NEURO FUZZY APPLICATIONS IN CIVIL ENGINEERING (4-0-0)

Module I

Introduction: Basic concepts of Neural Networks and Fuzzy Logic, Differences between conventional computing and Neuro-Fuzzy computing, Characteristics of Neuro-Fuzzy computing

Fuzzy Set Theory: Basic definitions and terminology and membership functions – Formulation and parameters, basic operations of fuzzy sets – complement, intersection vision, T-norm and T-conorm

Introduction

Fuzzy Sets and Fuzzy Logic

Fuzzy sets were introduced by Zadeh in 1965 to represent/manipulate data and information possessing non-statistical uncertainties.

It was specifically designed to mathematically represent uncertainty and vagueness and to provide formalized tools for dealing with the imprecision intrinsic to many problems. However, the story of fuzzy logic started much more earlier . To devise a concise theory of logic, and later mathematics, Aristotle posited the so-called"Laws of Thought".

One of these, the "Law of the Excluded Middle," states that every proposition must either be True (**T**) or False (**F**). Even when Parminedes proposed the first version of this law (around 400 Before Christ) there were strong and immediate objections: for example, Heraclitus proposed that things could be simultaneously True and not True.

It was Plato who laid the foundation for what would become fuzzy logic, indicating that there was a third region (beyond **T** and **F**) where these opposites "tumbled about." A systematic alternative to the bi-valued logic of Aristotle was first proposed by Łukasiewicz around 1920, when he described a three-valued logic, along with the mathematics to accompany it. The third value he proposed can best be translated as the term "possible," and he assigned it a numeric value between **T** and **F**. Eventually, he proposed an entire notation and axiomatic system from which he hoped to derive modern mathematics.

Later, he explored four-valued logics, five-valued logics, and then declared that in principle there was nothing to prevent the derivation of an infinite-valued logic. Łukasiewicz felt that three- and

infinite-valued logics were the most intriguing, but he ultimately settled on a four-valued logic because it seemed to be the most easily adaptable to Aristotelian logic.

It should be noted that Knuth also proposed a three valued logic similar to Lukasiewicz's, from which he speculated that mathematics would become even more elegant than in traditional bi-valued logic.

The notion of an infinite-valued logic was introduced in Zadeh's seminal work "Fuzzy Sets" where he described the mathematics of fuzzy set theory, and by extension fuzzy logic. This theory proposed making the membership function (or the values **F** and **T**) operate over the range of real numbers [0, 1]. New operations for the calculus of logic were proposed, and showed to be in principle at least a generalization of classic logic.

Fuzzy logic provides an inference morphology that enables approximate human reasoning capabilities to be applied to knowledge-based systems. The theory of fuzzy logic provides a mathematical strength to capture the uncertainties associated with human cognitive processes, such as thinking and reasoning.

The conventional approaches to knowledge representation lack the means for representating the meaning of fuzzy concepts. As a consequence, the approaches based on first order logic and classical probability theory do not provide an appropriate conceptual framework for dealing with the representation of commonsense knowledge, since such knowledge is by its nature both lexically imprecise and non-categorical.

The developement of fuzzy logic was motivated in large measure by the need for a conceptual frame-work which can address the issue of uncertainty and lexical imprecision.

Some of the essential characteristics of fuzzy logic relate to the following (Zadeh, 1992):

• In fuzzy logic, exact reasoning is viewed as a limiting case of approximate reasoning.

• In fuzzy logic, everything is a matter of degree.

• In fuzzy logic, knowledge is interpreted a collection of elastic or, equivalently, fuzzy constraint on a collection of variables.

• Inference is viewed as a process of propagation of elastic constraints.

• Any logical system can be fuzzified.

There are two main characteristics of fuzzy systems that give them better performance for specific applications.

• Fuzzy systems are suitable for uncertain or approximate reasoning, especially for the system with a mathematical model that is difficult to derive.

• Fuzzy logic allows decision making with estimated values under incomplete or uncertain information.

An assertion can be more or less true in fuzzy logic. In classical logic an assertion is either true or false — not something in between — and fuzzy logic extends classical logic by allowing intermediate truth values between zero and one. Fuzzy logic enables a computer to interpret a linguistic statement such as'if the washing machine is half full, then use less water.'

It adds intelligence to the washing machine since the computer infers an action from a set of such if-then rules. Fuzzy logic is 'computing with words,' quoting the creator of fuzzy logic,

Lotfi A. Zadeh.

The objective of this tutorial is to explain the necessary and sufficient parts of the theory, such that engineering students understand how fuzzy logic enables fuzzy reasoning by computers.

Fuzzy Set Theory

Fuzzy sets are a further development of mathematical set theory, first studied formally by the German mathematician Georg Cantor (1845-1918). It is possible to express most of mathematics in the language of set theory, and researchers are today looking at the

consequences of 'fuzzifying' set theory resulting in for example fuzzy logic, fuzzy numbers, fuzzy intervals, fuzzy arithmetic, and fuzzy integrals. Fuzzy logic is based on fuzzy sets, and with fuzzy logic a computer can process words from natural language, such as 'small', 'large', and 'approximately equal'.

Although elementary, the following sections include the basic definitions of classical set theory. This is to shed light on the original ideas, and thus provide a deeper understanding.

But only those basic definitions that are necessary and sufficient will be presented; students interested in delving deeper into set theory and logic, can for example read the book by Stoll (1979[10]); it provides a precise and comprehensive treatment .

Fuzzy Sets

According to Cantor a set X is a collection of definite, distinguishable objects of our intuition which can be treated as a whole. The objects are the members of X. The concept 'objects of our intuition' gives us great freedom of choice, even sets with infinitely many members. Objects must be 'definite': given an object and a set, it must be possible to determine whether the object

is, or is not, a member of the set. Objects must also be 'distinguishable': given a set and its members, it must be possible to determine whether any two members are different, or the same.

The members completely define a set. To determine membership, it is necessary that the sentence 'x is a member of X', where x is replaced by an object and X by the name of a set, is either true or false. We use the symbol $\in$ and write $x \in X$ if object x is a member of the set X. The assumption that the members determine a set is equivalent to saying: Two sets X and Y are equal, X = Y, iff (if and only if) they have the same members. The set whose members are the objects $x1, x2, \ldots, xn$ is written $\{x1, x2, \ldots, xn\}$.

In particular, the set with no members is the empty set symbolized by $\emptyset$. The set X is included in Y, $X \subseteq Y$ iff each member of X is a member of Y. We also say that X is a subset of Y, and it means that, for all x, if $x \in X$, then $x \in Y$. The empty set is a subset of every set. Almost anything called a set in ordinary conversation is acceptable as a mathematical set, as the next example indicates.

Example 1 Classical sets

The following are lists or collections of definite and distinguishable objects, and therefore sets in the mathematical sense.

(a) The set of non-negative integers less than 3. This is a finite set with three members {0, 1, 2}.

(b) The set of live dinosaurs in the basement of the British Museum. This set has no members, it is the empty set $\emptyset$.

(c) The set of measurements greater than 10 volts. Even though this set is infinite, it is possible to determine whether a given measurement is a member or not.

(d) The set {0, 1, 2} is the set from (a). Since {0, 1, 2} and {2, 1, 0} have the same members, they are equal sets. Moreover, {0, 1, 2} = {0, 1, 1, 2} for the same reason.

(e) The members of a set may themselves be sets. The set X = {{1, 3} , {2, 4}, {5, 6}}is a set with three members, namely, {1, 3} , {2, 4}, and {5, 6} . Matlab supports sets of sets, or nested sets, in cell arrays. The notation in Matlab for assigning the above sets to a cell array x is the same.

(f) It is possible in Matlab to assign an empty set, for instance: x = {[]}.

Although the brace notation {·} is practical for listing sets of a few elements, it is impractical for large sets and impossible for infinite sets. How do we then define a set with a large number of members?

An answer requires a few more concepts. A proposition is an assertion (declarative statement) which can be classified as either true or false. By a predicate in x we understand an assertion formed using a formula in x. For instance, ’0 < x ≤ 3’, or ’x > 10 volts’ are predicates. They are not propositions, however, since they are not necessarily true or false. Only if we assign a value to the variable x, each predicate becomes a proposition. A predicate P (x) in x defines a set X by the convention that the members of X are exactly those objects such that P (a) is true. In mathematical notation: {x | P (x)} , read ’the set of all x such that P (x).’ Thus a ∈ {x | P (x)} iff P (a) is a true proposition.

A system in which propositions must be either true or false, but not both, uses a two-valued logic. As a consequence, what is not true is false and vice versa; that is the law of the excluded middle. This is only an approximation to human reasoning, as Zadeh observed:

Clearly, the "class of all real numbers which are much greater than 1," or "the class of beautiful women," or "the class of tall men," does not constitute classes or sets in the usual mathematical sense of these terms. (Zadeh, 1965[12])

Zadeh’s challenge we might call it, because for instance ’tall’ is an elastic property. To define the set of tall men as a classical set, one would use a predicate P (x), for instance x ≥ 176, where x is the height of a person, and the right hand side of the inequality a threshold value in centimeters (176 centimeters ' 5 foot 9 inches). This is an abrupt approximation to the concept ’tall’. From an engineering viewpoint, it is likely that the measurement is uncertain, due to a source of noise in the equipment. Thus, measurements within the narrow band 176±ε, where ε expresses variation in the noise, could fall on either side of the threshold randomly.

Following Zadeh a membership grade allows finer detail, such that the transition from membership to non-membership is gradual rather than abrupt. The membership grade for all members defines a fuzzy set (Fig. 1). Given a collection of objects U, a fuzzy set A in U is defined as a set of ordered pairs

$$\mathcal{A} \equiv \{\langle x, \mu_{\mathcal{A}}(x)\rangle \mid x \in \mathcal{U}\} \qquad (1)$$

where μA (x) is called the membership function for the set of all objects x in U — for the symbol ’≡’ read ’defined as’. The membership function relates to each x a membership grade μA (x), a real number in the closed interval [0, 1]. Notice it is now necessary to work with pairs hx, μA (x)i whereas for classical sets a list of objects suffices, since their membership is understood. An

ordered pair hx, yi is a list of two objects, in which the object x is considered first and y second (note: in the set {x, y} the order is insignificant).

The term ’fuzzy’ (indistinct) suggests an image of a boundary zone, rather than an abrupt frontier. Indeed, fuzzy logicians speak of classical sets being crisp sets, to distinguish them from fuzzy sets. As with crisp sets, we are only guided by intuition in deciding which objects are members and which are not; a formal basis for how to determine the membership grade of a fuzzy set is absent. The membership grade is a precise, but arbitrary measure: it rests on personal opinion, not reason.

The definition of a fuzzy set extends the definition of a classical set, because membership values μ are permitted in the interval $0 \leq \mu \leq 1$, and the higher the value, the higher the membership. A classical set is consequently a special case of a fuzzy set, with membership values restricted to $\mu \in \{0, 1\}$.

A single pair hx, μ(x)i is a fuzzy singleton; thus the whole set can be viewed as the union of its constituent singletons.

Example 2 Fuzzy sets

The following are sets which could be described by fuzzy membership functions.

(a) The set of real numbers x À 1 (x much greater than one).

(b) The set of high temperatures, the set of strong winds, or the set of nice days are fuzzy sets in weather reports.

(c) The set of young people. A one year old baby will clearly be a member of the set of young people, and a 100-year-old person will not be a member of this set. A person aged 30 might be young to the degree 0.5.

(d) The set of adults. The Danish railways allow children under the age of 15 to travel at half price. An adult is thus defined by the set of passengers aged 15 or older. By their definition the set of adults is a crisp set.

(e) A predicate may be crisp, but perceived as fuzzy: a speed limit of 60 kilometres per hour is by some drivers taken to be an elastic range of more or less acceptable speeds within, say, 60 − 70 kilometres per hour (' 37 − 44 miles per hour). Notice how, on the one hand, the traffic law is crisp while, on the other hand, those drivers’s understanding of the law is fuzzy.

Members of a fuzzy set are taken from a universe of discourse, or universe for short. The universe is all objects that can come into consideration, confer the set U in (1). The universe depends on the context, as the next example shows.

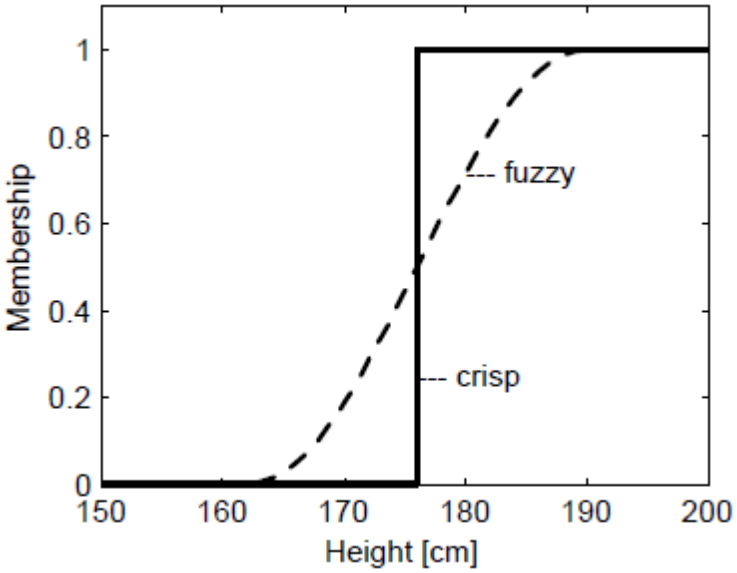


Figure 1. Two definitions of the set of "tall men", a crisp set and a fuzzy set

Example 3 Universes

(a) The set x À 1 could have as a universe all real numbers, alternatively all positive integers.

(b) The set of young people could have all human beings in the world as its universe.

Alternatively it could have the numbers between 0 and 100; these would then represent age in years.

(c) The universe depends on the measuring unit; a duration in time depends on whether it is measured in hours, days, or weeks.

(d) A non-numerical quantity, for instance taste, must be defined on a psychological continuum; an example of such a universe is U = {bitter, sweet, sour, salt, hot}.

A programmer can exploit the universe to suppress faulty measurement data, for instance negative values for a duration of time, by making the program consult the universe.

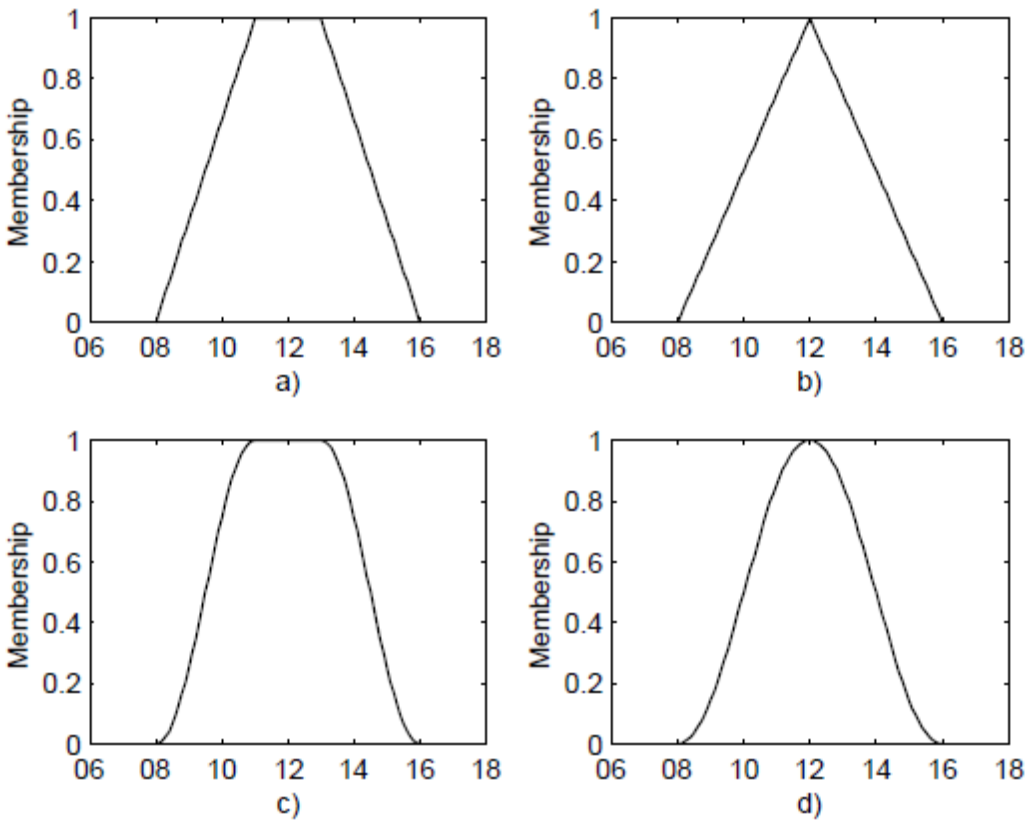


Figure 2. Around noon. Four possible membership functions representing the time 'around noon': a) trapeziodal, b) triangular, c) smooth trapezoid, and d) smooth triangular. The universe is the hours of the day in 24-hour format

There are two alternative ways to represent a membership function: continuous or discrete. A continuous fuzzy setA is defined by means of a continuous membership function μA (x). A trapezoidal membership function is a piecewise linear, continuous function, controlled by four parameters {a, b, c, d} (Jang et al., 1997[4])

$$\mu_{trapezoid}(x; a, b, c, d) = \left\{ \begin{array}{ll} 0 & , x \leq a \\ \frac{x-a}{b-a} & , a \leq x \leq b \\ 1 & , b \leq x \leq c \\ \frac{d-x}{d-c} & , c \leq x \leq d \\ 0 & , d \leq x \end{array} \right\}, x \in \mathbb{R} \qquad (2)$$

The parameters a ≤ b ≤ c ≤ d define four breakpoints, here designated: left foot point (a), left shoulder point (b), right shoulder point (c), and right foot point (d). Figure 2 (a) illustrates a trapezoidal membership function.

A triangular membership function is piecewise linear, and derived from the trapezoidal membership function by merging the two shoulder points into one, that is, setting b = c, Fig. 2 (b).

Smooth, differentiable versions of the trapezoidal and triangular membership functions can be obtained by replacing the linear segments corresponding to the intervals a ≤ x ≤ b and c ≤ x ≤ d by a nonlinear function, for instance a half period of a cosine function,

$$\mu_{STrapezoid}(x; a, b, c, d) = \left\{ \begin{array}{ll} 0 & , x \leq a \\ \frac{1}{2} + \frac{1}{2}\cos(\frac{x-b}{b-a}\pi) & , a \leq x \leq b \\ 1 & , b \leq x \leq c \\ \frac{1}{2} + \frac{1}{2}\cos(\frac{x-c}{d-c}\pi) & , c \leq x \leq d \\ 0 & , d \leq x \end{array} \right\}, x \in \mathbb{R}$$

We call it here S Trapezoid for 'smooth trapezoid' or 'soft trapezoid'. Figures 2 (c-d) illustrate the smooth membership functions. Other possibilities exist for generating smooth trapezoidal functions, for example Gaussian, generalized bell, and sigmoidal membership functions (Jang et al., 1997[4]).

Discrete fuzzy sets are defined by means of a discrete variable xi (i = 1, 2, . . .). A discrete fuzzy set A is defined by ordered pairs,

$$\mathcal{A} = \{\langle x_1, \mu(x_1)\rangle, \langle x_2, \mu(x_2)\rangle, \ldots \mid x_i \in \mathcal{U}, i = 1, 2, \ldots\}$$

Each membership value μ(xi) is an evaluation of the membership function μ at a discrete point xi in the universe U, and the whole set is a collection, usually finite, of pairs hxi, μ(xi)i.

Example 4 Discrete membership function

To achieve a discrete triangular membership function from the trapezoid (2) Assume the universe is a vector u of 7 elements. In Matlab notation,

u=[9 10 11 12 13 14 15]

Assume the defining parameters are a = 10, b = 12, c = 12, and d = 14 then, by (2), the corresponding membership values are a vector of 7 elements, 0 0 0.5 1 0.5 0 0

Each membership value corresponds to one element of the universe, more clearly displayed as

0	0	0.5	1	0.5	0	0
9	10	11	12	13	14	15

with the universe in the bottom row, and the membership values in the top row. When this is impractical, in a program, the universe and the membership values can be kept in separate vectors.

As a crude rule of thumb, the continuous form is more computing intensive, but less storage demanding than the discrete form.

