learn» from the randomly-generated lists of letters with a genetic algorithm. Since there are 26 possible letters plus space character for each of 14 locations in the list, the probability that we get the correct phrase in a pure random way is $(1/27)^{14} = 9.14\times 10^{-22}$, which is almost equal to zero.

We use a list of ASCII integers to encode the string of letters. The lowercase letters in ASCII are represented by numbers in the range [97,122] and the space character is 32 in the decimal number system. For example, the string «live and learn» is converted into the following chromosome represented with ASCII integers:

```
[108,105,118,101, 32, 97,110,100, 32,108,101, 97,114,110]
```

Generate an initial population of 10 random phrases as follows:

```
[115,111,113,114,100,109,119,115,118,106,108,116,112,106]
[116,111,112,122,122,119,103,106,122,100,114, 99,115,103]
[117,106,111,102,113, 97, 32,114,114,112,117,117,103,115]
[ 32, 97,114,118,104, 99,117,105,100,118, 98,114,102, 32]
[119, 99,117,103,102,122,112, 32,114,122,101,107,101,106]
```

```
[116,117,100,120, 32, 32, 97,122,118,121,104,103, 97,113]
[118,100,104,122,101,102,114,113,113, 98,111,114, 98,116]
[120,106,105,101, 98,110,108,116, 97,118,104,116,103,118]
[102,117,115,100,122,107,118,104,107,112, 99,109,120,109]
[100,110,100,102,115, 32,107,104,104, 32,121,109, 99,120]
```

Now, we convert this population to string to see what they look like:

```
«soqrdmwsvjltpj»
«topzzwgjzdrcsg»
«ujofqa rrpuugs»
« arvhcuidvbrf »
«wcugfzp rzekej»
«tudx  azvyhgaq»
«vdhzefrqqborbt»
«xjiebnltavhtgv»
«fusdzkvhkpcmxm»
«dndfs khh ymcx»
```

Fitness is calculated as the number of matched letters. For example, the fitness for string «ujofqa rrpuugs» is 1. Only mutation is used which results in a change to a given letter with a given probability. Now, we run our genetic algorithm with 32 generations to see how well it works. The best one of each generation is listed in Table 1.2.

Table 1.2. The best string for each generation

Gen.	String	Fitness function	Gen.	String	Fitness function
1	ujofqa rrpuugs	1	17	liie xnd leaez	10
2	wfugfzpnrzewen	2	18	liye xnt learn	11
3	wiipvap ozekej	3	19	liye xnt learn	11
4	wi gvahdlzerej	4	20	liye xnt learn	11
5	liigvapt yekej	5	21	live xnd nearn	12
6	liigvapt yekej	5	22	live xnd nearn	12
7	lqie zp zekrj	6	23	live xnd nearn	12
8	lqie zp zekrj	6	24	live xnd nearn	12
9	lqie zp zekrj	6	25	live gnd learn	13
10	ljie zni yeaez	7	26	live gnd learn	13
11	ljie zni yeaez	7	27	live gnd learn	13
12	liie xnt beaez	8	28	live gnd learn	13
13	liye nd yeaez	9	29	live and learn	14
14	liye nd yeaez	9	30	live and learn	14
15	liye nd yeaez	9	31	live and learn	14
16	liie xnd leaez	10	32	live and learn	14

After 29 generations, the population produced the desired phrase. The total examined chromosomes are 290. If we use pure random method to produce 290 random phrases, could we have a match?

1.3 Neural Networks

This chapter is written on the basis of the works [3, 4, 24, 25]. The additional information relative to artificial neural networks can be found in the works [26 – 31].

1.3.1 Neural Net Basics

The imitation of human minds in machines has inspired scientists for the last century. About 50 years ago, researchers created the first electronic hardware models of nerve cells. Since then, the greater scientific community has been working on new mathematical models and training algorithms. Today, so-called neural nets absorb most of the interest in this domain. Neural nets use a number of simple computational units called "neurons", of which each tries to imitate the behavior of a single human brain cell. The brain is considered as a "biological neural net" and implementations on computers are considered as "neural nets". Fig. 1.9 shows the basic structure of such a neural net.

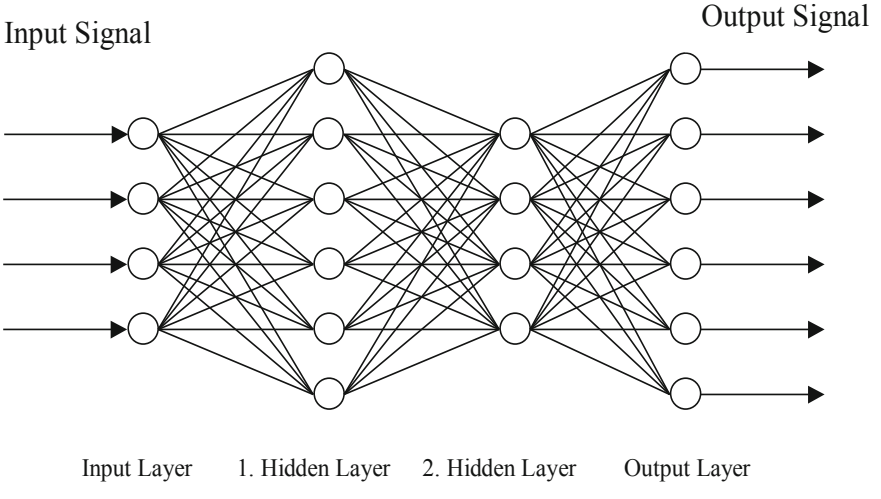


Fig 1.9. Basic structure of an artificial neural net

Each neuron in a neural net processes the incoming inputs to an output. The output is then linked to other neurons. Some of the neurons form the interface of the neural net. The neural net shown in Fig. 1.9 has a layer for the input signals and one for the output signals. The information enters the neural net at the input layer. All layers of the neural net process these signals through the net until they reach the output layer.

The objective of a neural net is to process the information in a way that it is previously trained. Training uses either sample data sets of inputs and corresponding outputs or a teacher who rates the performance of the neural net. For this training, neural nets use so-called learning algorithms. Upon creation, a neural net is dumb and does not exhibit any behavior at all. The learning algorithms then modify the individual neurons of the net and the weight of their connections in such a way that the behavior of the net reflects the desired one.

1.3.2 Mimic of Human Nerve Cells

Researchers in the area of neural nets have analyzed various models of human brain cells. In the following, we only describe the one most commonly used in industrial applications.

The human brain contains about 10^{11} nerve cells with about 10^{14} connections to each other. Fig. 1.10 shows the simplified scheme of such a human neuron. The cell itself contains a kernel, and the outside is an electrical membrane. Each neuron has an activation level, which ranges between a maximum and a minimum. Hence, in contrast to Boolean logic, more then two values exist.

To increase or decrease the activation of this neuron by other neurons, so-called synapses exist. These synapses carry the activation level from a sending neuron to a receiving neuron. If the synapse is an excitatory one, the activation level from the sending neuron increases the activation of the receiving neuron. If the synapse is an inhibiting one, the activation from the sending neuron decreases the activation of the receiving neuron. Synapses differ not only in whether they excite or inhibit the receiving neuron, but also in the amount of this effect (synaptic strength). The output of each neuron is transferred by the so-called axon, which ends in as much as 10,000 synapses influencing other neurons.

The considered neuron model underlies most of today's neural net applications. Note that this model is only a very coarse approximation of reality. You cannot exactly model even one single human neuron; it is beyond the ability of humans to model. Hence, every work based on this simple neuron model is unable to exactly copy the human brain. However, many successful applications using this technique prove the benefit of neural nets based on the simple neuron model.