Zadeh writes that a fuzzy set induces a possibility distribution on the universe, meaning, one can interpret the membership values as possibilities. How are then possibilities related to probabilities? First of all, probabilities must add up to one, or the area under a density curve must be one. Memberships may add up to anything (discrete case), or the area under the membership function may be anything (continuous case). Secondly, a probability distribution concerns the likelihood of an event to occur, based on observations, whereas a possibility distribution (membership function) is subjective. The word 'probably' is synonymous with presumably, assumably, doubtless, likely, presumptively. The word 'possible' is synonymous with doable, feasible, practicable, viable, workable. Their relationship is best described in the sentence: what is probable is always possible, but not vice versa. This is illustrated next.

Example 5 Probability versus possibility

a) Consider the statement 'Hans ate x eggs for breakfast', where x ∈ U = h1, 2, ..., 8i (Zadeh in Zimmermann, 1993[16]). We may associate a probability distribution p by observing Hans eating breakfast for 100 days,

.1	.8	.1	0	0	0	0	0
1	2	3	4	5	6	7	8

A fuzzy set expressing the grade of ease with which Hans can eat x eggs may be the following possibility distribution π,

1	1	1	1	.8	.6	.4	.2
1	2	3	4	5	6	7	8

Where the possibility π(3) = 1, the probability p(3) = 0.1 only.

b) Consider a universe of four car models

U = {T rabant, F iat U no, BMW, F errari}.

We may associate a probability p(x) of each car model driving 100 miles per hour (161 kilometres per hour) on a motorway, by observing cars for 100 days, p(T rabant) = 0, p(F iat Uno) = 0.1, p(BMW) = 0.4, p(F errari) = 0.5

The possibilities may be π(T rabant) = 0, π(F iat U no) = 0.5, π(BMW) = 1, π(F errari) = 1

Notice that each possibility is at least as high as the corresponding probability.

Equality and inclusion are defined by means of membership functions. Two fuzzy sets A and B are equal, iff they have the same membership function, $A = B \equiv \mu A(x) = \mu B(x)$ (3) for all x. A fuzzy set A is a subset of (included in) a fuzzy set B, iff the membership of A is less than equal to that of B, $A \subseteq B \equiv \mu A(x) \leq \mu B(x)$ (4) for all x.

Definition 1. (fuzzy set) Let X be a nonempty set. A fuzzy set A in X is characterized by its membership function

$\mu A : X \rightarrow [0, 1]$ and $\mu A(x)$ is interpreted as the degree of membership of element x in fuzzy set A for each $x \in X$.

It is clear that A is completely determined by the set of tuples $A = \{(u, \mu A(u)) | u \in X\}$.

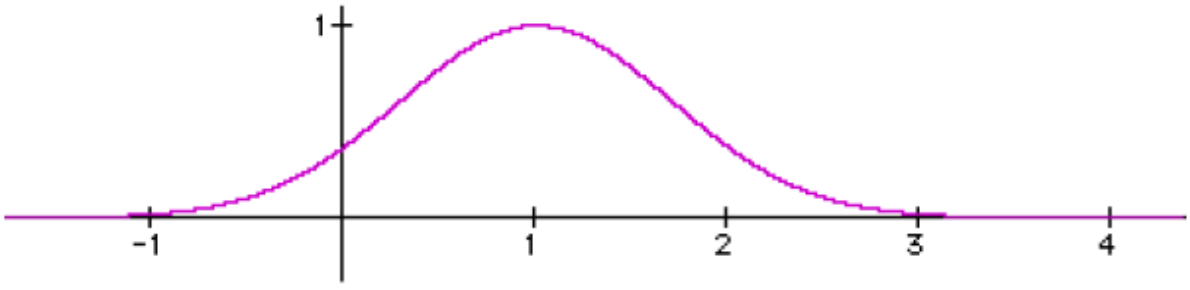
Frequently we will write A(x) instead of $\mu A(x)$. The family of all fuzzy sets in X is denoted by F(X).

If $X = \{x1, \ldots, xn\}$ is a finite set and A is a fuzzy set in X then we often use the notation $A = \mu 1/x1 + \cdot \cdot \cdot + \mu n/xn$ where the term $\mu i/xi$, $i = 1, \ldots, n$ signifies that μi is the grade of membership of xi in A and the plus sign represents the union.

A discrete membership function for ”x is close to 1”.

Example 6. The membership function of the fuzzy set of real numbers” close to 1”, is can be defined as $A(t) = \exp(-\beta(t - 1)2)$ where β is a positive real number.

A membership function for ”x is close to 1”.



A membership function for ”x is close to 1”.

Example 7. Assume someone wants to buy a cheap car. Cheap can be represented as a fuzzy set on a universe of prices, and depends on his purse. For instance, from the Figure cheap is roughly interpreted as follows:

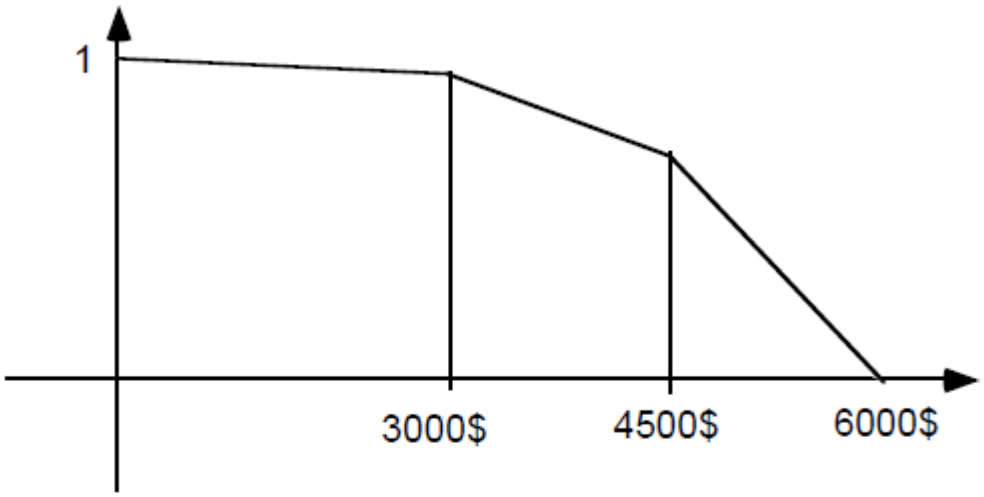
• Below 3000$ cars are considered as cheap, and prices make no real difference to buyer's eyes.

• Between 3000$ and 4500$, a variation in the price induces a weak preference in favor of the cheapest car.

• Between 4500$ and 6000$, a small variation in the price induces a clear preference in favor of the cheapest car.

• Beyond 6000$ the costs are too high (out of consideration).

Membership function of "cheap".



Membership function of "cheap".

Definition 2. (support) Let A be a fuzzy subset of

X; the support of A, denoted supp(A), is the crisp subset of X whose elements all have nonzero membership grades in A.

$\text{supp}(A) = \{x \in X | A(x) > 0\}$.

Definition 3. (normal fuzzy set) A fuzzy subset A of a classical set X is called normal if there exists an $x \in X$ such that $A(x) = 1$. Otherwise A is subnormal.

Definition 4. (α-cut) An α-level set of a fuzzy set A of X is a non-fuzzy set denoted by $[A]\alpha$ and is defined by

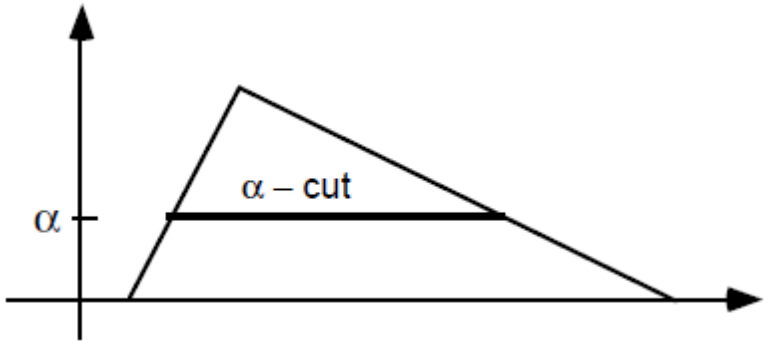
$$[A]^{\alpha} = \begin{cases} \{t \in X | A(t) \geq \alpha\} & if\, \alpha > 0 \\ cl(suppA) & if\, \alpha = 0 \end{cases}$$

where cl(suppA) denotes the closure of the support of A.

Example 8. Assume X = {−2,−1, 0, 1, 2, 3, 4} and A = 0.0/ −2 + 0.3/ − 1 +0.6/0 + 1.0/1 + 0.6/2 + 0.3/3 + 0.0/4, in this case

$$[A]^{\alpha} = \begin{cases} \{-1, 0, 1, 2, 3\} & if\ 0 \leq \alpha \leq 0.3 \\ \{0, 1, 2\} & if\ 0.3 < \alpha \leq 0.6 \\ \{1\} & if\ 0.6 < \alpha \leq 1 \end{cases}$$

Definition 5. (convex fuzzy set) A fuzzy set A of X is called convex if [A]α is a convex subset of X ∀α ∈ [0, 1].



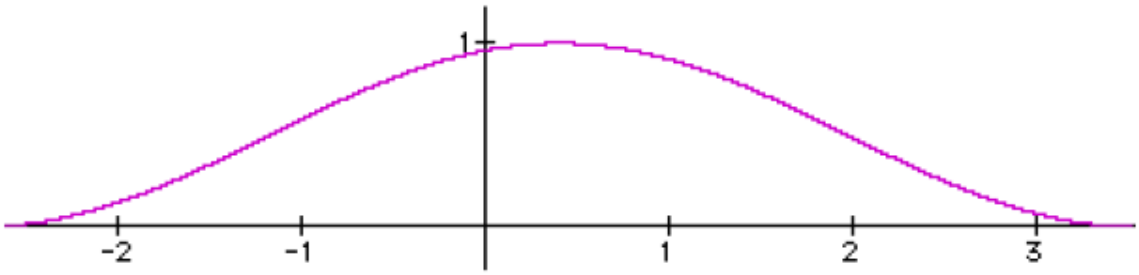
An α-cut of a triangular fuzzy number.

In many situations people are only able to characterize numeric information imprecisely. For example, people use terms such as, about 5000, near zero, or essentially bigger than 5000. These are examples of what are called fuzzy numbers. Using the theory of fuzzy subsets we can represent these fuzzy numbers as fuzzy subsets of the set of real numbers. More exactly,

Definition 6. (fuzzy number) A fuzzy number A is a fuzzy set of the real line with a normal, (fuzzy) convex and continuous membership function of bounded support. The family of fuzzy numbers will be denoted by F.

Definition 7. (quasi fuzzy number) A quasi fuzzy number A is a fuzzy set of the real line with a normal, fuzzy convex and continuous membership function satisfying the limit conditions

$$\lim_{t\to\infty} A(t) = 0, \qquad \lim_{t\to-\infty} A(t) = 0.$$



Fuzzy number.

Fuzzy number.

Let A be a fuzzy number. Then $[A]\gamma$ is a closed convex (compact) subset of R for all $\gamma \in [0, 1]$.

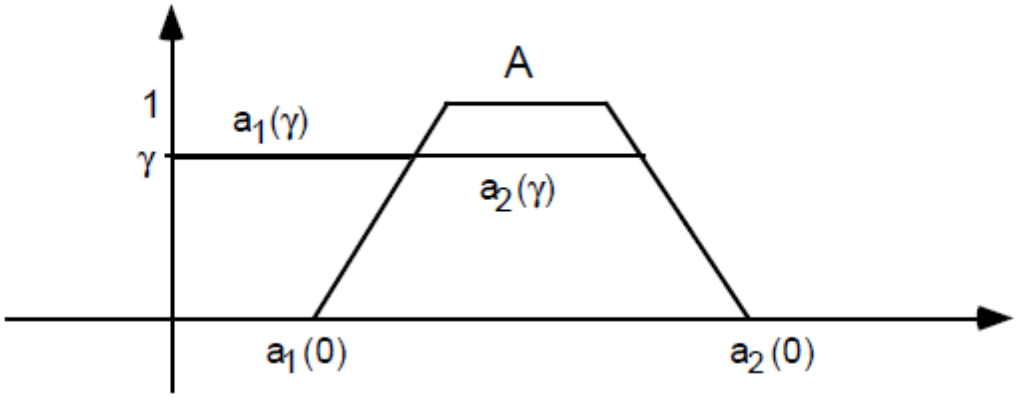
Let us introduce the notations $a1(\gamma) = \min[A]\gamma$, $a2(\gamma) = \max[A]\gamma$

In other words, $a1(\gamma)$ denotes the left-hand side and $a2(\gamma)$ denotes the right-hand side of the γ-cut.

It is easy to see that If $\alpha \leq \beta$ then $[A]\alpha \supset [A]\beta$

Furthermore, the left-hand side function $a1: [0, 1] \rightarrow R$ is monoton increasing and lower semi-continuous, and the right-hand side function $a2: [0, 1] \rightarrow R$ is monoton decreasing and upper semi-continuous.

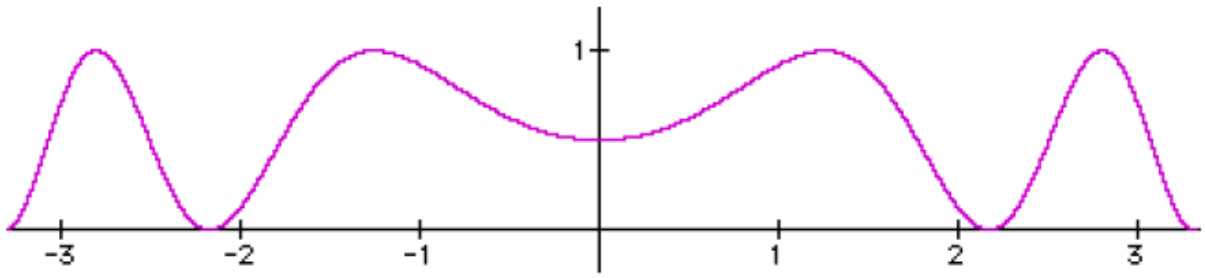
We shall use the notation $[A]\gamma = [a1(\gamma), a2(\gamma)]$.



The support of A is the open interval (a1(0), a2(0)).

The support of A is (a1(0), a2(0)).

If A is not a fuzzy number then there exists an $\gamma \in [0, 1]$ such that $[A]\gamma$ is not a convex subset of R.



Not fuzzy number.

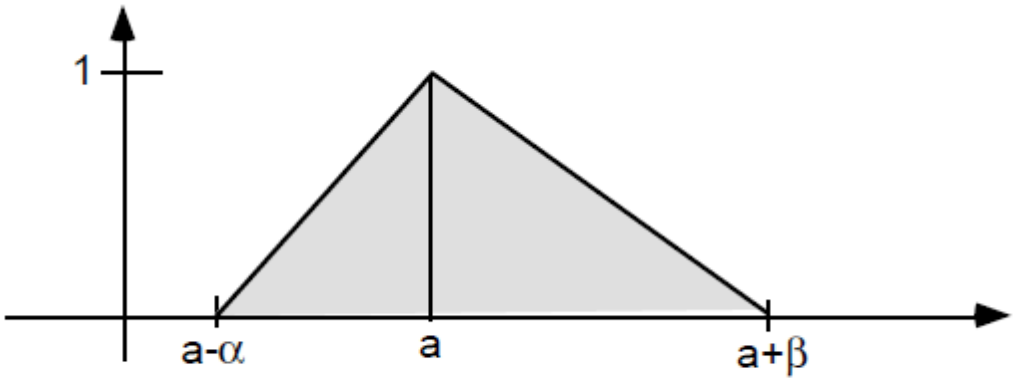
Not fuzzy number.

Definition 8. (triangular fuzzy number) A fuzzy set

A is called triangular fuzzy number with peak (or center) a, left width $\alpha > 0$ and right width $\beta > 0$ if its membership function has the following form and we use the notation $A = (a, \alpha, \beta)$. It can easily be verified that $[A]\gamma = [a - (1 - \gamma)\alpha, a + (1 - \gamma)\beta], \forall\gamma \in [0, 1]$.

The support of A is $(a - \alpha, b + \beta)$.

$$A(t) = \begin{cases} 1 - \dfrac{a - t}{\alpha} & if\ a - \alpha \leq t \leq a \\ 1 - \dfrac{t - a}{\beta} & if\ a \leq t \leq a + \beta \\ 0 & otherwise \end{cases}$$



Triangular fuzzy number.

Triangular fuzzy number.

A triangular fuzzy number with center a may be seen as a fuzzy quantity

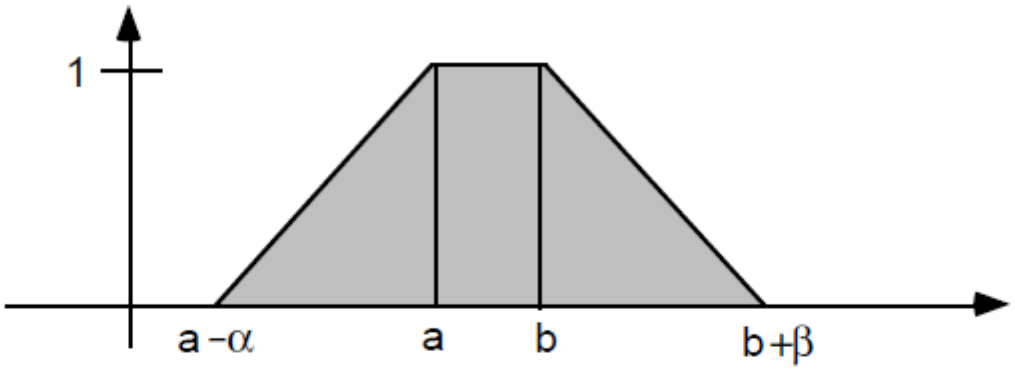
”x is approximately equal to a”.

Definition 9. (trapezoidal fuzzy number) A fuzzy set

A is called trapezoidal fuzzy number with tolerance interval [a, b], left width α and right width β if its membership function has the following form and we use the notation A = (a, b, α, β). It can easily be shown that $[A]_\gamma = [a - (1 - \gamma)\alpha, b + (1 - \gamma)\beta]$, $\forall \gamma \in [0, 1]$.

The support of A is $(a - \alpha, b + \beta)$.

$$A(t) = \begin{cases} 1 - (a - t)/\alpha & if\ a - \alpha \leq t \leq a \\ 1 & if\ a \leq t \leq b \\ 1 - (t - b)/\beta & if\ a \leq t \leq b + \beta \\ 0 & otherwise \end{cases}$$



Trapezoidal fuzzy number.

Trapezoidal fuzzy number.

A trapezoidal fuzzy number may be seen as a fuzzy quantity ”x is approximately in the interval [a, b]”.

Definition 10. Any fuzzy number A ∈ F can be described as

$$A(t) = \begin{cases} L\left(\dfrac{a-t}{\alpha}\right) & if\, t \in [a-\alpha, a] \\ 1 & if\, t \in [a, b] \\ R\left(\dfrac{t-b)}{\beta}\right) & if\, t \in [b, b+\beta] \\ 0 & otherwise \end{cases}$$

where [a, b] is the peak or core of A,

L: [0, 1] → [0, 1], R: [0, 1] → [0, 1], are continuous and non-increasing shape functions with L(0) = R(0) = 1 and R(1) = L(1) = 0. We call this fuzzy interval of LR-type and refer to it by A = (a, b, α, β) LR.

The support of A is (a − α, b + β).

Let A = (a, b, α, β)LR be a fuzzy number of type LR. If a = b then we use the notation A = (a, α, β)LR and say that A is a quasi-triangular fuzzy number.

Furthermore if L(x) = R(x) = 1 − x then instead of A = (a, b, α, β) LR we simply write A = (a, b, α, β).

Nonlinear and linear reference functions.

Definition 11. (subset hood) Let A and B are fuzzy subsets of a classical set X. We say that A is a subset of B if $A(t) \leq B(t)$, $\forall t \in X$.

A is a subset of B.