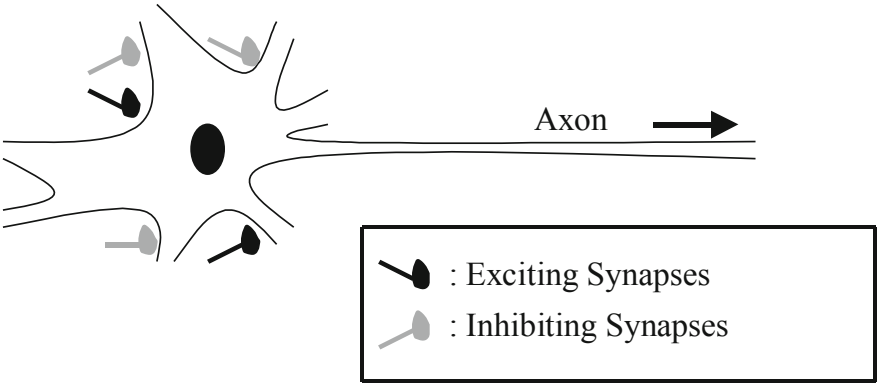


Fig. 1.10. Simplified scheme of a human neuron

1.3.3 Mathematical Model of a Neuron

Various mathematical models are based on the simple neuron concept. Fig. 1.11 shows the most common one. First, the so-called propagation function combines all inputs X_i that stem from the sending neurons. The means of combination is a weighted sum, where the weights w_i represent the synaptic strength. Exciting synapses have positive weights, inhibiting synapses have negative weights. To express a background activation level of the neuron, an offset (bias) Θ is added to the weighted sum.

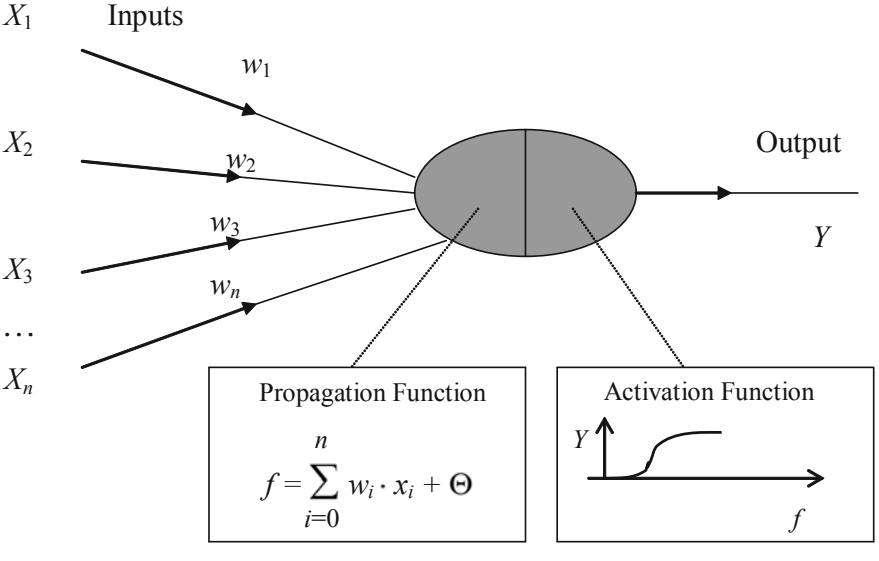


Fig. 1.11. Simple mathematical model of a neuron.

The so-called activation function computes the output signal *Y* of the neuron from the activation level *f*. For this, the activation function is of the sigmoid type as plotted in the lower right box of Fig. 1.11. Other types of the activation function are the linear function and the radial-symmetric function showed in Fig. 1.12.

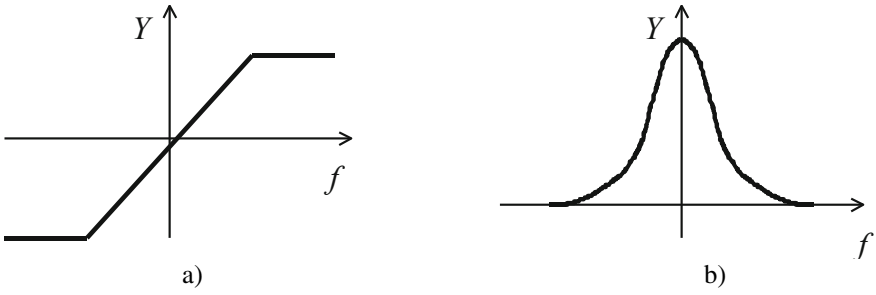


Fig. 1.12. Activation functions of a neuron:a) linear; b) radial-symmetric

1.3.4 Training Neural Nets

There are multiple ways to build a neural net. They differ in their topology and the learning methods they employ.

The first step in designing a neural net solution is teaching the desired behavior. This is called the learning phase. Here, you can either use sample data sets or a "teacher". A teacher is either a mathematical function or a person who rates the quality of the neural net performance. Since neural nets are mostly used for complex applications where no good mathematical models exist, and rating the performance of a neural net is hard in most applications, most applications use sample data training.