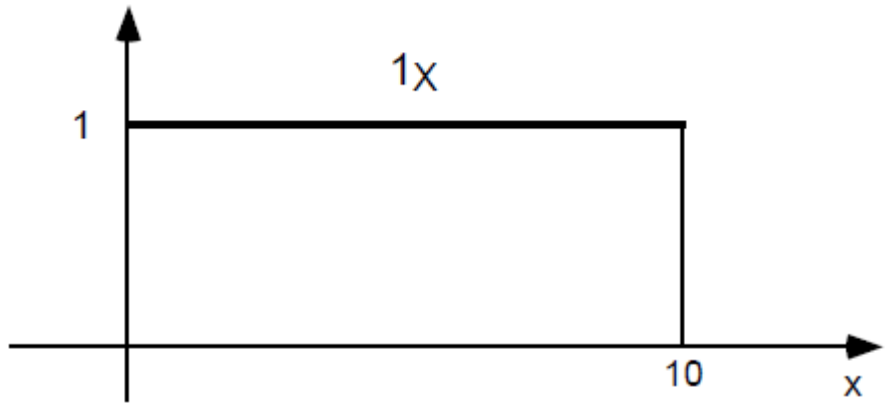
Definition 12. (equality of fuzzy sets) Let A and B are fuzzy subsets of a classical set X. A and B are said to be equal, denoted A = B, if $A \subset B$ and $B \subset A$. We note that A = B if and only if A(x) = B(x) for $x \in X$.

Definition 13. (empty fuzzy set) The empty fuzzy subset of X is defined as the fuzzy subset ∅ of X such that ∅(x) = 0 for each $x \in X$.

It is easy to see that $\emptyset \subset A$ holds for any fuzzy subset A of X.

Definition 14. The largest fuzzy set inX, called universal fuzzy set in X, denoted by 1X, is defined by 1X(t) = 1, $\forall t \in X$.

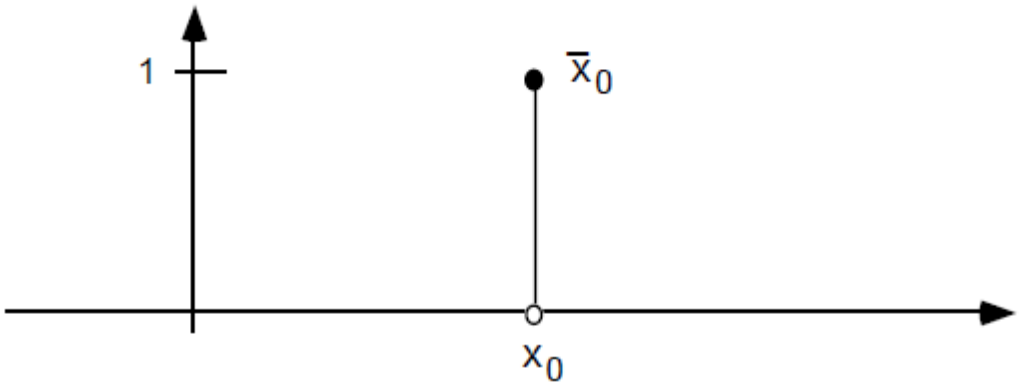
It is easy to see that $A \subset 1X$ holds for any fuzzy subset A of X.



The graph of the universal fuzzy subset in X = [0, 10].

Definition 15. (Fuzzy point) Let A be a fuzzy number.

If supp(A) = {x0} then A is called a fuzzy point and we use the notation A = ¯x0.



Fuzzy point.

Let A = $\bar{}$x0 be a fuzzy point. It is easy to see that

$[A]_\gamma = [x_0, x_0] = \{x_0\}, \forall \gamma \in [0, 1]$.

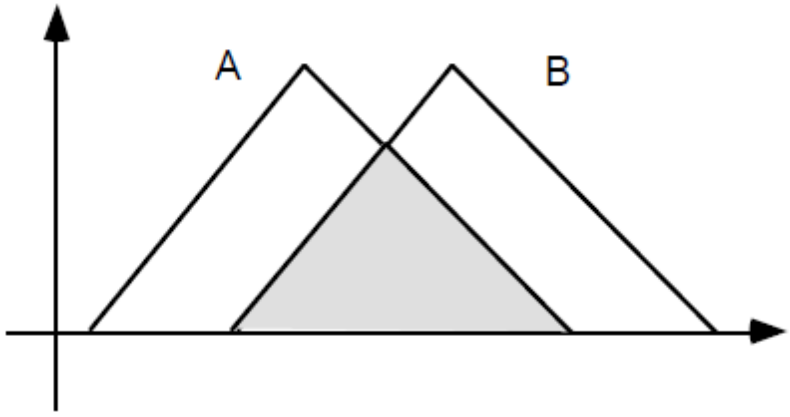
Operations on fuzzy sets

We extend the classical set theoretic operations from ordinary set theory to fuzzy sets. We note that all those operations which are extensions of crisp concepts reduce to their usual meaning when the fuzzy subsets have membership degrees that are drawn from {0, 1}. For this reason, when extending operations to fuzzy sets we use the same symbol as in set theory.

Let A and B are fuzzy subsets of a nonempty (crisp) set X.

Definition 16. (intersection) The intersection of A and B is defined as

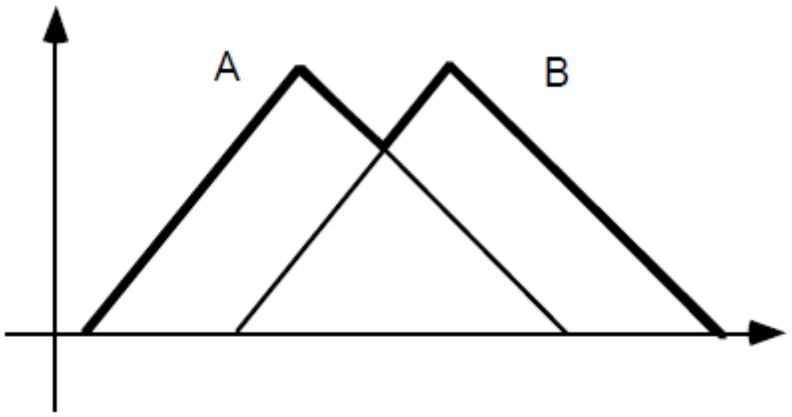
$(A \cap B)(t) = \min\{A(t),B(t)\} = A(t) \wedge B(t)$, for all $t \in X$.



Intersection of two triangular fuzzy numbers.

Definition 17. (union) The union of A and B is defined as

$(A \cup B)(t) = \max\{A(t),B(t)\} = A(t) \vee B(t)$, for all $t \in X$.



Union of two triangular fuzzy numbers.

Definition 18. (complement) The complement of a fuzzy set A is defined as (¬A)(t) = 1 − A(t)

A closely related pair of properties which hold in ordinary set theory are the law of excluded middle

A∨¬A = X

and the law of non-contradiction principle

A ∧¬A = ∅

It is clear that ¬1X = ∅ and ¬∅ = 1X, however, the laws of excluded middle and non-contradiction are not satisfied in fuzzy logic.

Lemma 1. The law of excluded middle is not valid.

Let A(t) = 1/2, ∀t ∈ R, then it is easy to see that

(¬A ∨ A)(t) = max{¬A(t),A(t)} = max{1 − 1/2, 1/2} =

1/2 _= 1.

Lemma 2. The law of non-contradiction is not valid.

Let A(t) = 1/2, ∀t ∈ R, then it is easy to see that

(¬A ∧ A)(t) = min{¬A(t),A(t)} = min{1 − 1/2, 1/2} =

1/2 _= 0.

However, fuzzy logic does satisfy De Morgan's laws

¬(A ∧ B) = ¬A ∨¬B, ¬(A ∨ B) = ¬A∧¬B.

Fuzzy Set Operations

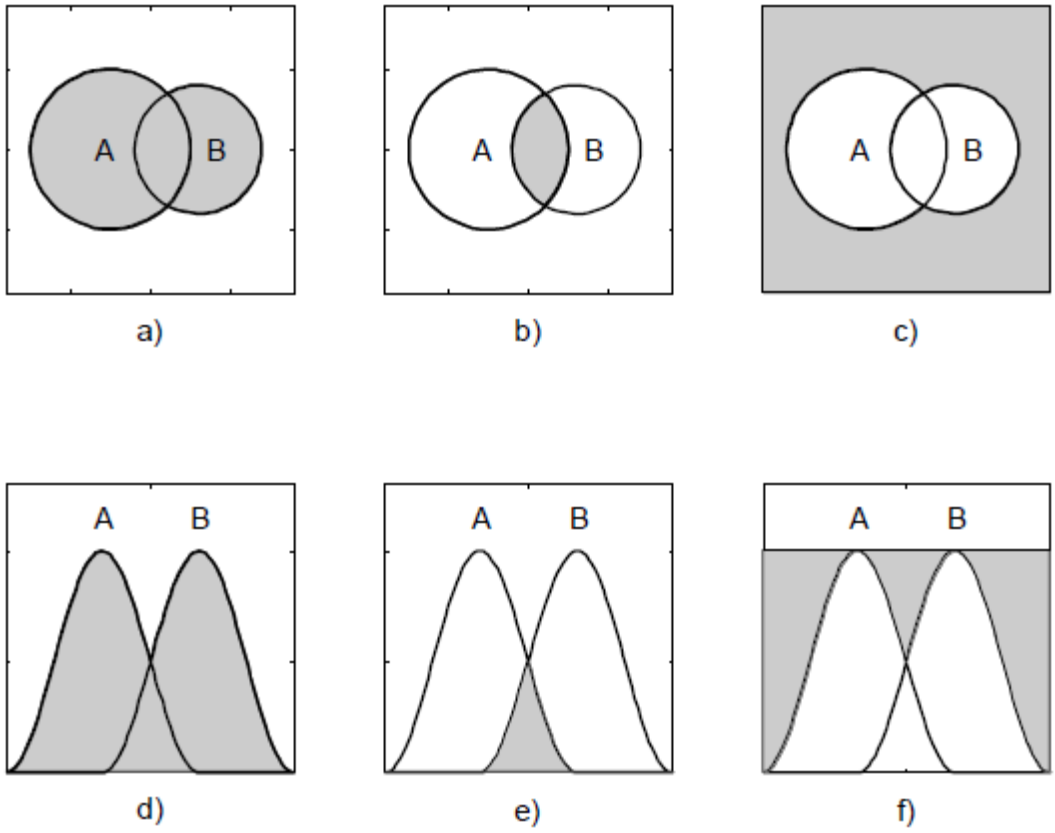


Figure 3. Set operations. The top row are classical Venn diagrams; the universe is represented by the points within the rectangle, and sets by the interior of the circles. The bottom row their fuzzy equivalents; the universal set is represented by a horisontal line at membership $\mu = 1$, and sets by membership functions. The shaded areas are: a) and d) union $A \cup B$, b) and e) intersection $A \cap B$, and c) and f) complement $A \cup B$.

In order to generate new sets from existing sets we define two operations, in certain respects analogous to addition and multiplication. The (classical) union of the sets X and Y, symbolized by $X \cup Y$ and read 'X union Y', is the set of all objects which are members of X or Y, or both. That is,

$X \cup Y \equiv \{x \mid x \in X \text{ or } x \in Y\}$ Thus, by definition, $x \in X \cup Y$ iff x is a member of at least one of X and Y. For example,

$\{1, 2, 3\} \cup \{1, 3, 4\} = \{1, 2, 3, 4\}$

The (classical) intersection of the sets X and Y, symbolized by $X \cap Y$ and read 'X intersection Y', is the set of all objects which are members of both X and Y. That is,

$X \cap Y \equiv \{x \mid x \in X \text{ and } y \in Y\}$

For example,

$\{1, 2, 3\} \cap \{1, 3, 4\} = \{1, 3\}$ The (classical) complement of a set X, symbolized by X and read 'the complement of X' is

$X \equiv \{x \mid x / \in X\}$

That is, the set of those members of the universe which are not members ($/ \in$) of X. Venn diagrams clearly illustrate the set operations, Fig. 3 (a-c).

When turning to fuzzy sets, the gradual membership complicates matters. Figure 3 (d-f) shows an intuitively acceptable modification of the Venn diagrams. The following fuzzy set operations are defined accordingly:

Let A and B be fuzzy sets defined on a mutual universe U. The fuzzy union of A and B is

$A \cup B \equiv \{hx, \mu A \cup B\ (x)i \mid x \in U \text{ and } \mu A \cup B\ (x) = \max(\mu A\ (x)\ ,\ \mu B\ (x))\}$ The fuzzy intersection of A and B is

$A \cap B \equiv \{hx, \mu A \cap B\ (x)i \mid x \in U \text{ and } \mu A \cap B\ (x) = \min(\mu A\ (x)\ ,\ \mu B\ (x))\}$

The fuzzy complement of A is

$A \equiv \{hx, \mu A\ (x)i \mid x \in U \text{ and } \mu A\ (x) = 1 - \mu A\ (x)\}$

While the notation may look cumbersome, it is in practice easy to apply the fuzzy set operations max, min, and $1 - \mu$.

Example 6 Buying a house (after Zimmermann, 1993)

A four-person family wishes to buy a house. An indication of their level of comfort is the number of bedrooms in the house. But they also wish a large house. The universe U =

h1, 2, 3, 4, 5, 6, 7, 8, 9, 10i is the set of houses to be considered described by their number of bedrooms. The fuzzy set Comfortable may be described as a vector c, or in Matlab

c =[0.2 0.5 0.8 1 0.7 0.3 0 0 0 0]

Let l describe the fuzzy set Large, defined as

l =[0 0 0.2 0.4 0.6 0.8 1 1 1 1]

The intersection of Comfortable and Large is then min(c,l),

0 0 0.2 0.4 0.6 0.3 0 0 0 0

To interpret, five bedrooms is optimal having the largest membership value 0.6. It is, however, not fully satisfactory, since the membership is less than 1. The second best solution is four bedrooms, membership 0.4. If the market is a buyer's market, the family will probably wait until a better offer comes up, thus hoping to achieve full satisfaction (membership 1).

The union of Comfortable and Large is max(c,l)

0.2 0.5 0.8 1 0.7 0.8 1 1 1 1

Here four bedrooms is fully satisfactory (membership 1) because it is comfortable. Also 7-10 bedrooms are satisfactory, because the house is large. If the market is a seller's market, the family might buy the house, being content that at least one of the criteria is fulfilled.

If the children are about to move away from the family within the next couple of years, the parents may wish to buy a house that is Comfortable and Not Large, or min(c, 1-l)

0.2 0.5 0.8 0.6 0.4 0.2 0 0 0 0

In that case, three bedrooms is satisfactory to the degree 0.8. The example indicates how fuzzy sets can be used for computer aided decision support.

In mathematics the word 'relation' is used in the sense of relationship, for example the predicates: x is less than y, or y is a function of x. A binary relation R is a set of ordered pairs. We may write it xRy which reads: 'x is related to y.' There are established symbols for various relations, for example x = y, x < y. One simple relation is the set of all pairs

hx, yi, such that x is a member of a set X and y is amember of a set Y. This is the (classical) cartesian product of X and Y,

$$\mathcal{X} \times \mathcal{Y} = \{\langle x, y\rangle \mid x \in \mathcal{X}, y \in \mathcal{Y}\}$$

In fact, any binary relation xRy is a subset of the cartesian product X ×Y, and we can think of those instances of X ×Y, that are members of R as having membership 1.

By analogy, a binary fuzzy relation consists of pairs hx, yi with an associated fuzzy membership value. For example, given X = Y = {1, 2, 3} we can set up a relation 'approximately equal' between all pairs of the three numbers, most clearly displayed in a tabular arrangement,

$\mathcal{X}$ \ $\mathcal{Y}$	1	2	3
1	1	0.8	0.3
2	0.8	1	0.8
3	0.3	0.8	1

In the fuzzy cartesian product each object is defined by a pair: the object, which is a pair itself, and its membership. Let A and B be fuzzy sets defined on X and Y respectively, then the cartesian product A×B is a fuzzy set in X ×Y with the membership function

$$\mathcal{A} \times \mathcal{B} = \left\{ \left\langle \langle x, y \rangle , \mu_{\mathcal{A}\times\mathcal{B}}(x, y) \right\rangle \mid x \in \mathcal{X}, y \in \mathcal{Y}, \mu_{\mathcal{A}\times\mathcal{B}}(x, y) = \min\left(\mu_{\mathcal{A}}(x), \mu_{\mathcal{B}}(y)\right) \right\}$$

For example, assume X and Y are as above, and μA(xi) = h0, 0.5, 1i, with i = 1, 2, 3, and μB(yj) = h1, 0.5, 0i , with j = 1, 2, 3, then A×B is a two-dimensional fuzzy set

		$\mathcal{B}$		
		1	0.5	0
$\mathcal{A}$	0	0	0	0
	0.5	0.5	0.5	0
	1	1	0.5	0

The element at row i and column j is the intersection of μA(xi) and μB(yj). Again we note that to each object hxi, yj i is associated a membership μA×B(xi, yj), whereas the classical cartesian product consists of objects hxi, yji only.

In order to see how to combine relations, let us look at an example from the cartoon Donald Duck.

Example 7 Donald Duck's family

Assume that Donald Duck's nephew Huey resembles nephew Dewey to the grade 0.8, and Huey resembles nephew Louie to the grade 0.9. We have therefore a relation between two subsets of the nephews in the family. This is conveniently represented in a matrix, with one row and two columns (and additional headings),

$\mathbf{R}_1 =$

	Dewey	*Louie*
Huey	*0.8*	*0.9*

Let us assume another relation between nephews Dewey and Louie on the one side, and uncle Donald on the other, a matrix with two rows and one column,

$\mathbf{R}_2 =$

	Donald
Dewey	*0.5*
Louie	*0.6*

We wish to find out how much Huey resembles Donald by combining the information in the two matrices. Observe that

(i) Huey resembles Dewey (R1(1, 1) = 0.8), and Dewey in turn resembles Donald (R2(1, 1) = 0.5), or

(ii) Huey resembles Louie (R1(1, 2) = 0.9), and Louie in turn resembles Donald (R2(2, 1) = 0.6).

Assertion (i) contains two relationships combined by 'and'; it seems reasonable to take the intersection. With our definition, this corresponds to choosing the smallest membership value for the (transitive) Huey-Donald relationship, or min(0.8, 0.5) = 0.5. Similarly with statement (ii). Thus from chains (i) and (ii), respectively, we deduce that

(iii) Huey resembles Donald to the degree 0.5, or

(iv) Huey resembles Donald to the degree 0.6.

Although the results in (iii) and (iv) differ, we are equally confident in either result; we have to choose either one or the other, so it seems reasonable to take the union. With our definition, this corresponds to choosing the largest membership value, or max (0.5, 0.6) = 0.6. Thus, the answer is that Huey resembles Donald to the degree 0.6.

Generally speaking, this was an example of composition of relations. Let R and S be two fuzzy relations defined on X ×Y and Y ×Z respectively. Their composition is a fuzzy

set defined by

$$\mathcal{R} \circ \mathcal{S} = \left\{ \left\langle \langle x, z \rangle, \bigcup_{y} \mu_{\mathcal{R}}(x, y) \cap \mu_{\mathcal{S}}(y, z) \right\rangle \mid x \in \mathcal{X}, y \in \mathcal{Y}, z \in \mathcal{Z} \right\}$$

When R and S are expressed as matrices R and S, the composition is equivalent to an inner product. The inner product is similar to an ordinary matrix (dot) product, except multiplication is replaced by any function and summation by any function. Suppose R is an m × p matrix and S is a p × n matrix. Then the inner ∪ − ∩ product (read 'cup-cap product') is an m × n matrix T = (tij) whose ij-entry is obtained by combining the ith row of R with the jth column of S, such that

$$t_{ij} = (r_{i1} \cap s_{1j}) \cup (r_{i2} \cap s_{2j}) \cup \ldots \cup (r_{ip} \cap s_{pj}) = \bigcup_{k=1}^{p} r_{ik} \cap s_{kj} \qquad (5)$$

With our definitions of the set operations, the composition is specifically called max-min composition (Zadeh in Zimmermann, 1993[16]). Sometimes the min operation is replaced by ***** for multiplication, then it is called max-star composition.

Example 8 Inner product

For the tables R1 and R2 above, the inner product yields,

$$\mathbf{R}_1 \circ \mathbf{R}_2 = \begin{array}{|c|c|}\hline 0.8 & 0.9 \\ \hline \end{array} \circ \begin{array}{|c|}\hline 0.5 \\ \hline 0.6 \\ \hline \end{array} = (0.8 \cap 0.5) \cup (0.9 \cap 0.6) = 0.5 \cup 0.6 = 0.6$$

which agrees with the previous result.

Module II

Fuzzy Reasoning and Fuzzy Inference: Fuzzy relations, Fuzzy rules, Fuzzy reasoning, Fuzzy Inference Systems, Fuzzy modeling, Applications of Fuzzy reasoning and modeling in Civil Engineering Problems.

Fundamental concepts of Artificial Neural Networks: Model of a neuron, activation functions, neural processing, Network architectures, learning methods.

Fuzzy Logic

Logic started as the study of language in arguments and persuasion, and it can be used to judge the correctness of a chain of reasoning — in a mathematical proof for instance. The goal of the theory is to reduce principles of reasoning to a code. The 'truth' or 'falsity' assigned to a proposition is its truth-value. In fuzzy logic a proposition may be true or false, or an intermediate truth-value such as maybe true. The sentence 'John is a tall man' is an example of a fuzzy proposition. For convenience we shall here restrict the possible truth-values to a discrete domain {0, 0.5, 1} for false, maybe true, and true. In doing so we are in effect restricting the theory to multi-valued logic, or rather three-valued logic to be specific. In practice one would subdivide the unit interval into finer divisions, or work with a continuous truth-domain. Nevertheless, much of what follows is valid even in a continuous domain as we shall see.

The idea of fuzzy logic is to approximate human decision-making using natural- language terms instead of quantitative terms. Fuzzy logic is similar to neural networks, and one can create behavioral systems with both methodologies. A good example is the use of fuzzy logic for automatic control: a set of rules or a table is constructed that specifies how an effect is to be achieved, provided input and the current system state. Using fuzzy arithmetic, one uses a model and makes a subset of the system components fuzzy so that fuzzy arithmetic must be used when executing the model. In a broad sense, fuzzy logic refers to fuzzy sets, which are sets with blurred boundaries, and, in a narrow sense, fuzzy logic is a logical system that aims to formalize approximate reasoning. For example, a temperature control system has three settings: cold, moderate, and hot (see Fig. 1).

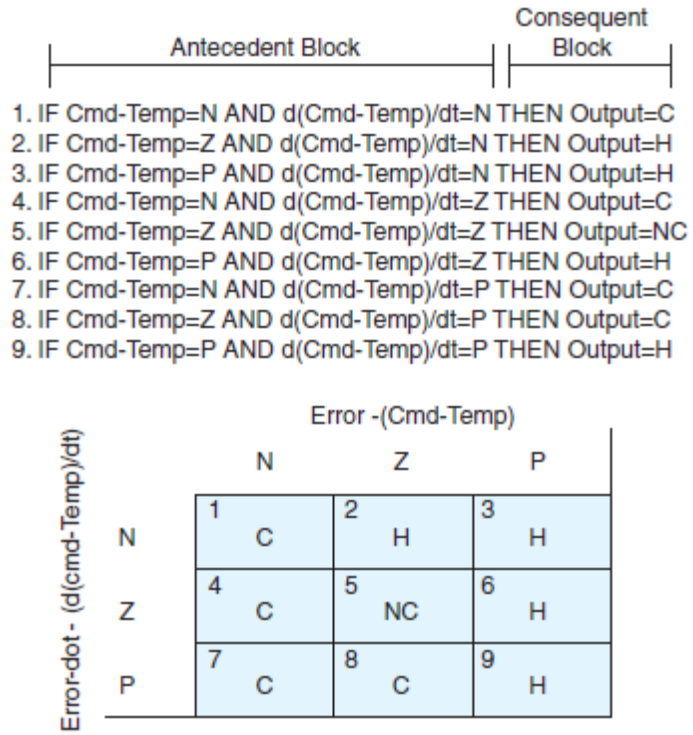


Fig. 1 Rule structure and rule matrix

The first step is to develop a matrix. Because there are three conditions, the matrix will be 3×3. There will also be quite a few variables. These include N for negative, P for positive, and Z for zero. Each represents the possible input error and its derivative, the direction of temperature change; in other words: it is hot, getting hotter, or cold, getting older, or moderate. The variables inside the matrix represent the responses to changing conditions. C represents a "cool" response, H represents a "heat" response, and NC represents a "no change" response. Together, these variables provide nine rules to conditions and their responses depending on varying situations. This is how fuzzy logic "makes decisions." If the temperature is hot (N) and getting hotter (N), then the response should be to turn the cooling feature on the temperature control system (N). This can be implemented very easily by computer hardware, software, or a combination of the two. While this is a very simple example, more practical applications would make up very large matrices and a more complex set of rules. It is key to note, though, that conditional rules can easily be stated linguistically and that conditional statements can use the AND, OR, or NOT operators.

Fuzzy Logic verses Neural Network

Neural networks, however, are a different paradigm for computing. Neural networks process information in a similar way that the human brain does. The network is composed of a large number of highly interconnected processing elements (neurons) working in parallel to solve a specific problem. Neural networks learn by example and cannot be programmed to perform a specific task. Neural networks, with their remarkable ability to derive meaning from complicated or imprecise data, can be used to extract patterns and detect trends that are too complex to be noticed by either humans or other computer techniques.

Neural networks are a form of multiprocessor computer system, with

• simple processing elements

• a high degree of interconnection

• simple scalar messages

• adaptive interaction betweenelements.

A biological neuron may have as many as 10,000 different inputs, and may send its output (the presence or absence of a short-duration spike) to many other neurons. Neurons are wired up in a three-dimensional (3-D) pattern.Real brains, however, are orders of magnitude more complex than any artificial neural network so far considered.

A simple, single-unit adaptive network is shown in Fig. 2. The network has two inputs and one output. All are binary. The output is 1 if $W0 * I0 + W1 * I1 + Wb > 0$, and 0 if $W0 * I0 + W1 * I1 + Wb \leq 0$. We want it to learn simple OR: output a 1 if either I0 or I1 is 1. Conventional computers use an algorithmic approach; i.e., the computer follows a set of instructions in order to solve a problem. Unless the specific steps that the computer needs to follow are known, the computer cannot solve the problem. This restricts the problem-solving capability of conventional computers to problems that we already understand and know how to solve. But computers would be so much more useful if they could do things that we don't exactly know how to do. This is where neural networks come in. Neural network systems help when formulating an algorithmic solution is extremely difficult, lots of examples of the behavior that are required, or there is a need to pick out the structure from existing data.

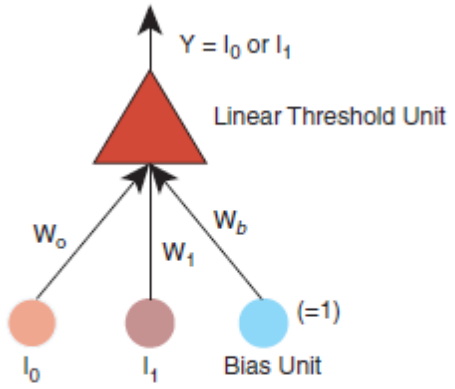


Fig. 2 A simple, single-unit adaptive network

3.1 Propositions

In daily conversation and mathematics, sentences are connected with the words and, or, if-then (or implies), and if and only if. These are called connectives. A sentence which is modified by the word 'not' is called the negation of the original sentence. The word 'and' is used to join two sentences to form the conjunction of the two sentences. The word 'or' is used to join two sentences to form the disjunction of the two sentences. From two sentences we may construct one, of the form 'If ... then ...'; this is called an implication. The sentence following 'If' is the antecedent, and the sentence following 'then' is the consequent. Other
idioms which we shall regard as having the same meaning are 'p implies q', 'p only if q', 'q if p', etc.

Letters and special symbols make the connective structure stand out. Our choice of symbols is

¬ for 'not'

∧ for 'and'

∨ for 'or'

⇒ for 'if-then' (implication)

⇔ for 'if and only if' (equivalence)

The next example illustrates how the symbolic forms expose the underlying logical structure.

Example 9 Baseball betting (Stoll, 1979[10])

Consider the assertion about four baseball teams: If either the Pirates or the Cubs loose and the Giants win, then the Dodgers will be out of first place, and I will loose a bet. Since it is an implication, it may be symbolized in the form r ⇒ s. The antecedent is composed from the three sentences p (The Pirates lose), c (The Cubs lose), and g (The Giants win). The consequent is the

conjunction of d (The Dodgers will be out of first place) and b (I will lose a bet). The original sentence may thus be represented by ((p ∨ c) ∧ g) ⇒ (d ∧ b).

An assertion which contains at least one propositional variable is called a propositional form. The main difference between proposition and propositional form is that every proposition has a truth-value, whereas a propositional form is an assertion whose truth-value cannot be determined until propositions are substituted for its propositional variables. But when no confusion results, we will refer to propositional forms as propositions.

A truth-table summarizes the possible truth-values of an assertion. Take for example the truth-table for the two-valued propositional form p ∨ q. The truth-table (below, left) lists all possible combinations of truth-values — the Cartesian product— of the arguments p and q in the two leftmost columns. The rightmost column holds the truth-values of the proposition.

Alternatively, the truth-table can be rearranged into a two-dimensional array, a so-called Cayley table (below, right).

p	q	$p \vee q$
0	0	0
0	1	1
1	0	1
1	1	1

is equivalent to

Or

$p \vee q$	0	1	$\rightarrow q$
0	0	1	
1	1	1	
$\downarrow p$			

Along the vertical axis in the Cayley table, symbolized by arrow ↓, are the possible values 0 and 1 of the first argument p. Along the horizontal axis, symbolized by arrow →, are the possible values 0 and 1 of the second argument q. Above the table, the proposition p ∨ q reminds us that the table concerns disjunction. At the intersection of row i and column j (only counting the inside of the box) is the truth-value of the expression pi ∨ qj. By inspection, one entry renders p ∨ q false, while three entries render p ∨ q true. Truth-tables for binary connectives are thus given by two-by-two matrices. A total of 16 such tables can be constructed, and each has been associated with a connective.

We can derive the truth-table for 'nand' ('not and') from 'or'. By the definition (¬p) ∨ (¬q) we negate each variable of the previous truth-table, which is equivalent to reversing the axes and permuting the entries back in the usual ascending order on the axes,

Nand

$(\neg p) \vee (\neg q)$

$\downarrow p$ \ $\rightarrow q$	0	1
0	1	1
1	1	0

From ’nand’ (’not and’) we derive ’and’ by negating the entries of the table (’not not and’ = ’and’),

And

$\neg((\neg p) \vee (\neg q))$

$\downarrow p$ \ $\rightarrow q$	0	1
0	0	0
1	0	1

Notice that it was possible to define ’and’ by means of ’or’ and ’not’, rather than assume its definition given as an axiom.

By means of ’or’ and ’not’ we can proceed to define ’implication’. Classical logic defines implication ¬p ∨ q, which is called material implication. We negate the p-axis of the ’or’ table, which is equivalent to reversing the axis and permuting the entries back in the usual ascending order, thus

Implication

$\neg p \vee q$

$\downarrow p$ \ $\rightarrow q$	0	1
0	1	1
1	0	1

Equivalence is taken to mean (p ⇒ q) ∧ (q ⇒ p), which is equivalent to the conjunction of each entry of the implication table with the elements of the transposed table, element by element, or

Equivalence

$(p \Rightarrow q) \wedge (q \Rightarrow p)$

$\downarrow p$ \ $\rightarrow q$	0	1
0	1	0
1	0	1

It is possible to evaluate, in principle at least, a logical proposition by an exhaustive test of all combinations of truth-values of the variables, and this is the idea behind array based logic (Franksen, 1979[1]). The next example illustrates an application of array based logic.

Example 10 Array based logic

In the baseball example, we had the relation ((p ∨ c) ∧ g) ⇒ (d ∧ b). The proposition contains five variables, and each variable can take two truth-values. There are therefore 25 = 32 possible combinations. Only 23 are legal, in the sense that the proposition is true for these combinations, and 32 − 23 = 9 cases are illegal, in the sense that the proposition is false for those combinations. If we are interested only in the legal combinations for which 'I win the bet' (b = 0), then the following table results

p	c	g	d	b
0	0	0	0	0
0	0	0	1	0
0	0	1	0	0
0	0	1	1	0
0	1	0	0	0
0	1	0	1	0
1	0	0	0	0
1	0	0	1	0
1	1	0	0	0
1	1	0	1	0

There are thus 10 winning outcomes out of 32 possible.

By analogy we can define similar truth-tables for fuzzy logic connectives. If we start again by defining negation and disjunction, we can derive the truth-tables of other connectives from that point of departure. Let us define disjunction as set union, that is,

$$p \vee q \equiv \max(p, q). \qquad (6)$$

We can then build the truth-table for the fuzzy connective 'or',

Or

$p \vee q$

p \ q	0	0.5	1
0	0	0.5	1
0.5	0.5	0.5	1
1	1	1	1

→q

↓p

Like before, the p-axis is vertical and the q-axis horizontal. At the intersection of row i and column j (only regarding the inside of the box) is the value of the expression max(pi, qj) in

accordance with (6). When looking for definitions of fuzzy connectives, we will require that such connectives should agree with their classical counterparts for the truth-domain {0, 1}.

In terms of truth-tables, the values in the four corners of the fuzzy Cayley table, should agree with the Cayley table for the classical connective.

Let us define negation as set complement, that is, $\neg p \equiv 1 - p$.

The truth-table for 'nand' is derived from 'or' by the definition $(\neg p) \vee (\neg q)$ by negating the variables. That is equivalent to reversing the axes in the Cayley table. And moving further,

'and' is the negation of the entries in the truth-table for 'nand', thus

Nand

$(\neg p) \vee (\neg q)$

p \ q	0	0.5	1
0	1	1	1
0.5	1	0.5	0.5
1	1	0.5	0

And

$\neg((\neg p) \vee (\neg q))$

p \ q	0	0.5	1
0	0	0	0
0.5	0	0.5	0.5
1	0	0.5	1

(7)

It is reassuring to observe that even though the truth-table for 'and' is derived from the truth-table for 'or', the truth-table for 'and' is identical to one generated using the min operation, set intersection.

The implication, however, has always troubled fuzzy logicians. If we define it as material implication, $\neg p \vee q$, then we get a fuzzy truth-table which is unsuitable, as it causes several useful logical laws to break down. It is important to realize, that we must make a design choice at this point, in order to proceed with the definition of implication and equivalence.

The choice is which logical laws we wish to apply.

Not all laws known from two-valued logic can be valid in fuzzy logic. Take for instance the propositional form

$$p \vee \neg p \Leftrightarrow 1 \qquad (8)$$

which is equivalent to the law of the excluded middle. Testing with the truth-value p = 0.5

(Fuzzy logic) the left hand side yields $0.5 \vee \neg 0.5 = \max(0.5, 1 - 0.5) = 0.5$.

This is different from the right hand side, and thus the law of the excluded middle is invalid in fuzzy logic. If a proposition is true with a truth-value of 1, for any combination of truth-values assigned to the variables, we shall say it is valid. Such a proposition is a tautology. If the proposition is true for some, but not all combinations, we shall say it is satisfiable. Thus (8) is satisfiable, since it is true in two-valued logic, but not in three-valued logic.

One tautology that we definitely wish to apply in fuzzy logic applications is

Tautology 1: $[p \wedge (p \Rightarrow q)] \Rightarrow q$ (9)

Or in words: If p, and p implies q, then q. We have labelled it tautology 1, because we need it later in connection with the modus ponens rule of inference. Another tautology that we definitely wish to apply in fuzzy logic applications is the transitive relationship

Tautology 2: $[(p \Rightarrow q) \wedge (q \Rightarrow r)] \Rightarrow (p \Rightarrow r)$ (10)

Or in words from left to right: if p implies q which in turn implies r, then p implies r. Whether these propositions are valid in fuzzy logic depends on how we define the connectives. Or rather, we must define the connectives, implication in particular, such that those propositions become valid (Jantzen, 1995).

Many researchers have proposed definitions for the implication connective (e.g., Zadeh, 1973; Mizumoto, Fukami & Tanaka, 1979; Fukami, Mizumoto & Tanaka, 1980; Wenstøp, 1980; see also the survey by Lee, 1990). In fact Kiszka, Kochanska & Sliwinska list 72 alternatives to choose from (1985). One candidate, not necessarily the best, is the so-called sharp implication. It can be defined

$p \Rightarrow q \equiv p \leq q$ (11)

This choice is motivated by the following argument. When the crisp set X is included in the crisp set Y, then X is said to imply Y. The fuzzy set A is included in the fuzzy set B, iff $\mu A(x) \leq \mu B(x)$ by (4), therefore we define implication by (11).

Once it is agreed that $p \Leftrightarrow q$ is the same as $(p \Rightarrow q) \wedge (q \Rightarrow p)$, the truth-table for equivalence ($\Leftrightarrow$) is determined from implication and conjunction,

Implication $p \leq q$

p \ $\rightarrow q$	0	0.5	1
0	1	1	1
0.5	0	1	1
1	0	0	1

Equivalence $(p \Rightarrow q) \wedge (q \Rightarrow p)$

p \ $\rightarrow q$	0	0.5	1
0	1	0	0
0.5	0	1	0
1	0	0	1

(12)

Example 11 Fuzzy baseball

The baseball example illustrates what difference three-valued logic makes. The proposition $((p \vee c) \wedge g) \Rightarrow (d \wedge b)$ contains five variables, and now each can take three truth-values. This implies $35 = 243$ possible combinations; 148 of these are legal in the sense that the proposition is true (truth value 1). If we are interested again in the legal combinations for which 'I win the bet'

(b = 0), then there are 33 winning outcomes out of 148. Instead of listing all, we show one for illustration,

(p, c, g, d, b) = (0.5, 0.5, 0, 1, 0) .

With two-valued logic we found 10 winning outcomes out of 32 possible.

The example indicates that fuzzy logic provides more solutions, compared to two-valued logic, and it requires more computational effort.

It is straight forward to test whether tautologies 1 and 2 are valid, because it is possible to perform an exhaustive test of all combinations of truth-values of the variables, provided the domain of truth-values is discrete and limited.

Example 12 Testing tautology 1

Tautology 1 is,

$[p \wedge (p \Rightarrow q)] \Rightarrow q$

Since the proposition contains two variables p and q, and each variable can take three truth-values, there will be 32 = 9 possible combinations to test. The truth-table has 9 rows,

p	q	$p \Rightarrow q$	$[p \wedge (p \Rightarrow q)]$	$[p \wedge (p \Rightarrow q)] \Rightarrow q$
0	0	1	0	1
0	0.5	1	0	1
0	1	1	0	1
0.5	0	0	0	1
0.5	0.5	1	0.5	1
0.5	1	1	0.5	1
1	0	0	0	1
1	0.5	0	0	1
1	1	1	1	1

Columns 1 and 2 are the input combinations of p and q. Column 3 is the result of sharp implication, and column 4 is the left hand side of tautology 1. Since the rightmost column is all 1's, the proposition is a tautology, even in three-valued logic.

Example 12 suggests a new tautology, in fact. If we study closely the truth-values for $[p \wedge (p \Rightarrow q)]$ (column 4 in the example), and compare them with the truth-table for conjunction, we discover they are identical. We can thus postulate,

Tautology 3: $[p \wedge (p \Rightarrow q)] \Leftrightarrow p \wedge q$

We shall use this result later in connection with fuzzy inference.

The main objective of the approach taken here, is not to show that one implication is better than another, but to reduce the proof of any tautology to a test that can be programmed on a computer.

The scope is so far limited to the chosen truth domain {0, 0.5, 1}; this could be extended with intermediate values, however, and the test performed again in case a higher resolution is required. Nevertheless, it suffices to check for all possible combinations of {0, 0.5, 1} if we wish to check the equality of two propositions involving variables connected with ∧, ∨, and ¬ (Gehrke,Walker &Walker, 2003[3]). For propositions involving sharp implication, which cannot be transcribed into a definition involving the said three connectives only, we have to conjecture the results for fuzzy logic.

Since implication can be defined in many possible ways, one has to determine a design criterion first, namely the tautologies, before choosing a proper definition for the implication connective.

Originally, Zadeh interpreted a truth-value in fuzzy logic as a fuzzy set, for instance

Very true (Zadeh, 1988). Thus Zadeh based fuzzy (linguistic) logic on treating Truth as a linguistic variable that takes words or sentences as values rather than numbers (Zadeh,1975). Please be aware that our approach differs, as it is built on scalar truth-values rather than vectors.