After completion of learning, the neural net is ready to use. This is called the working phase. As a result of the training, the neural net will output values similar to those in the sample data sets when the input values match one of the training samples. For input values in between, it approximates output values. In the working phase, the behavior of the neural net is deterministic. That is, for every combination of input values, the output value will always be the same. During the working phase, the neural net does not learn. This is important in most technical applications to ensure that the system never drifts to hazardous behavior.

Pavlov's dogs. So, how do you teach a neural net? Basically, it works like Pavlov's dogs. More then hundred years ago, the researcher Pavlov experimented with dogs. When he showed the dogs food, the dogs salivated. He also installed bells in the dogs' cages. When he rang the bell, the dogs did not salivate, as they saw no link between the bell and the food. Then he trained the dogs by always letting the bell ring when he presented the dogs food. After a while, the dogs also salivated when just the bell rang and he showed no food.

Fig. 1.13 shows how the simple neuron model can represent Pavlov's experiment. There are two input neurons: one represents the fact that the dog sees food, the other one the fact that the bell rings. Both input neurons have links to the output neuron. These links are the synapses. The thickness of the lines represents synapse weights. Before learning, the dog only reacts to the food and not the bell. Hence, the line from the left input neuron to the output neuron is thick, while the line from the right input neuron to the output neuron is very thin.

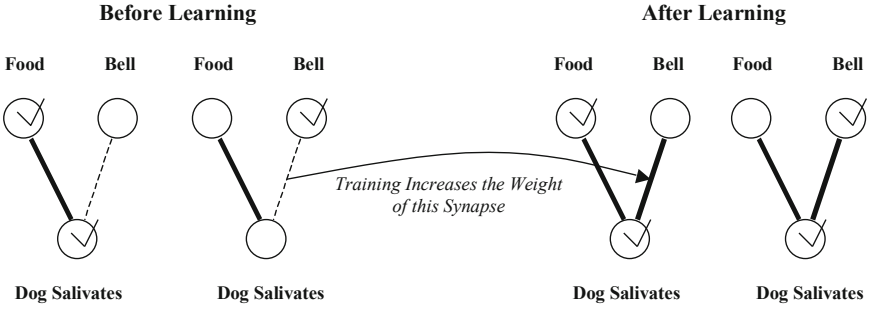


Fig. 1.13. Principle of the Pavlov dog experiment

The Hebbian learning rule. Constantly letting the bell ring when food is presented creates an association between the bell and the food. Hence, the right line also becomes thicker - the synapse weight increases. From these experiments, in 1949 a researcher by the name of Hebb deduced the following learning rule:

***Increase weight* to active input neuron, if the output of this neuron *should be active*.**
***Decrease weight* to active input neuron, if the output of this neuron *should be inactive*.**

This rule, called the Hebbian rule, is the forerunner of all learning rules, including today's most used neural net learning algorithm, the so-called error back propagation algorithm.

1.3.5 Error Back Propagation Algorithm

The learning rule for multilayer neural nets is called the "generalized delta rule", or the "back propagation rule", and was suggested in 1986 by Rumelhart, McClelland, and Williams. It signaled the renaissance of the entire subject. It was later found that Parker had published similar results in 1982, and then Werbos was shown to have done the work in 1984. Such is the nature of science; groups working in diverse fields cannot keep up with all the advances in other areas, and there is often duplication of effort. However, the paper of Rumelhart et al. published in "Nature" (1986) is still one of the most important works in this field.

Learning of the net is begun by the net being shown a pattern and calculating its response. Comparison with the desired response enables the weights to be altered so that the network can produce a more accurate output the next time. The learning rule provides the method for adjusting the weights in the network. Information about the output is available to units in earlier layers, so that these units can have their weights adjusted so as to decrease the error the next time.

When we show the untrained network an input pattern, it will produce any random output. An error function represents the difference between the network's current output and the correct output that we want it to produce. In order to learn successfully we want to make the output of the net approach the designed output, that is, we want to continually reduce the value of this error function. This is achieved by adjusting the weights on the links between the units; the generalized delta rule does this by calculating the value of the error function for that particular input, and then back-propagating (hence the name!) the error from one layer to the previous one. Each unit in the net has its weights adjusted so that it reduces the value of the error function; for units actually on the output, their output and desired output are known, so adjusting the weights is relatively simple, but for units in the middle layer, the adjustment is not so obvious. Intuitively, we might guess that the hidden units that are connected to outputs with a large error should have their weights adjusted a lot, while units that feed almost correct outputs should not be altered much. In other words, the weights for a particular node should be adjusted in direct proportion to the error in the units to which it is connected; that is why back-propagating these errors through the net allows the weights between all the layers to be correctly adjusted. In this way the error function is reduced and the network learns.

The main formulae for the error back propagation method have been obtained in [3, 4].

The notation used is as follows:

E_p is the error function for pattern p;

t_{pj} is the target output for pattern p on node j;

o_{pj} is the actual output for pattern p on node j;

w_{ij} is the weight from node i to node j.

Let us define the error function to be proportional to the square of the difference between the actual and desired output, for all the patterns to be learnt:

$$E_p = \frac{1}{2}\sum_j \left(t_{pj} - o_{pj}\right)^2 \quad . \tag{1.2}$$

The $\frac{1}{2}$ makes the math a bit simpler, and brings this specific error function into line with other similar measures.

The activation of each unit *j*, for pattern *p*, can be written simply as the weighted sum:

$$net_{pj} = \sum_i w_{ij} o_{pi} \quad . \tag{1.3}$$

The output of each unit *j* is the threshold function f_j activated on the weighted sum. In the multilayer networks, it is usually the sigmoid function, although any continuously differentiable monotonic function can be used:

$$o_{pj} = f_j\left(net_{pj}\right) \quad . \tag{1.4}$$

We can write by the chain rule:

$$\frac{\partial E_p}{\partial w_{ij}} = \frac{\partial E_p}{\partial net_{pj}} \frac{\partial net_{pj}}{\partial w_{ij}} \quad . \tag{1.5}$$

Looking at the second term in (1.5), and substituting in (1.3)

$$\frac{\partial net_{pj}}{\partial w_{ij}} = \frac{\partial}{\partial w_{ij}} \sum_k w_{kj} o_{pk} = \sum_k \frac{\partial w_{jk}}{\partial w_{ij}} o_{pk} = o_{pi} \ , \tag{1.6}$$

since $\frac{\partial w_{kj}}{\partial w_{ij}} = 0$, except when $k = i$, and this derivative is equal to unity.

We can define the change in error as a function of the change in the net inputs to a unit as

$$-\frac{\partial E_p}{\partial net_{pj}} = \delta_{pj} \ , \tag{1.7}$$

and so (1.5) becomes

$$-\frac{\partial E_p}{\partial w_{ij}} = \delta_{pj} o_{pi} \quad . \tag{1.8}$$

Decreasing the value E_p therefore means making the weight changes proportional to $\delta_{pj} o_{pj}$, i.e.,

$$\Delta_p w_{ij} = \eta \delta_{pj} o_{pi} \ , \tag{1.9}$$

where η is a learning rate.

We now need to know what δ_{pj} is for each of the units. Using (1.7) and the chain rule, we can write:

$$\delta_{pj} = -\frac{\partial E_p}{\partial net_{pj}} = -\frac{\partial E_p}{\partial o_{pj}} \frac{\partial o_{pj}}{\partial net_{pj}} \quad . \tag{1.10}$$

Consider the second term, and from (1.4):

$$\frac{\partial o_{pj}}{\partial net_{pj}} = f_j'\left(net_{pj}\right) \quad . \tag{1.11}$$

Consider now the first term in (1.10). From (1.2), we can easy obtain

$$\frac{\partial E_p}{\partial o_{pj}} = -\left(t_{pj} - o_{pj}\right) . \tag{1.12}$$

Thus

$$\delta_{pj} = f_j^{'}\left(net_{pj}\right)\left(t_{pj} - o_{pj}\right) . \tag{1.13}$$

This is useful for the output units, since the target and output are both available, but not for the hidden units, since their targets are not known.