Example 13 Mamdani implication

The so-called Mamdani implication, is often used in fuzzy control (Mamdani, 1977[8]). Let

A and B be fuzzy sets defined on X and Y respectively, then the Mamdani implication is a fuzzy set inX ×Y with the membership function

{hhx, yi , μA0⇒0B(x, y)i | x ∈ X, y ∈ Y, μA0⇒0B(x, y) =min(μA(x), μB(y))}

Notice the definition is similar to the definition of the fuzzy cartesian product. Its Cayley table, using the minimum operation, is

Mamdani 'implication'

$\min(p, q)$

p \ q	0	0.5	1
0	0	0	0
0.5	0	0.5	0.5
1	0	0.5	1

One discovers by inspection that only one out of the four corner-values matches the truth table for two-valued implication. Therefore, it is not an implication, and Mamdani 'inference' would be a more appropriate name, as we shall see later.

3.2 Inference

Logic provides principles of reasoning, by means of inference, the drawing of conclusions from assertions. The verb 'to infer' means to conclude from evidence, deduce, or to have as a logical

consequence (do not confuse 'inference' with 'interference'). Rules of inference specify conclusions drawn from assertions known or assumed to be true.

One such rule of inference is modus ponens. It is often presented in the form of an argument:

$$\frac{P \quad P \Rightarrow Q}{Q}$$

In words: If 1) P is known to be true, and 2) we assume that P ⇒ Q is true, then 3) Q must be true. Restricting for a moment to two-valued logic, we see from the truth-table for implication,

Implication

$p \Rightarrow q$

p \ q	0	1
0	1	1
1	0	1

→q

(13)

whenever P ⇒ Q and P are true then so is Q; by P true we consider only the second row, leaving Q true as the only possibility. In such an argument the assertions above the line are the premises, and the assertion below the line the conclusion. Notice the premises are assumed to be true, we are not considering all possible truth combinations as we did when proving tautologies.

On the other hand, underlying modus ponens is tautology 1, which expresses the same, but is valid for all truth-values. Therefore modus ponens is valid in fuzzy logic, if tautology 1 is valid in fuzzy logic.

Example 14 Four useful rules of inference

There are several useful rules of inference, which can be represented by tautological forms.

Four are presented here by examples from medicine.

(a) Modus ponens. Its tautological form is

[p ∧ (p ⇒ q)] ⇒ p.

Let p stand for 'altitude sickness,' and let p ⇒ q stand for 'altitude sickness causes a headache'. If it is known that John suffers from altitude sickness, p is true, and p ⇒ q is assumed to be true for the case of illustration, then the conclusion q is true, that is, John has a headache.

(b) Modus tollens. Its tautological form is

[¬q ∧ (p ⇒ q)] ⇒ ¬p.

Let p and q be as in (a). Thus, if John does not have a headache, then we may infer that John does not suffer from altitude sickness.

(c) Disjunctive syllogism. Its tautological form is

$[(p \vee q) \wedge \neg p] \Rightarrow q$.

Let p stand for 'altitude sickness' as previously, but let q stand for 'dehydration'. Thus, if it is known for a fact that John's headache is due to either altitude sickness or dehydration, and it is not altitude sickness, then we may infer that John suffers from dehydration.

(d) Hypothetical syllogism. Its tautological form is tautology 2:

$[(p \Rightarrow q) \wedge (q \Rightarrow r)] \Rightarrow (p \Rightarrow r)$.

Let p stand for 'high altitude and fast ascent', let q stand for 'altitude sickness', and let r stand for 'a headache'. Further assume that high altitude and fast ascent together cause altitude sickness, and in turn that altitude sickness causes a headache. Thus, we may infer that John will get a headache in high altitude if John ascends fast.

Provided the tautological forms are valid in fuzzy logic, the inference rules may be applied in fuzzy logic as well.

The inference mechanism in modus ponens can be generalized. The pattern is: given a relation R connecting logical variables p and q, we infer the possible values of q given a particular instance of p. Switching to vector-matrix representation, to emphasize the computer implementation, with p a (column) vector and R a two-dimensional truth-table, with the p-axis vertical, the inference is defined

$qt = pt \circ R$

The operation $\circ$ is an inner $\vee - \wedge$ product. The $\wedge$ operation is the same as in $p \wedge (p \Rightarrow q)$ and the $\vee$ operation along the columns yields what can possibly be implied about q, confer the rightmost implication in $[p \wedge (p \Rightarrow q)] \Rightarrow p$. Assuming p is true corresponds to setting

$$\mathbf{p} = \begin{pmatrix} 0 \\ 1 \end{pmatrix}.$$

But the scheme is more general, because we could also assume p is false, compose with R and study what can be inferred about q. Take for instance modus ponens, thus

$$\mathbf{R} = \begin{pmatrix} 1 & 1 \\ 0 & 1 \end{pmatrix}$$

which is the truth-table for $p \Rightarrow q$. Assigning p as above,

$$\mathbf{q}^t = \mathbf{p}^t \circ \mathbf{R} = \begin{pmatrix} 0 & 1 \end{pmatrix} \circ \begin{pmatrix} 1 & 1 \\ 0 & 1 \end{pmatrix} = \begin{pmatrix} 0 & 1 \end{pmatrix}$$

The outcome qt is a truth-vector pointing at q true as the only possible conclusion, as expected. Testing for instance with p = (1 0)t yields

$$\mathbf{q}^t = \mathbf{p}^t \circ \mathbf{R} = \begin{pmatrix} 1 & 0 \end{pmatrix} \circ \begin{pmatrix} 1 & 1 \\ 0 & 1 \end{pmatrix} = \begin{pmatrix} 1 & 1 \end{pmatrix}$$

Thus q could be anything, true or false, as expected. The inference could even proceed in the reverse direction, from q to p, but then we must compose from the right side of R to match the axes. Assume for instance q is true, or q = (1 0)t, then

$$\mathbf{p} = \mathbf{R} \circ \mathbf{q} = \begin{pmatrix} 1 & 1 \\ 0 & 1 \end{pmatrix} \circ \begin{pmatrix} 1 \\ 0 \end{pmatrix} = \begin{pmatrix} 1 \\ 0 \end{pmatrix}$$

To interpret: if q is false and p ⇒ q, then p is false (modus tollens).

The array based inference mechanism is even more general, because R can be any dimension n (n > 0 and integer). Given values of n − 1 variables, the possible outcomes of the remaining variable can be inferred by an n-dimensional inner product. Furthermore, given values of n − d variables (d integer and 0 < d < n), then the truth-array connecting the remaining d variables can be inferred.

Generalizing to three-valued logic p and q are vectors of three elements, R a 3-by-3 matrix, and the inner ∨ − ∧ product interpreted as the inner max-min product. Assuming p is true, corresponds to assigning

$$\mathbf{p} = \begin{pmatrix} 0 \\ 0.5 \\ 1 \end{pmatrix}$$

In modus ponens

$$\mathbf{R} = \begin{pmatrix} 1 & 1 & 1 \\ 0 & 1 & 1 \\ 0 & 0 & 1 \end{pmatrix}$$

which is the truth-table for sharp implication. With p as above,

$$\mathbf{q}^t = \mathbf{p}^t \circ \mathbf{R} = \begin{pmatrix} 0 & .5 & 1 \end{pmatrix} \circ \begin{pmatrix} 1 & 1 & 1 \\ 0 & 1 & 1 \\ 0 & 0 & 1 \end{pmatrix} = \begin{pmatrix} 0 & .5 & 1 \end{pmatrix}$$

To interpret, p true and p ⇒ q true, implies q true, as expected because tautology 1 is valid in three valued logic.

The inner product pt ◦R can be decomposed into three operations.

1. A cartesian product

$$\mathbf{R}^1 = \mathbf{p} \times \mathbf{1}^t = \begin{pmatrix} 0 \\ 0.5 \\ 1 \end{pmatrix} \times \begin{pmatrix} 1 & 1 & 1 \end{pmatrix} = \begin{pmatrix} 0 & 0 & 0 \\ 0.5 & 0.5 & 0.5 \\ 1 & 1 & 1 \end{pmatrix}$$

called the cylindrical extension. The result is a matrix whose columns are the vector p repeated as many times as necessary to fit the size of R.

2. An element-wise intersection

$$\mathbf{R}^2 = \mathbf{R}^1 \wedge \mathbf{R} = \begin{pmatrix} 0 & 0 & 0 \\ 0.5 & 0.5 & 0.5 \\ 1 & 1 & 1 \end{pmatrix} \wedge \begin{pmatrix} 1 & 1 & 1 \\ 0 & 1 & 1 \\ 0 & 0 & 1 \end{pmatrix} = \begin{pmatrix} 0 & 0 & 0 \\ 0 & 0.5 & 0.5 \\ 0 & 0 & 1 \end{pmatrix}$$

This operation does not have a name in the literature.

3. An ∨-operation along the columns of R2,

$$\mathbf{q}^t = \bigvee_p r^2_{pq} = \begin{pmatrix} 0 & 0.5 & 1 \end{pmatrix}$$

which is the projection of R on the q-axis.

In general, the max-min composition

$$\mathbf{b}^t = \mathbf{a}^t \circ \mathbf{R}$$

consists of three operations: 1) the cylindrical extension $\mathbf{R}^1 = \mathbf{a}^t \times 1_b$, 2) the element wise intersection $\mathbf{R}^2 = r^1_{ab} \wedge r_{ab}$, and 3) the projection onto the b-axis $\mathbf{b}^t = \bigvee_b r^2_{ab}$. This constitutes the compositional rule of inference, where b is inferred from a by means of R.

4 Fuzzy Rules

A fuzzy rule has the form

If x is A then y is B, in which A and B are fuzzy sets, defined on universes X and Y, respectively. This is an implication, where the antecedent is ’x is A’, and the consequent is ’y is B’. Examples of such rules in everyday conversation are

1. If it is dark, then drive slowly
2. If the tomato is red, then it is ripe
3. If it is early, then John can study
4. If the room is cold, then increase the heat
5. If the washing machine is half full, then wash shorter time

Other forms can be transcribed into the if-then form, for example ’when in Rome, do like the Romans’ becomes ’if in Rome, then do like the Romans’. Examples 4 and 5 could appear in a computer embedded in a heating unit or a washing machine.

To understand how a computer can infer a conclusion, take rule 3,

If it is early, then John can study

Assume that ’early’ is a fuzzy set defined on the universe

U = (4, 8, 12, 16, 20, 24) .

These are times t of the day, in steps of four hours, in 24 hour format to distinguish night from day numerically. Define ’early’ as a fuzzy set on U,

early = {(4, 0) , (8, 1) , (12, 0.9) , (16, 0.7) , (20, 0.5) , (24, 0.2)}

For simplicity define ’can study’ as a singleton fuzzy set μstudy = 1. If the hour is truly early, for instance 8 o’clock in the morning, then μearly(8) = 1, and thus John can study to the fullest degree, that is μstudy = 1. However, at 20 (8 pm) then μearly(20) = 0.5, and accordingly John can study to the degree 0.5. The degree of fulfillment of the antecedent (the if -side) weights the degree of fulfillment of the conclusion—a useful mechanism, that enables one rule to cover a range of hours. The procedure is: given a particular time instant t0, the resulting truth-value is computed as min (μearly(t0), μstudy).

4.1 Linguistic Variables

Whereas an algebraic variable takes numbers as values, a linguistic variable takes words or sentences as values (Zadeh in Zimmermann, 1993[16]). The name of such a linguistic variable is its label. The set of values that it can take is called its term set. Each value in the term set is a linguistic value or term defined over the universe. In summary: A linguistic variable takes a linguistic value, which is a fuzzy set defined on the universe.

Example 15 Term set Age

Let x be a linguistic variable labelled ’Age’. Its term set T could be defined as

T (age) = {young, very young, not very young, old, more or less old}

Each term is defined on the universe, for example the integers from 0 to 100 years.

A hedge is a word that acts on a term and modifies its meaning. For example, in the sentence ’very near zero’, the word ’very’ modifies the term ’near zero’. Examples of other hedges are ’a little’, ’more or less’, ’possibly’, and ’definitely’. In fuzzy reasoning a hedge operates on a membership function, and the result is a membership function.

Even though it is difficult precisely to say what effect the hedge 'very' has, it does have an intensifying effect. The hedge 'more or less' (or 'morl' for short) has the opposite effect.

Given a fuzzy term with the label A and membership function μA(x) defined on the universe X, the hedges 'very' and 'morl' are defined

$$very\ \mathcal{A} \equiv \{\langle x, \mu_{very\ \mathcal{A}}(x)\rangle \mid \mu_{very\ \mathcal{A}}(x) = \mu^2_{\mathcal{A}}(x),\ x \in \mathcal{X}\}$$

$$morl\ \mathcal{A} \equiv \left\{\langle x, \mu_{morl\ \mathcal{A}}(x)\rangle \mid \mu_{morl\ \mathcal{A}}(x) = \mu^{\frac{1}{2}}_{\mathcal{A}}(x),\ x \in \mathcal{X}\right\}$$

We have applied squaring and square root, but a whole family of hedges is generated by μ^k_A or $\mu^{1/k}_A$ (with integer k). For example

$$extremely\ \mathcal{A} \equiv \{\langle x, \mu_{very\ \mathcal{A}}(x)\rangle \mid \mu_{very\ \mathcal{A}}(x) = \mu^3_{\mathcal{A}}(x),\ x \in \mathcal{X}\}$$

$$slightly\ \mathcal{A} \equiv \left\{\langle x, \mu_{very\ \mathcal{A}}(x)\rangle \mid \mu_{very\ \mathcal{A}}(x) = \mu^{\frac{1}{3}}_{\mathcal{A}}(x),\ x \in \mathcal{X}\right\}$$

A derived hedge is for example somewhat A defined as morl A and not slightly A. For the special case where k = 2, the operation μ^2 is concentration and $\mu^{1/2}$ is dilation. With k = ∞ the hedge μ^k_A could be named exactly, because it would suppress all memberships lower than 1.

Example 16 Very on a discrete membership function

Assume a discrete universe U = h0, 20, 40, 60, 80i of ages. In Matlab we can assign

u = [0 20 40 60 80] and young = [1 .6 .1 0 0]

The discrete membership function for the set 'very young' is young.^2,

1 0.36 0.01 0 0

The notation '.^' is Matlab notation for the power operator. The set 'very very young' is, by repeated application, young.^4,

1 0.13 0 0 0

The derived sets inherit the universe of the primary set.

A primary term is a term that must be defined a priori, for example Young and Old in Fig. 4, whereas the sets Very young and Not very young are modified sets. The primary terms can be modified by negation ('not') or hedges ('very', 'more or less'), and the resulting sets can be connected using connectives ('and', 'or', 'implies', 'equals'). Long sentences can be built using this vocabulary, and the result is still a membership function.

4.2 Modus Ponens Inference

Modus ponens generalized to fuzzy logic is the core of fuzzy reasoning. Consider the argument

$$\frac{A' \qquad A \Rightarrow B}{B'} \tag{14}$$

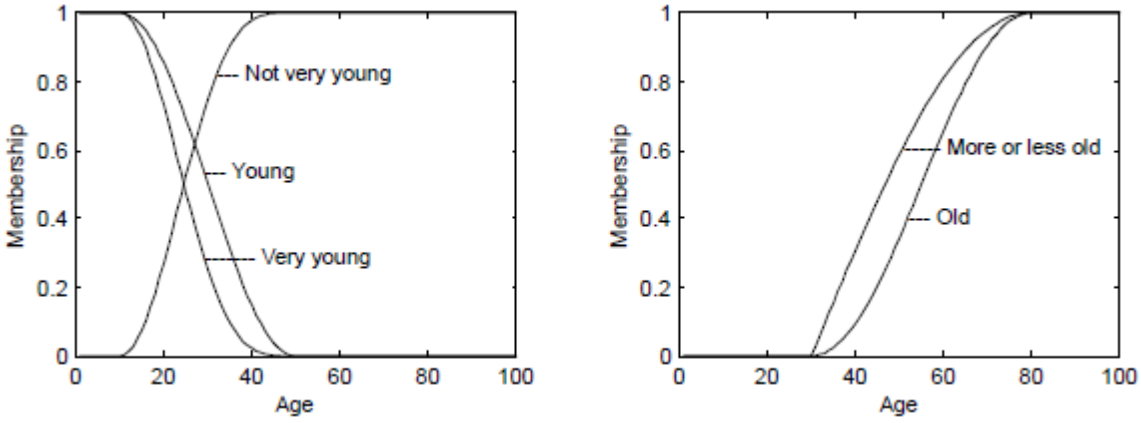


Figure 4. The membership functions for very young and not very young are derived from young (left), and the membership function for more or less old from old (right).

It is based on modus ponens, but the premise A0 is slightly different from A and thus the conclusion B0 is slightly different from B. For instance, given the rule 'if x is High, then y is Low'; if x in fact is 'very high', we would like to conclude that y is 'very low'.

Let A and A0 be fuzzy sets defined on X, and B a fuzzy set defined on Y. Then the fuzzy set B0, induced by 'x is A' from the fuzzy rule if x is A then y is B, is given by

$B0 = A0 \circ (A \Rightarrow B)$.

The operation $\circ$ is composition, the inner $\vee - \wedge$ product.

Example 17 Generalized modus ponens

Given the rule 'if altitude is High, then oxygen is Low'. Let the fuzzy set High be defined on a set of altitude ranges from 0 to 4000 metres (about 12 000 feet),

High = {(0, 0) , (1000, 0.25) , (2000, 0.5) , (3000, 0.75) , (4000, 1)} and Low be defined on a set of percentages of normal oxygen content,

Low = {(0, 1) , (25, 0.75) , (50, 0.5) , (75, 0.25) , (100, 1)}

As a shorthand notation we write the rule as a logical proposition High $\Rightarrow$ Low, where it is understood that the proposition concerns altitude and oxygen. We construct the relation R connecting High and Low using sharp implication, or $x \leq y$,

$\mathbf{R} =$

	1	.75	.5	.25	0
0	1	1	1	1	1
.25	1	1	1	1	0
.5	1	1	1	0	0
.75	1	1	0	0	0
1	1	0	0	0	0

The matrix is displayed with boxes and axis annotations to make the construction of the table clear: each element rxy is the evaluation of μHigh(x) ≤ μLow(y). The numbers on the vertical axis correspond to μHigh and the numbers on the horizontal axis correspond to μLow . Assuming altitude is High, we find by modus ponens

$$\begin{aligned}\mu^t &= \mu^t_{High} \circ \mathbf{R} \\ &= \begin{pmatrix} 0 & .25 & .5 & .75 & 1 \end{pmatrix} \circ \begin{pmatrix} 1 & 1 & 1 & 1 & 1 \\ 1 & 1 & 1 & 1 & 0 \\ 1 & 1 & 1 & 0 & 0 \\ 1 & 1 & 0 & 0 & 0 \\ 1 & 0 & 0 & 0 & 0 \end{pmatrix} \\ &= \begin{pmatrix} 1 & .75 & .5 & .25 & 0 \end{pmatrix}\end{aligned}$$

The result is identical to Low, thus modus ponens returns a result as expected in that case. Assume instead altitude is Very High,

$\mu^t_{VeryHigh} = \begin{pmatrix} 0 & .06 & .25 & .56 & 1 \end{pmatrix}$, the square of μ^t_{High}. Modus ponens yields

$$\begin{aligned}\mu^t &= \mu^t_{VeryHigh} \circ \mathbf{R} \\ &= \begin{pmatrix} 0 & .06 & .25 & .56 & 1 \end{pmatrix} \circ \begin{pmatrix} 1 & 1 & 1 & 1 & 1 \\ 1 & 1 & 1 & 1 & 0 \\ 1 & 1 & 1 & 0 & 0 \\ 1 & 1 & 0 & 0 & 0 \\ 1 & 0 & 0 & 0 & 0 \end{pmatrix} \\ &= \begin{pmatrix} 1 & .56 & .25 & .06 & 0 \end{pmatrix}\end{aligned}$$

The result is identical to the square of μ_{Low} . Written as an argument, we have

Very High
High ⇒ Low

Very Low

This is not always the case, though. With a slightly different definition of Low, say, Low = {(0, 1) , (25, 0.8) , (50, 0.5) , (75, 0.3) , (100, 0)} the resulting vector μt would be the same as previously, but only approximately equal to Very Low.

11.2 Fuzzy inferences

Fuzzy logic operators can be used as the basis for inference systems. Such fuzzy inference methods have been extensively studied by the expert systems community. Knowledge that can only be formulated in a fuzzy, imprecise manner can be captured in rules that can be processed by a computer.

11.2.1 Inferences from imprecise data

Fuzzy inference rules have the same structure as classical ones. The rules R1 and R2, for example, may have the form

R1:If(A ˜∧ B)then C.

R2:If(A ˜∨ B)then D.

The difference in conventional inference rules is the semantics of the fuzzy operators. In this section we identify the fuzzy operators ˜∧ and ˜∨ with the functions min and max respectively.

Let the truth values of A and B be 0.4 and 0.7 respectively. In this case

A ˜∧ B = min(0.4, 0.7) = 0.4

A ˜∨ B = max(0.4, 0.7) = 0.7

This is interpreted by the fuzzy inference mechanism as meaning that the rules R1 and R2 can only be partially applied that is rule R1 is applied to 40% and rule R2 to 70%. The result of the inference is a combination of the propositions C and D.

Let us consider another example. Assume that a temperature controller must regulate an electrical heater. We can formulate three rules for such a system:

R1:If (temperature = cold) then heat.

R2:If(temperature = normal)then maintain.

R3:If (temperature=warm) then reduce power.

Assume that a temperature of 12 degrees Celsius has a membership degree of 0.5 in relation to the set of cold temperatures and a membership degree of 0.3 in relation to the temperatures classified as normal. The temperature of 12 degrees is converted first of all into a fuzzy category which is the list of membership values of an element x of X in relation to previously selected fuzzy sets of the universal set X.

The fuzzy category T can be expressed using a similar notation as for fuzzy sets. In our example:

T = cold/0.5+ normal/0.3+ warm/0.0.

Note the difference in the notation for fuzzy sets. If a fuzzy category x is defined in relation to fuzzy sets A, B, and C, it is written as

$x = A/\mu A(x) + B/\mu B(x) + C/\mu C(x)$

and not as $x = \mu A(x)/A + \mu B(x)/B + \mu C(x)/C$.

Using T we can now evaluate the rules R1, R2, and R3 in parallel. The result is that each rule is valid to a certain extent. A fuzzy inference is the combination of the three possible consequences, weighted according to their validity. The result of a fuzzy inference is therefore a fuzzy category. In our example we deduce that

action = heat/0.5+ maintain/0.3+ reduce/0.0.

Fuzzy inference systems compute inferences of this type. Imprecise data, which is represented by fuzzy categories, leads to new fuzzy categories which represent the conclusion. In expert systems this kind of result can be processed further or it can be transformed into a crisp value. In the case of an electronic fuzzy controller this last step is always necessary.

The advantage of formulating fuzzy inference rules is their low granularity.

In many cases apparently complex control problems can be modeled using just a few rules. If we tried to express all actions as consequences of exact numerical rules, we would have to write more of them or make each one much more complex.

11.2.2 Fuzzy numbers and inverse operation

The example in the last section shows that a fuzzy controller operates, in general, in three steps: a) A measurement is transformed into a fuzzy category using the membership functions of all defined categories; b) All pertinent inference rules of the control system are evaluated and a fuzzy inference is produced; c) In the last step the result of the fuzzy inference is transformed into a crisp value.

There are several alternative ways to transform a measurement into fuzzy categories. A frequent approach is to use triangular or trapezium-shaped membership functions. Figure 11.10 shows how a measurement interval can be subdivided using triangular-shaped membership functions and Figure 11.11 shows the same kind of subdivision but with trapezium-shaped membership functions.

The transformation of a measurement x into a fuzzy category is given by the membership values $\alpha 1$, $\alpha 2$, $\alpha 3$ derived from the membership functions (as shown in Figure 11.10).

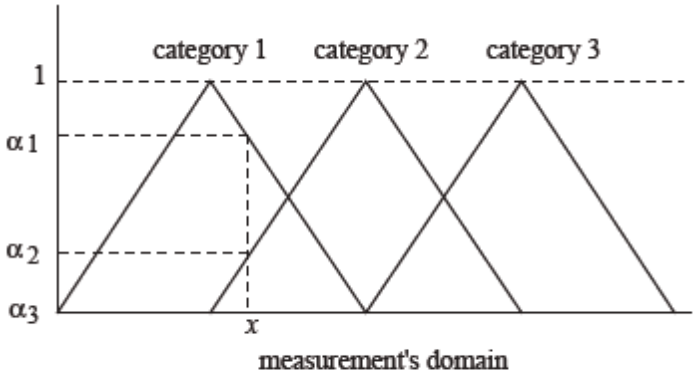


Fig. 11.10. Categories with triangular membership functions

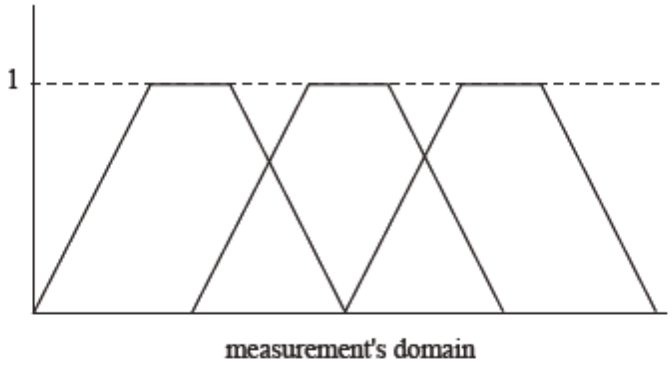


Fig. 11.11. Categories with trapezium-shaped membership functions

An important problem is how to transform the membership values α1, α2, α3 back into the measurement x, that is, how to implement the inverse operation to the fuzzifying of the crisp number. A popular approach is the centroid method. Figure 11.12 shows the value x and its transformation into α1, α2, α3. From these three values we can reconstruct the original x. To do this, the surfaces of the triangular regions limited by the heights α1, α2 and α3 are computed. The horizontal component of the centroid of the total surface is the approximation to x we are looking for (Figure 11.12).

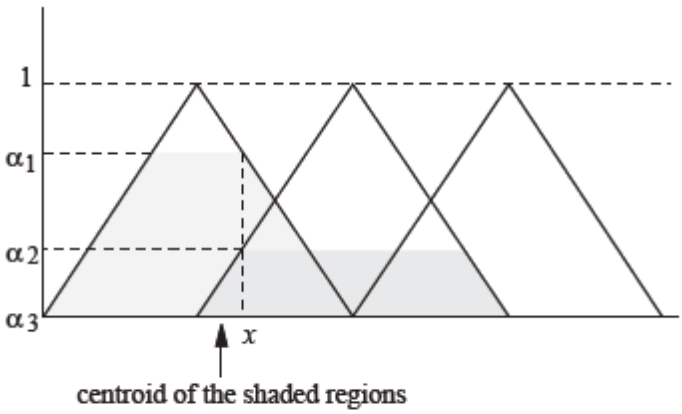


Fig. 11.12. The centroid method

For all x values for which at least two of the three numbers α1, α2, α3 are different from zero, we can compute a good approximation using the centroid method. Figure 11.13 shows the difference between x and its approximation when the basis of the triangular regions is of length 2, their height is 1 and the arrangement is the one shown in Figure 11.12. The value of x has been chosen in the interval [1, 2]. Figure 11.13 shows that the relative difference from the correct value of x is never greater than 10%.

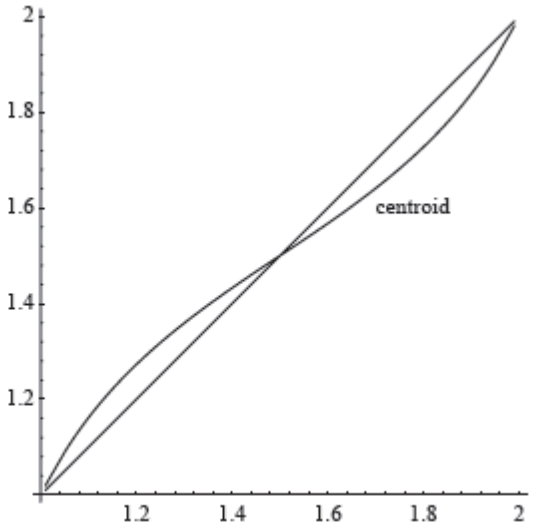


Fig. 11.13. Reconstruction error of the centroid method

The centroid method produces better or worse inverse transformations depending on the placement of the triangular categories. Weighting the surfaces of the triangular regions according to their position can also affect how good the inverse transformation is.

Summary

Fuzzy reasoning is based on fuzzy logic, which is again based on fuzzy set theory. The idea of a fuzzy set is basic and simple: an object is allowed to have a gradual membership of a set. This simple idea pervades all derived mathematical aspects of set theory. In fuzzy logic an assertion is allowed to be more or less true. A truth value in fuzzy logic is a real number in the interval [0, 1], rather than just the set of two truth values {0, 1} of classical logic. Classical logic can be fuzzified in many ways, but the central problem is to find a suitable definition for the connective 'implication'. Fuzzy reasoning is based on the modus ponens rule of inference, which again rests on the definition of 'implication'.

Not all laws in classical logic can be valid in fuzzy logic, therefore the design of a fuzzy system is a trade-off between mathematical rigour and engineering requirements.

A backward approach is recommended:

1. Start by deciding which laws are required to hold;
2. define 'and', 'or', 'not', 'implication', and 'equivalence';
3. check by means of their truth tables whether the laws in step 1 hold; and
4. if not, go to 2.

An assumption made in the tutorial is that the laws can be checked by just checking all combinations of three truth-values {0, 0.5, 1}. This is true for all expressions involving variables connected with $\wedge$, $\vee$, and $\neg$, but one must be cautious with expressions involving implication, as these generally cannot be reduced to those three connectives. Furthermore, it is assumed in the tutorial that truth-values are taken from a finite, discrete set of truth-values. This is not the case in general.

11.3 Control with fuzzy logic

A fuzzy controller is a regulating system whose modus operandi is specified with fuzzy rules. In general it uses a small set of rules. The measurements are processed in their fuzzified form, fuzzy inferences are computed, and the result is defuzzified, that is, it is transformed back into a specific number.

11.3.1 Fuzzy controllers

The example with the electrical heater will be completed in this section. We must determine the domain of definition of the variables used in the problem. Assume that the room temperature is a

number between 0 and 40 degrees Celsius. The controller can vary the electrical power consumed between 0 and 100 (in some suitable units), whereby 50 is the normal stand-by value. Figure 11.14 shows the membership functions for the temperature categories “cold”, “normal ”, and “warm” and the control categories “reduce”,“maintain”, and “heat”.

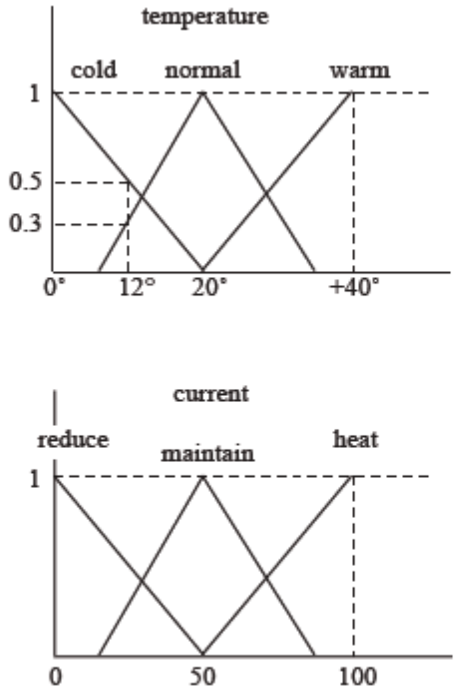


Fig. 11.14. Membership functions for temperature and electric current categories

The temperature of 12 degrees corresponds to the fuzzy number T = cold/0.5+normal/0.3+warm/0.0. These values lead to the previously computed inference action = heat/0.5+maintain/0.3+reduce/0.0.The controller must transform the result of this fuzzy inference into a definite value. The surfaces of the membership triangles below the inferred degree of membership are calculated. The light shaded surface in Figure 11.15 corresponds to the action “heat”, which is valid to 50%. The darker region corresponds to the action “maintain” that is valid to 30%. The centroid of the two shaded regions lies at about 70. This value for the power consumption is adjusted by the controller in order to heat the room.

It is of course possible to formulate more complex rules involving more than two variables. In all cases, though, we have to evaluate all rules simultaneously.

Fuzzy logic in control systems—case studies

Fuzzy logic in design methodology and for nonlinear control systems

Fuzzy logic is a paradigm for an alternative design methodology that can be applied in developing both linear and nonlinear systems for embedded control. Using fuzzy logic, designers can realize lower development costs, superior features, and better end-product performance. Furthermore, products can be brought to market faster and more cost effectively.

Simpler and faster design methodology

To appreciate why a fuzzy-based design methodology is very attractive in embedded control applications, let us examine a typical design flow. Figure 3 illustrates a sequence of design steps required to develop a controller using a conventional and a fuzzy approach. Using the conventional approach, the first step is to understand the physical system and its control requirements.

Based on this understanding, the second step is to develop a model that includes the plant, sensors, and actuators. The third step is to use linear-control theory in order to determine a simplified version of the controller, such as the parameters of a proportional-integral-derivative (PID) controller. The fourth step is to develop an algorithm for the simplified controller. The last step is to simulate the design including the effects of nonlinearity, noise, and parameter variations. If the performance is not satisfactory, we need to modify our system modeling, re-design the controller, rewrite the algorithm, and retry. Fuzzy-logic approach reduces the design process to three steps, starting with understanding and characterizing the system behavior through knowledge and experience. The second step is to directly design the control algorithm using fuzzy rules that describe the principles of the controller's regulation in terms of the relationship between its inputs and outputs. The last step is to simulate and debug the design. The fact that one only needs to modify some fuzzy rules and retry the process satisfies the performance requirements.

Though the two design methodologies are similar, fuzzy-based methodology substantially simplifies the design loop, resulting in significantly reduced development time, simpler design, and faster time to market. Fuzzy-logic design methodology simplifies the steps, especially during the debugging and tuning cycle, in which the system can be changed by simply modifying rules rather than redesigning the controller. The fuzzy rule-based feature focuses more on the application instead of programming, therefore substantially reducing the overall development cycle. Commercial applications in embedded control require a significant development effort, a majority of which is spent on the software portion of the project. Due to its simplicity, the

description of a fuzzy controller is not only is transportable across design teams, but also provides a superior medium to preserve, maintain, and upgrade intellectual property.

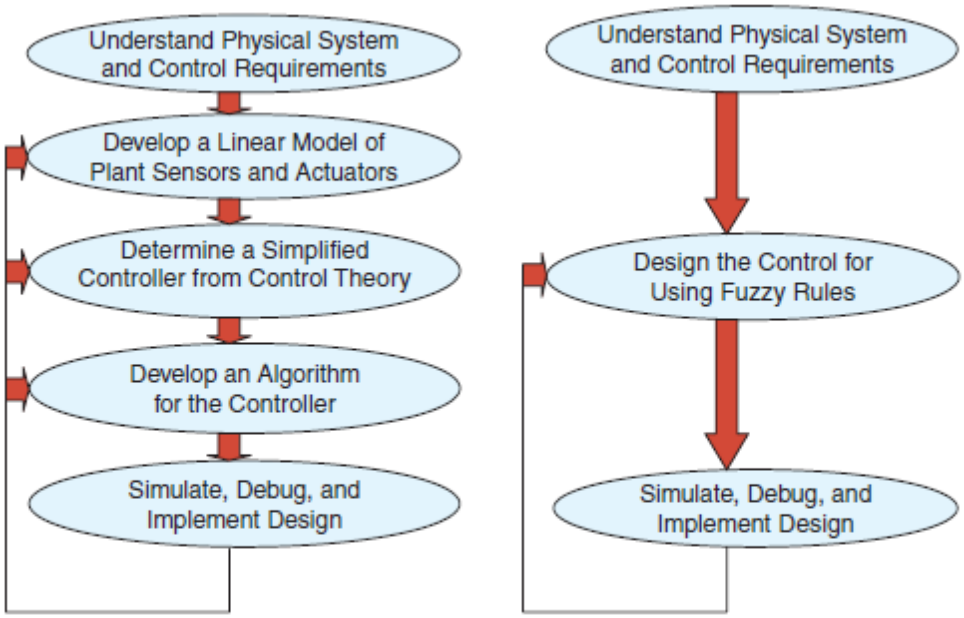


Fig. 3 Conventional and fuzzy design

A better alternative solution to nonlinear control

Most real-life physical systems are actually nonlinear systems. Conventional design approaches use different approximation methods to handle nonlinearity: linear, piecewise linear, and lookup table approximations to trade off factors of complexity, cost, and system performance. Fuzzy logic rules and membership functions approximate any continuous function to a degree of precision used as in Fig. 4, which shows an approximate desired control curve for temperature controller using four rules (or points). More rules can be added to increase the accuracy of the approximation, which yields improved control performance. Rules are much simpler to implement and much easier to debug and tune than piecewise linear or lookup table techniques.

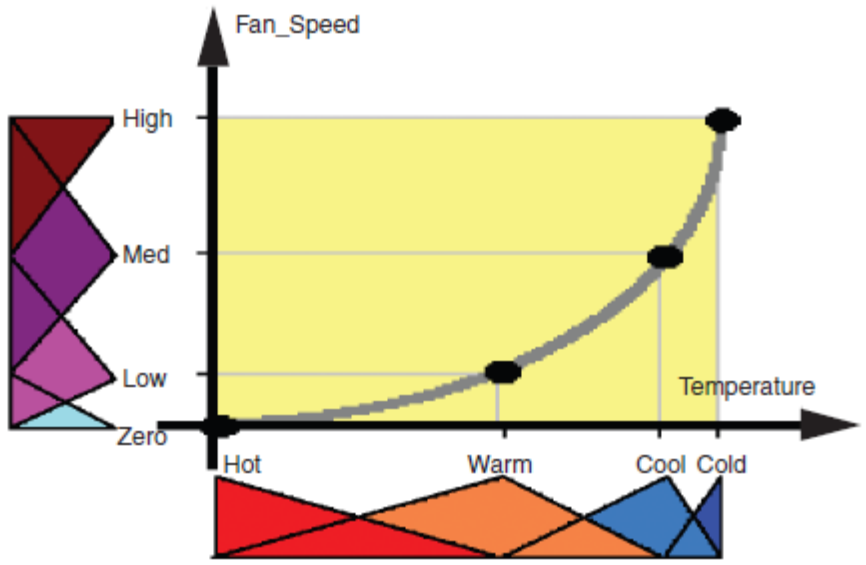


Fig. 4 Rules and membership function approximating a nonlinear function

The desired control curve for the temperature controller can be approximated using four points (or four rules) as in the following.

IF temperature IS cold **THEN** force IS high.

IF temperature IS cool **THEN** force IS medium.

IF temperature IS warm **THEN** force IS low.

IF temperature IS hot **THEN** force IS zero

The fuzzy arithmetic interpolates the shape of the nonlinear function. The combined memory required for the labels and fuzzy inference is substantially less than a lookup table, especially for multiple input systems. As a result, processing speed can be improved as well. Most control applications have multiple inputs and require modeling and tuning of a large number of parameters which makes implementation time consuming. Fuzzy rules can help simplify implementation by combining multiple inputs into single if-then statements while still handling nonlinearity as the following shows:

IF temperature IS cold AND humidity IS high THEN fan_spd IS high.

IF temperature IS cool AND humidity IS high THEN fan_spd IS medium.

IF temperature IS warm AND humidity IS high THEN fan_spd IS low.

IF temperature IS hot AND humidity IS high THEN fan_spd IS zero.

IF temperature IS cold AND humidity IS med THEN fan_spd IS medium.

IF temperature IS cool AND humidity IS med THEN fan_spd IS low.

IF temperature IS warm AND humidity IS med THEN fan_spd IS zero.

IF temperature IS hot AND humidity IS med THEN fan_spd IS zero.

IF temperature IS cold AND humidity IS low THEN fan_spd IS medium.

IF temperature IS cool AND humidity IS low THEN fan_spd IS low.

IF temperature IS warm AND humidity IS low THEN fan_spd IS zero.

IF temperature IS hot AND humidity IS low THEN fan_spd IS zero.

Fuzzy logic is used to describe the output as a function of two or more inputs linked with operators such as AND. It requires significantly less entries than a lookup table depending upon the number of labels for each input variable. Rules are much easier to develop and simpler to debug and tune than a lookup table, as in Fig. 5.