Therefore, if unit *j* is not an output unit, we can write, by the chain rule again, that

$$\frac{\partial E_p}{\partial o_{pj}} = \sum_k \frac{\partial E_p}{\partial net_{pk}} \frac{\partial net_{pk}}{\partial o_{pj}} = \sum_k \frac{\partial E_p}{\partial net_{pk}} \frac{\partial}{\partial o_{pj}} \sum_i w_{ik} o_{pi} , \tag{1.14}$$

$$\sum_k \frac{\partial E_p}{\partial net_{pk}} \frac{\partial}{\partial o_{pj}} \sum_i w_{ik} o_{pi} = -\sum_k \delta_{pk} w_{jk} , \tag{1.15}$$

using (1.3) and (1.7), noticing that the sum drops out since the partial differential is non-zero for only one value, just as in (1.6). Substituting (1.15) in (1.10), we get finally

$$\delta_{pj} = f_j^{'}\left(net_{pj}\right) \sum_k \delta_{pk} w_{jk} . \tag{1.16}$$

Equations (1.13) and (1.16) are the basis of the multilayer network learning method.

One advantage of using the sigmoid function as the nonlinear threshold function is that it is quite like the step function, and so should demonstrate behavior of a similar nature. The sigmoid function is defined as

$$f(net) = \frac{1}{1 + e^{-k \cdot net}}$$

and has the range $0 < f(net) < 1$. k is a positive constant that controls the "spread" of the function - large values of k squash the function until as $k \to \infty$ when $f(net) \to$ Heaviside function. It also acts as an automatic gain control, since for small input signals the slope is quite steep and so the function is changing quite rapidly, producing a large gain. For large inputs, the slope and thus the gain is much less. This means that the network can accept large inputs and still remain sensitive to small changes.

A major reason for its use is that it has a simple derivative, however, and this makes the implementation of the back-propagation system much easier. Given that the output of unit, o_{pj} is given by

$$o_{pj} = f(net) = \frac{1}{1+e^{-k \cdot net}} ,$$

the derivative with respect to that unit, $f'(net)$, is give by

$$f'(net) = \frac{ke^{-k \cdot net}}{(1+e^{-k \cdot net})^2} = \frac{kf(net)}{1-f(net)} = ko_{pj}(1-o_{pj}) .$$

The derivative is therefore a simple function of the outputs.

1.3.6 The Multilayer Neural Network Learning Algorithm

The algorithm for the multilayer neural network learning that implements the back-propagation training rule is shown below. It requires the units to have thresholding nonlinear functions that are continuously differentiable, i.e. smooth everywhere. We have assumed the use of the sigmoid function, $f(net) = \frac{1}{1+e^{-k \cdot net}}$, since it has a simple derivative.

The multilayer neural network learning algorithm includes the following steps.

1°. Initialize weights and thresholds. Set all weights and thresholds to small random values.

2°. Present input and desired output.

Present input $X_p = \{x_0, x_1, ..., x_{n-1}\}$ and target output $T_p = \{t_0, t_1, ..., t_{m-1}\}$, where n is a number of input nodes and m is a number of output nodes. Set $w_0 = -\Theta$ the bias, and $x_0 = 1$.

For classification, T_p is set to zero except for one element set to 1 that corresponds to the class that X_p is in.

3°. Calculate actual output.

Each layer calculates

$$y_{pj} = f\left[\sum_{i=0}^{n-1} w_i x_i\right]$$

and passes that as input to the next layer. The final layer output values are o_{pj}.

4°. Adapt weights.

Start from the output layer, and work backwards

$$w_{ij}(t+1) = w_{ij}(t) + \eta \delta_{pj} o_{pj} ,$$

where $w_{ij}(t)$ represents the weights from node i to node j at time t, η is a learning rate, and δ_{pj} is an error term for pattern p on node j.

For output units

$$\delta_{pj} = ko_{pj}\left(1-o_{pj}\right)\left(t_{pj}-o_{pj}\right) .$$

For hidden units

$$\delta_{pj} = ko_{pj}\left(1-o_{pj}\right)\sum_{k}\delta_{pk}w_{jk} ,$$

where the sum is over the k nodes in the layer above node j.