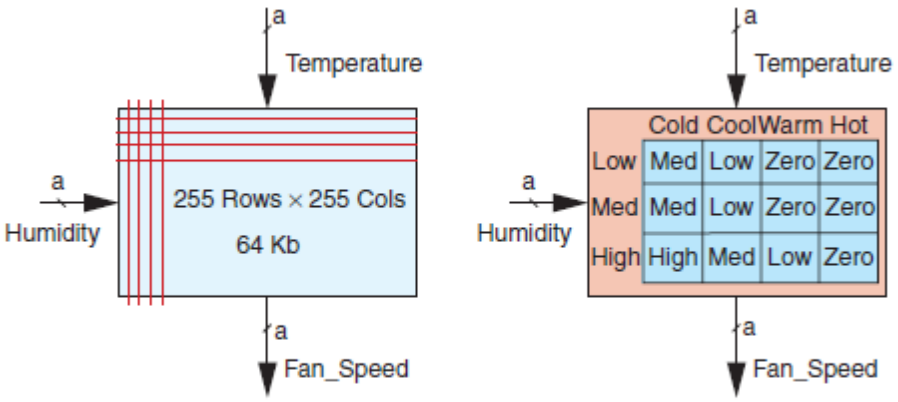


Fig. 5 Lookup table versus rules and membership functions

The lookup table for the two-input temperature controller requires 64 Kb of memory, while the fuzzy approach is accomplished with less than 0.5 Kb of memory for labels and the object code combined. This difference in memory savings implies a cheaper hardware implementation. Conventional techniques in most real life applications would require complex mathematical analysis and modeling, floating point algorithms, and complex branching. This typically yields a substantial size of object code, which requires a high end DSP chip to run. The fuzzy-logic approach uses a simple, rule-based approach that offers significant cost savings, both in memory and processor class.

11.3.2 Fuzzy networks

Fuzzy systems can be represented as networks. The computing units must implement fuzzy operators. Figure 11.16 shows a network with four hidden units. Each one of them receives the inputs x1, x2 and x3 which correspond to the fuzzy categorization of a specific number. The fuzzy operators are evaluated in parallel in the hidden layer of the network, which corresponds to the set of inference rules. The last unit in the network is the defuzzifier, which transforms the fuzzy inferences into a specific control variable. The importance of each fuzzy inference rule is weighted by the numbers α1, α2, and α3 as in a weighted centroid computation.

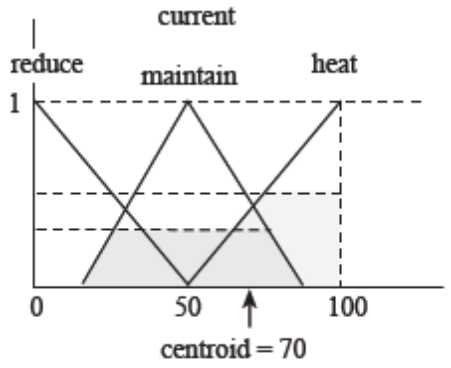


Fig. 11.15. Centroid computation

More complex rules can be implemented and this can lead to networks with several layers. However, fuzzy systems do not usually lead to very deep networks. Since at each fuzzy inference step the precision of the conclusion is reduced, it is not advisable to build too long an inference chain.

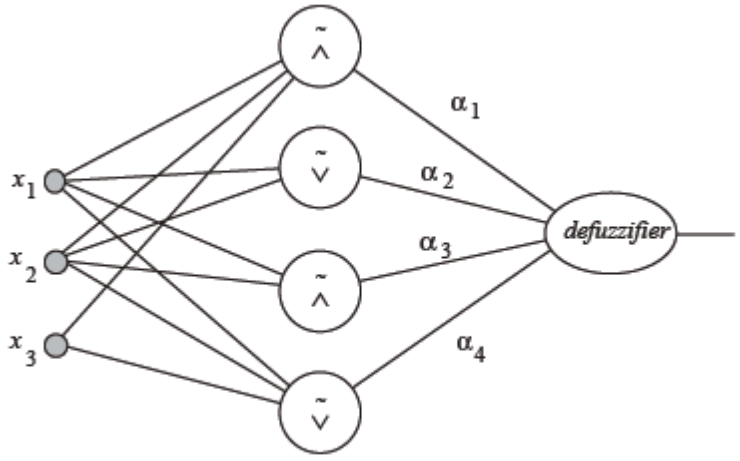


Fig. 11.16. Example of a fuzzy network

Fuzzy operators cannot be computed exactly by sigmoidal units, but for some of them a relatively good approximation is possible, for example, for bounded sum or bounded difference. A fuzzy inference chain using these operators can be approximated by a neural network.

The defuzzifier operator in the last layer can be approximated with standard units. If the membership functions are triangles, the surface of the triangles grows quadratically with the height. A quadratic function of this form can be approximated in the pertinent interval using sigmoids. The parameters of the approximation can be set with the help of a learning algorithm.

11.3.3 Function approximation with fuzzy methods

A fuzzy controller is just a system for the rapid computation of an approximation of a coarsely defined control surface, like the one shown in Figure 11.17.

The fuzzy controller computes a control variable according to the values of the variables x and y. Both variables are transformed into fuzzy categories.

Assume that each variable is transformed into a combination of three categories.

There are nine different combinations of the categories for x and y. For each of these nine combinations the value of the control variable is defined. This fixes nine points of the control surface.

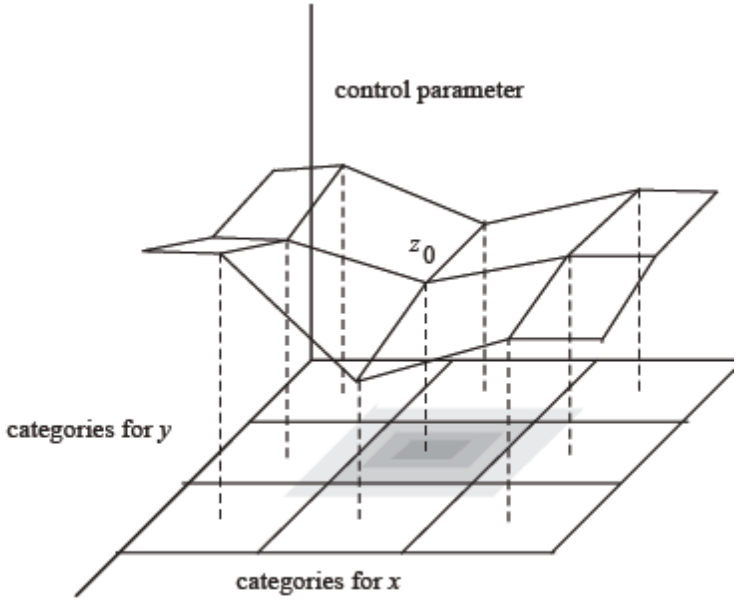


Fig. 11.17. Approximation of a control surface

Arbitrary values of x and y belong, to different degrees, to the nine combined categories. This means that for arbitrary combinations of x and y an interpolation of the known function values of

the control variable is needed. A fuzzy controller performs this computation according to the degree of membership of (x, y) in each combined category. In Figure 11.17 the different shadings of the quadratic regions in the xy plane represent the membership of the input in the category for which the control variable assumes the value z0. Other values, which correspond to the lighter shaded regions, receive a value for the control variable which is an interpolation of the neighboring z-values.

The control surface can be defined using a few points and, if the control function is smooth, a good approximation to other values is obtained with simple interpolation. The reduced number of given points corresponds to a reduced number of inference rules in the fuzzy controller. The advantage of such an approach lies in the economic use of rules. Inference rules can be defined in the fuzzy formalism in a straightforward manner. The interpolation mechanism is taken as given. This approach works of course only in the case where the control function has an adequate degree of smoothness.

Significant applications of fuzzy logic technology

Fuzzy logic has also seen its tremendous power used in business, finance, and management. FuzzyTECH for business is one of the new products with standard software already available for business, finance, and management in accurate forecasting and analysis. The models' applied fuzzy logic generates more accurate forecasting in the stock market. Pricing models for new products and fuzzy, zero-based budgeting are typical examples of fuzzy-logic technology applied for decision-making processes. One of the most important areas is the growing interest in fuzzy database models. The extensive and surging research interest in fuzzy relational database technology is now and will have an enormous impact on database technology. Fuzzy queries, a further approach from traditional database queries yet combined with the database technology, will provide very powerful new technology in database management systems. Fuzzy-logic defense applications include Aerospace-Missile A framework, which can be used as a guideline to design a Tagagi-Sugeno (T-S) fuzzy controller using a T-S fuzzy model. A T-S fuzzy controller can be designed using a T-S fuzzy model by using the antecedent part of the T-S fuzzy model as that of the T-S fuzzy controller. The synthesis of parallel-distributed compensation (PDC) type T-S fuzzy control systems can, in a systematic and analytic way, find proper feedback gains that simultaneously guarantee stability and system's performance. Therefore they have very broad use in fuzzy gain scheduling. Fuzzy logic and target maneuver detection of the

noise characteristics of a monopulse radar seeker are influenced by the motion of the target. When the relative rate of rotation between the radar and the target is low, the received noise is roughly Gaussian. If the target rotates rapidly, as in banking as a prelude to a turn, the noise statistics change. A set of fuzzy inference systems was developed that monitored the noise statistics and provided an indication of when the target maneuvered.

Hybrid modeling (HM) and land vehicle

HM is a methodology which fuses conventional mathematical techniques and intelligent techniques such as fuzzy logic, neural-networks, and genetic algorithms. HM can result in models without the cost of increased dimensionality when compared to models derived by applying strictly intelligent-based methods. In particular, HM is applied with a particular emphasis on the application of fuzzy logic in land vehicles. The force characteristics of electrically propelled land vehicles are fundamental in understanding and predicting the vehicles' handling and performance properties. The fuzzy logic methodology has been applied to predict the tyre forces by combining a well-balanced HM system representation that can provide a high-fidelity model.

Expert systems have been the most obvious recipients of the benefits of fuzzy logic, since their domain is often inherently fuzzy. Examples of expert systems with fuzzy logic central to their control are decision-support systems, financial planners, and diagnostic systems.

1. Introduction

The scope of this teaching package is to make a brief induction to Artificial Neural Networks (ANNs) for people who have no previous knowledge of them. We first make a brief introduction to models of networks, for then describing in general terms ANNs. As an application, we explain the backpropagation algorithm, since it is widely used and many other algorithms are derived from it.

The user should know algebra and the handling of functions and vectors. Differential calculus is recommendable, but not necessary. The contents of this package should be understood by people with high school education. It would be useful for people who are just curious about what are ANNs, or for people who want to become familiar with them, so when they study them more fully, they will already have clear notions of ANNs. Also, people who only want to apply the backpropagation algorithm without a detailed and formal explanation of it will find this material useful. This work should not be seen as "Nets for dummies", but of course it is not a treatise. Much of the formality is skipped for the sake of simplicity. Detailed explanations and demonstrations can be found in the referred readings. The included exercises complement the understanding of the theory. The on-line resources are highly recommended for extending this brief induction.

2. Networks

One efficient way of solving complex problems is following the lemma "divide and conquer". A complex system may be decomposed into simpler elements, in order to be able to understand it. Also simple elements may be gathered to produce a complex system (Bar Yam, 1997). Networks are one approach for achieving this. There are a large number of different types of networks, but they all are characterized by the following components: a set of nodes, and connections between nodes.

The nodes can be seen as computational units. They receive inputs, and process them to obtain an output. This processing might be very simple (such as summing the inputs), or quite complex (a node might contain another network...)

The connections determine the information flow between nodes. They can be unidirectional, when the information flows only in one sense, and bidirectional, when the information flows in either sense.

The interactions of nodes though the connections lead to a global behaviour of the network, which cannot be observed in the elements of the network. This global behaviour is said to be emergent. This means that the abilities of the network supercede the ones of its elements, making networks a very powerful tool.

Networks are used to model a wide range of phenomena in physics, computer science, biochemistry, ethology, mathematics, sociology, economics, telecommunications, and many other areas. This is because many systems can be seen as a network: proteins, computers, communities, etc. Which other systems could you see as a network? Why?

3. Artificial neural networks

One type of network sees the nodes as 'artificial neurons'. These are called artificial neural networks (ANNs). An artificial neuron is a computational model inspired in the natural neurons. Natural neurons receive signals through synapses located on the dendrites or membrane of the neuron. When the signals received are strong enough (surpass a certain threshold), the neuron is activated and emits a signal though the axon. This signal might be sent to another synapse, and might activate other neurons.

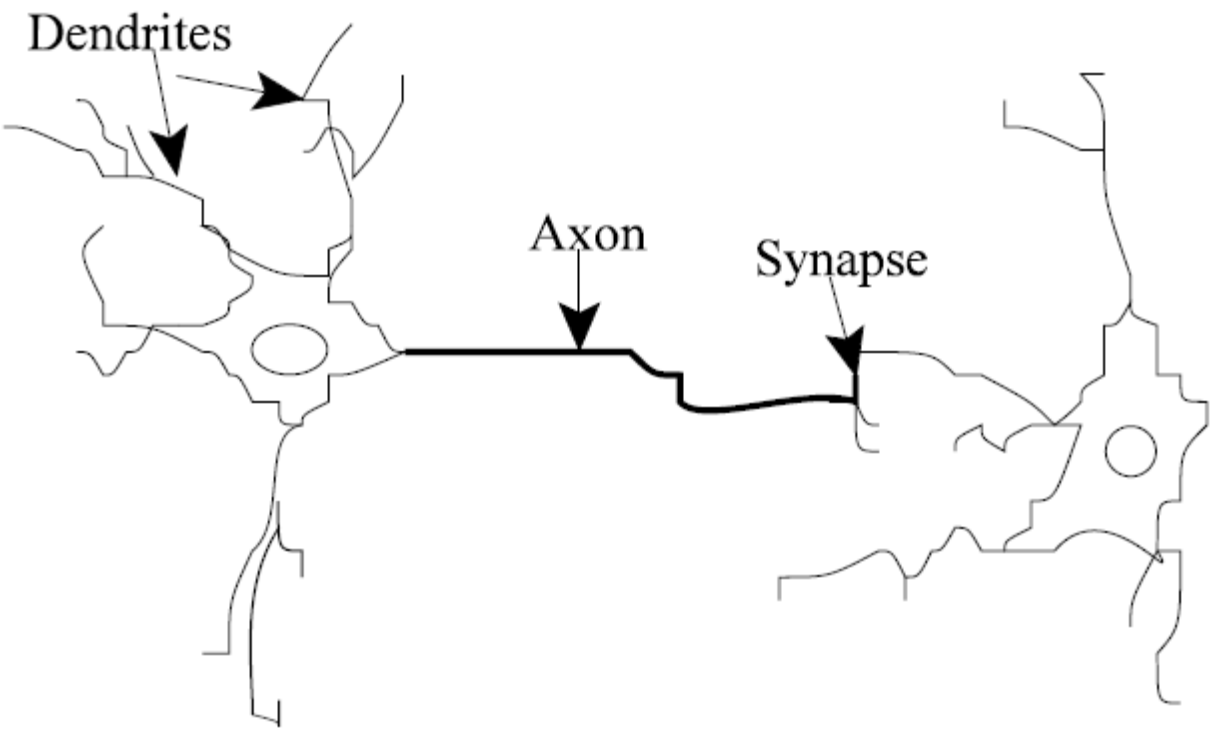


Figure 1. Natural neurons (artist's conception).

The complexity of real neurons is highly abstracted when modelling artificial neurons. These basically consist of inputs (like synapses), which are multiplied by weights (strength of the respective signals), and then computed by a mathematical function which determines the activation of the neuron. Another function (which may be the identity) computes the output of the artificial neuron (sometimes in dependance of a certain threshold). ANNs combine artificial neurons in order to process information.

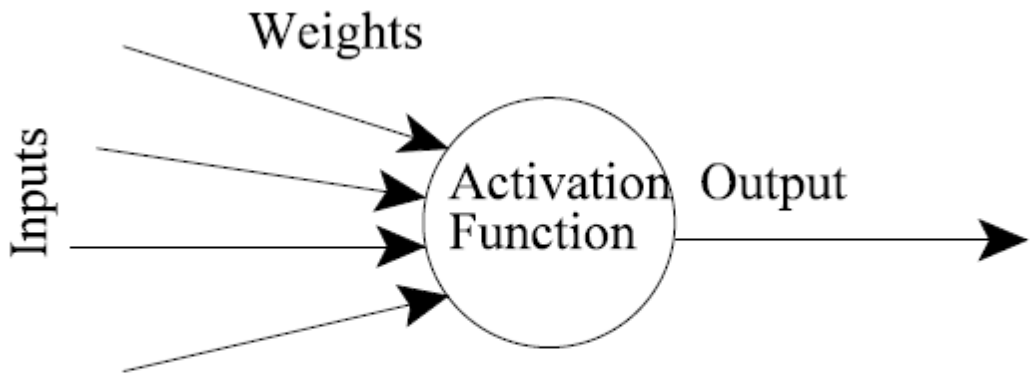


Figure 2. An artificial neuron

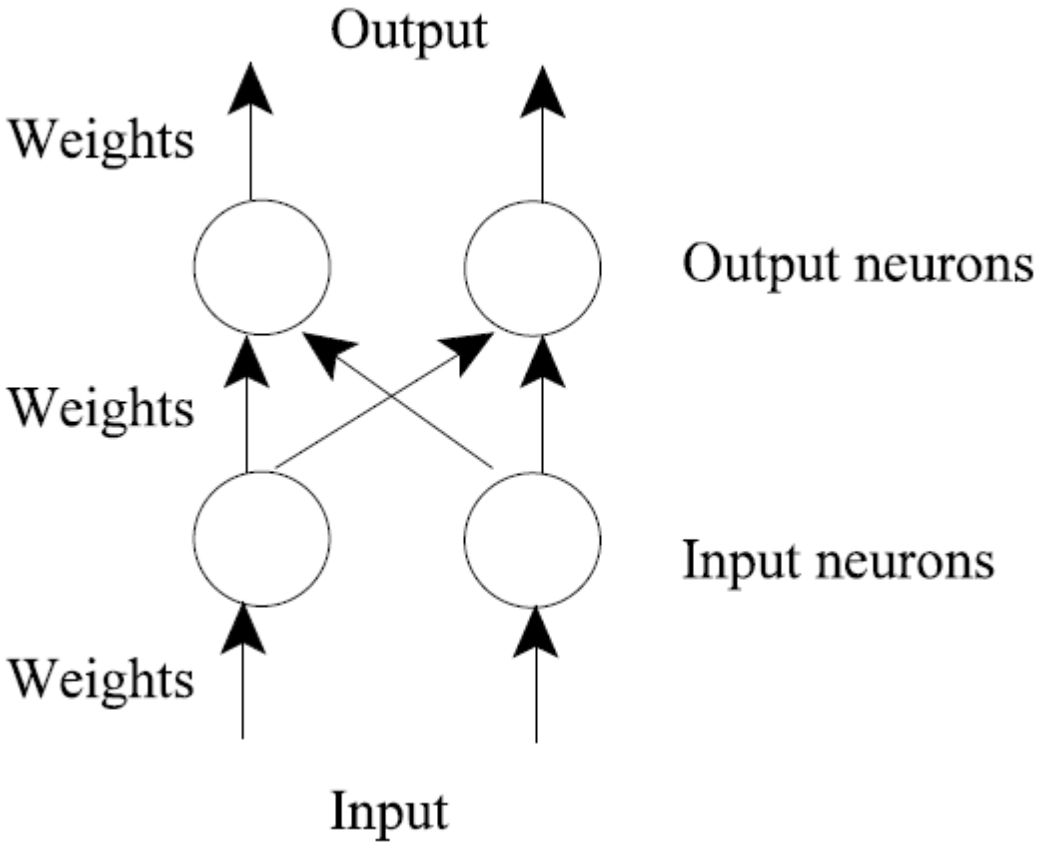
The higher a weight of an artificial neuron is, the stronger the input which is multiplied by it will be. Weights can also be negative, so we can say that the signal is inhibited by the negative weight. Depending on the weights, the computation of the neuron will be different. By adjusting the weights of an artificial neuron we can obtain the output we want for specific inputs. But when we have an ANN of hundreds or thousands of neurons, it would be quite complicated to find by hand all the necessary weights. But we can find algorithms which can adjust the weights of the ANN in order to obtain the desired output from the network. This process of adjusting the weights is called learning or training.