References

1. Zadeh, L.A.: The Concept of a Linguistic Variable and its Application to Approximate Reasoning, Part 1-3. Information Sciences 8, 199–251 (1975); 9, 301 – 357, 43 – 80 (1976)
2. Zadeh, L.A.: Fuzzy Sets as a Basic for a Theory of Possibility. Fuzzy Sets and Systems 1, 3–28 (1978)
3. Rummelhart, D.E., McClelland, J.L.: Parallel Distributed Processing: Explorations in the Microstructure of Cognition, vol. 1,2, p. 320. The MIT Press (1986)
4. Rumelhart, D.E., Hinton, G.E., Williams, R.J.: Learning Internal Representation by Back - Propagation Errors. Nature 323, 533–536 (1986)
5. Eickhoff, P.: System Identification: Parameter and State Estimation. Wiley, London (1974)
6. Tsypkin, Y.Z.: Information Theory of Identification, p. 320. Nauka, Moscow (1984) (in Russian)
7. Shteinberg, S.E.: Identification in Control Systems, p. 81. Energoatomizdat, Moscow (1987) (in Russian)
8. Reklaitis, G.V., Ravindran, A., Ragsdell, K.M.: Engineering Optimization. In: Methods and Applications. John Wiley & Sons, New York (1983)
9. Goldberg, D.: Genetic Algorithms in Search, Optimization and Machine Learning. Addison Wesley (1989)
10. Tang, K.S., Man, K.F., Kwong, S., He, Q.: Genetic Algorithms and Their Applications. IEEE Signal Processing Magazine, 22–36 (November 1996)
11. Kaufmann, A., Gupta, M.M.: Introduction to Fuzzy Arithmetic: Theory and Applications. Van Nostrand Reinhold, New York (1985)
12. Pospelov, D.A. (ed.): Fuzzy Sets in Management Models and Artificial Intelligence. Nauka, Moscow (1986) (in Russian)
13. Bellman, R.E., Zadeh, L.A.: Decision-Making in a Fuzzy Environment. Management Science 17(4), 141–164 (1970)
14. Yager, R.R.: Fuzzy Set and Possibility Theory: Recent Developments. Pergamon Press, New York (1982) (Russian Translation, 1986)
15. Dubois, D., Prade, H.: Possibility Theory: An Approach to Computerized Processing of Uncertainty, p. 263. Plenum Press, New York (1988)

16. Borisov, A.N., Krumberg, O.A., Fedorov, I.P.: Decision Making based on Fuzzy Models: Examples Use, p. 184. Zinatne, Riga (1990) (in Russian)
17. Zimmermann, H.-J.: Fuzzy Set Theory and Its Applications, p. 315. Kluwer, Dordrecht (1991)
18. Zadeh, L., Kacprzyk, J.: Fuzzy Logic for the Management of Uncertainty, p. 676. John Wiley & Sons, Chichester (1992)
19. Klir, G., Yuan, B.: Fuzzy Sets and Fuzzy Logic: Theory and Applications, p. 592. Prentice Hall PTR, New York (1995)
20. Ross, T.J.: Fuzzy Logic with Engineering Applications, p. 593. Wiley, Chichester (1995)
21. Pedrycz, W., Gomide, F.: An introduction to fuzzy sets: Analysis and Design. A Bradford Book, p. 465. The MIT Press (1998)
22. Gen, M., Cheng, R.: Genetic Algorithms and Engineering Design, p. 352. John Wiley & Sons, New York (1997)
23. Haupt, R., Haupt, S.: Practical Genetic Algorithms, p. 177. John Willey & Sons, New York (1998)
24. Hinton, G.E.: How Neural Networks Learn from Experience. Scientific American, 145–151 (September 1992)
25. Hung, S.L., Adeli, H.: Machine Learning, p. 211. John Willey & Sons, New York (1995)
26. Mkrtchan, S.O.: Neurons and Neural Networks, p. 272. Energia, Moscow (1971) (in Russian)
27. Amosov, N.M. (ed.): Neurocomputers and Intelligent Robots, Kiev, Naukova Dumka, p. 272 (1991) (in Russian)
28. von Altrock, C.: Fuzzy Logic & NeuroFuzzy Applications Explained, p. 350. Prentice Hall PTR, New Jersey (1995)
29. Lin, C.T., Lee, C.S.: Neural Fuzzy Systems. Prentice Hall PTR, New York (1996)
30. Nauck, D., Klawonn, F., Kruse, R.: Foundation of Neuro–Fuzzy Systems, p. 305. John Willey & Sons, New York (1997)
31. Bishop, C.M.: Neural Networks for Pattern Recognition, p. 482. Oxford University Press (2002)