The number of types of ANNs and their uses is very high. Since the first neural model by McCulloch and Pitts (1943) there have been developed hundreds of different models considered as ANNs. The differences in them might be the functions, the accepted values, the topology, the learning algorithms, etc. Also there are many hybrid models where each neuron has more properties than the ones we are reviewing here. Because of matters of space, we will present only an ANN which learns using the backpropagation algorithm (Rumelhart and McClelland, 1986) for learning the appropriate weights, since it is one of the most common models used in ANNs, and many others are based on it.

Since the function of ANNs is to process information, they are used mainly in fields related with it. There are a wide variety of ANNs that are used to model real neural networks, and study behaviour and control in animals and machines, but also there are ANNs which are used for engineering purposes, such as pattern recognition, forecasting, and data compression.

3.1. Exercise

This exercise is to become familiar with artificial neural network concepts. Build a network consisting of four artificial neurons. Two neurons receive inputs to the network, and the other two give outputs from the network.



There are weights assigned with each arrow, which represent information flow.

These weights are multiplied by the values which go through each arrow, to give more or less strength to the signal which they transmit. The neurons of this network just sum their inputs. Since the input neurons have only one input, their output will be the input they received multiplied by a weight. What happens if this weight is negative? What happens if this weight is zero?

The neurons on the output layer receive the outputs of both input neurons, multiplied by their respective weights, and sum them. They give an output which is multiplied by another weight.

Now, set all the weights to be equal to one. This means that the information will flow unaffected. Compute the outputs of the network for the following inputs: (1,1), (1,0), (0,1),
(0,0), (-1,1), (-1,-1).

Good. Now, choose weights among 0.5, 0, and -0.5, and set them randomly along the network. Compute the outputs for the same inputs as above. Change some weights and see how the behaviour of the networks changes. Which weights are more critical (if you change those weights, the outputs will change more dramatically)?

Now, suppose we want a network like the one we are working with, such that the outputs should be the inputs in inverse order (e.g. (0.3,0.7)->(0.7,0.3)).

That was an easy one! Another easy network would be one where the outputs should be the double of the inputs.

Now, let's set thresholds to the neurons. This is, if the previous output of the neuron (weighted sum of the inputs) is greater than the threshold of the neuron, the output of the neuron will be one, and zero otherwise. Set thresholds to a couple of the already developed networks, and see how this affects their behaviour.

Now, suppose we have a network which will receive for inputs only zeroes and/or ones. Adjust the weights and thresholds of the neurons so that the output of the first output neuron will be the conjunction (AND) of the network inputs (one when both inputs are one, zero otherwise), and the output of the second output neuron will be the disjunction

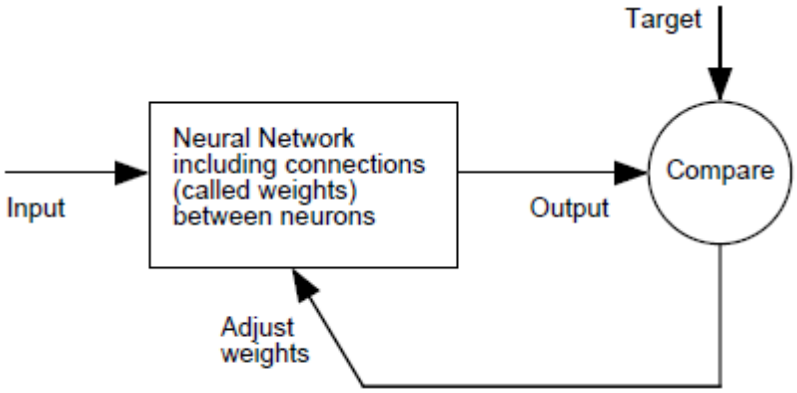
(OR) of the network inputs (zero in both inputs are zeroes, one otherwise). You can see that there is more than one network which will give the requested result.

Now, perhaps it is not so complicated to adjust the weights of such a small network, but also the capabilities of this are quite limited. If we need a network of hundreds of neurons, how would you adjust the weights to obtain the desired output? There are methods for finding them, and now we will expose the most common one.

Neural Networks

Neural networks are composed of simple elements operating in parallel. These elements are inspired by biological nervous systems. As in nature, the network function is determined largely by the connections between elements. We can train a neural network to perform a particular function by adjusting the values of the connections (weights) between elements.

Commonly neural networks are adjusted, or trained, so that a particular input leads to a specific target output. Such a situation is shown below. There, the network is adjusted, based on a comparison of the output and the target, until the network output matches the target. Typically many such input/target pairs are used, in this supervised learning, to train a network.



Batch training of a network proceeds by making weight and bias changes based on an entire set (batch) of input vectors. Incremental training changes the weights and biases of a network as needed after presentation of each individual input vector. Incremental training is sometimes referred to as “on line” or “adaptive” training.

Neural networks have been trained to perform complex functions in various fields of application including pattern recognition, identification, classification, speech, vision and control systems. A list of applications is given in Chapter 1.

Today neural networks can be trained to solve problems that are difficult for conventional computers or human beings. Throughout the toolbox emphasis is placed on neural network paradigms that build up to or are themselves used in engineering, financial and other practical applications.

The supervised training methods are commonly used, but other networks can be obtained from unsupervised training techniques or from direct design methods. Unsupervised networks can be used, for instance, to identify groups of data. Certain kinds of linear networks and Hopfield networks are designed directly. In summary, there are a variety of kinds of design and learning techniques that enrich the choices that a user can make.

The field of neural networks has a history of some five decades but has found solid application only in the past fifteen years, and the field is still developing rapidly. Thus, it is distinctly different from the fields of control systems or optimization where the terminology, basic mathematics, and design procedures have been firmly established and applied for many years. We do not view the Neural Network Toolbox as simply a summary of established procedures that are known to work well. Rather, we hope that it will be a useful tool for industry, education and research, a tool that will help users find what works and what doesn’t, and a tool that will

help develop and extend the field of neural networks. Because the field and the material are so new, this toolbox will explain the procedures, tell how to apply them, and illustrate their successes and failures with examples. We believe that an understanding of the paradigms and their application is essential to the satisfactory and successful use of this toolbox, and that without such understanding user complaints and inquiries would bury us. So please be patient if we include a lot of explanatory material. We hope that such material will be helpful to you.

Computational models of neurons

McCulloch and Pitts4 proposed a binary threshold unit as a computational model for an artificial neuron (see Figure 2).

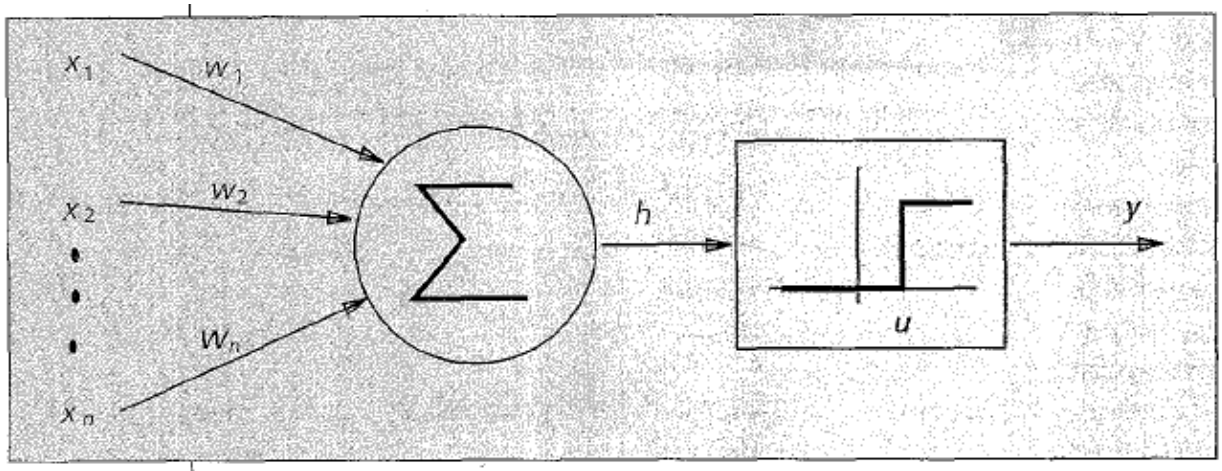


Figure 2. McCulloch-Pitts model of a neuron.

This mathematical neuron computes a weighted sum of its n input signals,x,, j = 1,2, . . . , n, and generates an output of 1 if this sum is above a certain threshold U. Otherwise, an output of 0 results. Mathematically,

$$y = \theta\left(\sum_{j=1}^{n} w_j x_j - u\right),$$

where **O(**) is a unit step function at 0, and w, is the synapse weight associated with the jth input. For simplicity of notation, we often consider the threshold U as another weight wo = - U attached to the neuron with a constant input **x,**

= 1. Positive weights correspond to excitatory synapses, while negative weights model inhibitory ones. McCulloch and Pitts proved that, in principle, suitably chosen weights let a synchronous

arrangement of such neurons perform universal computations. There is a crude analogy here to a biological neuron: wires and interconnections model axons and dendrites, connection weights represent synapses, and the threshold function approximates the activity in a soma. The McCulloch and Pitts model, however, contains a number of simplifylng assumptions that do not reflect the true behavior of biological neurons.

The McCulloch-Pitts neuron has been generalized in many ways. An obvious one is to use activation functions other than the threshold function, such as piecewise linear, sigmoid, or Gaussian, as shown in Figure 3. The sigmoid function i s by far the most frequently used in A"s. It is a strictly increasing function that exhibits smoothness and has the desired asymptotic properties.

The standard sigmoid function is the logistic function, defined by

$$g(x) = 1/(1 + \exp\{-\beta x\}),$$

where β is the slope parameter.

Network architectures

A"s can be viewed as weighted directed graphs in which artificial neurons are nodes and directed edges (with weights) are connections between neuron outputs and neuron inputs.

Based on the connection pattern (architecture), A"s can be grouped into two categories (see Figure 4) :

* feed-forward networks, in which graphs have no loops, and
* recurrent (or feedback) networks, in which loops occur because of feedback connections.

In the most common family of feed-forward networks, called multilayer perceptron, neurons are organized into layers that have unidirectional connections between them.

Figure 4 also shows typical networks for each category.

Different connectivities yield different network behaviors. Generally speaking, feed-forward networks are static, that is, they produce only one set of output values rather than a sequence of values from a given input. Feed-forward networks are memory-less in the sense that their response to an input is independent of the previous network state.

Recurrent, or feedback, networks, on the other hand, are dynamic systems. When a new input pattern is presented, the neuron outputs are computed. Because of the feedback paths, the inputs to each neuron are then modified, which leads the network to enter a new state. Different

network architectures require appropriate learning algorithms. The next section provides an overview of learning processes.

Learning

The ability to learn is a fundamental trait of intelligence. Although aprecise definition of learning is difficult to formulate, a learning process in the ANN context can be viewed as the problem of updating network architecture and connection weights so that a network can efficiently perform a specific task. The network usually must learn the connection weights from available training patterns. Performance is improved over time by iteratively updating the weights in the network. ANNs' ability to automatically learnfrom examples makes them attractive and exciting. Instead of following a set of rules specified by human experts, ANNs appear to learn underlying rules (like input-output relationships) from the given collection of representative examples. This is one of the major advantages of neural networks over traditional expert systems. To understand or design a learning process, you must first have a model of the environment in which a neural network operates, that is, you must know what information is available to the network. We refer to this model as a learning paradigm. Second, you must understand how network weights are updated, that is, which learning rules govern the updating process. A learning algorithm refers to a procedure in which learning rules are used for adjusting the weights.

There are three main learning paradigms: supervised, unsupervised, and hybrid. In supervised learning, or learning with a “teacher,” the network is provided with a correct answer (output) for every input pattern. Weights are determined to allow the network to produce answers as close as possible to the known correct answers.

Reinforcement learning is a variant of supervised learning in which the network is provided with only a critique on the correctness of network outputs, not the correct answers themselves. In contrast, unsupervised learning, or learning without a teacher, does not require a correct answer associated with each input pattern in the training data set. It explores the underlying structure in the data, or correlations between patterns in the data, and organizes patterns into categories from these correlations.

Hybrid learning combines supervised and unsupervised learning. Part of the weights is usually determined through supervised learning, while the others are obtained through unsupervised learning.

Learning theory must address three fundamental and practical issues associated with learning from samples: capacity, sample complexity, and computational complexity. Capacity concerns how many patterns can be stored, and what functions and decision boundaries a network can form.

Sample complexity determines the number of training patterns needed to train the network to guarantee a valid generalization. Too few patterns may cause "over-fitting" (wherein the network performs well on the training data set, but poorly on independent test patterns drawn from the same distribution as the training patterns, as in Figure A3). Computational complexity refers to the time required for a learning algorithm to estimate a solution from training patterns. Many existing learning algorithms have high computational complexity. Designing efficient algorithms for neural network learning is a very active research topic.

There are four basic types of learning rules: error correction, Boltzmann, Hebbian, and competitive learning.

Error-correction rules**.** In the supervised learning paradigm, the network is given a desired output for each input pattern. During the learning process, the actual output y generated by the network may not equal the desired output d. The basic principle of error-correction learning rules is to use the error signal (d -y) to modify the connection weights to gradually reduce this error. The perceptron learning rule is based on this error-correction principle. A perceptron consists of a single neuron with adjustable weights, **w,,**j = 1,2, . . . ,n , and threshold **U,**a s shown in Figure 2. Given an input vector **x= (xl,**x, , . . . , **xJt,** the net input to the neuron is

$$v = \sum_{j=1}^{n} w_j x_j - u$$

The outputy of the perceptron is + 1 ifv > 0, and 0 otherwise. In a two-class classification problem, the perceptron assigns an input pattern to one class ify = 1, and to the other class ify=O. The linear equation

$$\sum_{j=1}^{n} w_j x_j - u = 0$$

defines the decision boundary (a hyper plane in the n-dimensional input space) that halves the space. Rosenblatt5 developed a learning procedure to determine the weights and threshold in a perceptron, given a set of training patterns (see the "Perceptron learning algorithm" sidebar).

Note that learning occurs only when the perceptron makes an error. Rosenblatt proved that when training patterns are drawn from two linearly separable classes, the perceptron learning procedure converges after a finite number of iterations. This is the perceptron convergence theorem. In practice, you do not know whether the patterns are linearly separable. Many variations of this learning algorithm have been proposed in the literature.2 Other activation functions that lead to different learning characteristics can also be used. However, a single-layer perceptron can only separate linearly separable patterns as long **as** a monotonic activation function is used. The back-propagation learning algorithm (see the "Back-propagation algorithm sidebar") is also based on the error-correction principle.

Boltzmann machines are symmetric recurrent networks consisting of binary units (+ 1 for "on" and -1 for "off"). By symmetric, we mean that the weight on the connection from unit i to unit j is equal to the weight on the connection from unit j to unit i (wy = wJ. A subset of the neurons, called visible, interact with the environment; the rest, called hidden, do not. Each neuron is a stochastic unit that generates an output (or state) according to the Boltzmann distribution of statistical mechanics. Boltzmann machines operate in two modes: clamped, in which visible neurons are clamped onto specific states determined by the environment; and free-running, in which both visible and hidden neurons are allowed to operate freely.

Boltzmann learning is a stochastic learning rule derived from information-theoretic and thermodynamic principles. LOThe objective of Boltzmann learning is to adjust the connection weights so that the states of visible units satisfy a particular desired probability distribution. According to the Boltzmann learning rule, the change in the connection weight **wg** is given by

$$\Delta w_{ij} = \eta(\overline{\rho}_{ij} - \rho_{ij}),$$

where q is the learning rate, and **p,** and py are the correlations between the states of units z and J when the network operates in the clamped mode and free-running mode, respectively. The values of pli and p, are usually estimated from Monte Carlo experiments, which are extremely slow. Boltzmann learning can be viewed as a special case of error-correction learning in which error IS measured not as the direct difference between desired and actual outputs, but as the difference between the correlations among the outputs of two neurons under clamped and free running operating conditions.

HEBBIAN RULE. The oldest learning rule is Hebb's postulate of 1earning.Hebb based it on the following observation from neurobiological experiments: If neurons on both sides of a synapse are activated synchronously and repeatedly, the synapse's strength is selectively increased. Mathematically, the Hebbian rule can be described as

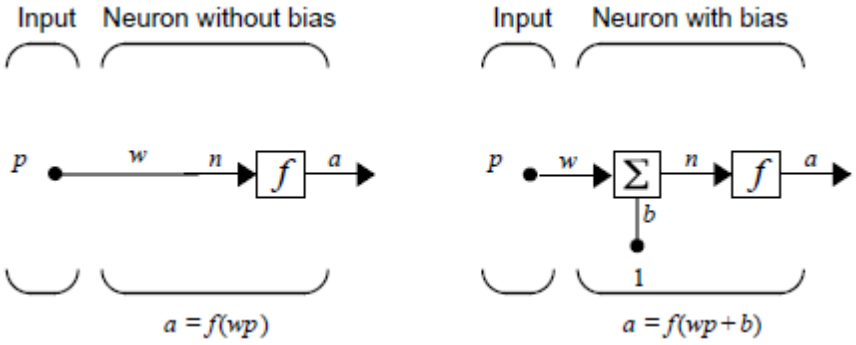
$$w_{ij}(t+1) = w_{ij}(t) + \eta\, y_j(t)\, x_i(t),$$

where x, and y, are the output values of neurons i and J , respectively, which are connected by the synapse w_{ij} and η is the learning rate. Note that x, is the input to the synapse. An important property of this rule is that learning is done locally, that is, the change in synapse weight depends only on the activities of the two neurons connected by it. This significantly simplifies the complexity of the learning circuit in a VLSI implementation. A single neuron trained using the Hebbian rule exhibits orientation selectivity. Figure 5 demonstrates this property. The points depicted are drawn from a two-dimensional Gaussian distribution and used for training a neuron. The weight vector of the neuron is initialized tow, as shown in the figure. As the learning proceeds, the weight vector moves progressively closer to the direction w of maximal variance in the data. In fact, **w** is the eigenvector of the covariance matrix of the data corresponding to the largest eigenvalue. -

Neuron Model

Simple Neuron

A neuron with a single scalar input and no bias appears on the left below.



The scalar input p is transmitted through a connection that multiplies its strength by the scalar weight w, to form the product wp, again a scalar. Here the weighted input wp is the only argument of the transfer function f, which produces the scalar output a. The neuron on the right has a scalar bias, b. You may view the bias as simply being added to the product wp as shown by

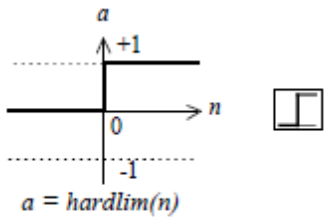
the summing junction or as shifting the function f to the left by an amount b. The bias is much like a weight, except that it has a constant input of 1. The transfer function net input n, again a scalar, is the sum of the weighted input wp and the bias b. This sum is the argument of the transfer function f.

Here f is a transfer function, typically a step function or a sigmoid function, which takes the argument n and produces the output a. Examples of various transfer functions are given in the next section. Note that w and b are both adjustable scalar parameters of the neuron. The central idea of neural networks is that such parameters can be adjusted so that the network exhibits some desired or interesting behavior. Thus, we can train the network to do a particular job by adjusting the weight or bias parameters, or perhaps the network itself will adjust these parameters to achieve some desired end.

All of the neurons in this toolbox have provision for a bias, and a bias is used in many of our examples and will be assumed in most of this toolbox. However, you may omit a bias in a neuron if you want. As previously noted, the bias b is an adjustable (scalar) parameter of the neuron. It is not an input. However, the constant 1 that drives the bias is an input and must be treated as such when considering the linear dependence of input vectors in Chapter 4, "Linear Filters."

Transfer Functions

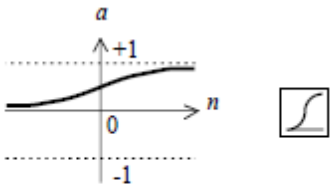
Many transfer functions are included in this toolbox. A complete list of them can be found in "Transfer Function Graphs" in Chapter 14. Three of the most commonly used functions are shown below.



Hard-Limit Transfer Function

The hard-limit transfer function shown above limits the output of the neuron to either 0, if the net input argument n is less than 0; or 1, if n is greater than or equal to 0. We use this function "Perceptrons" to create neurons that make classification decisions. Neurons of this type are used as linear approximators in "Linear Filters".

The sigmoid transfer function shown below takes the input, which may have any value between plus and minus infinity, and squashes the output into the range 0 to 1.



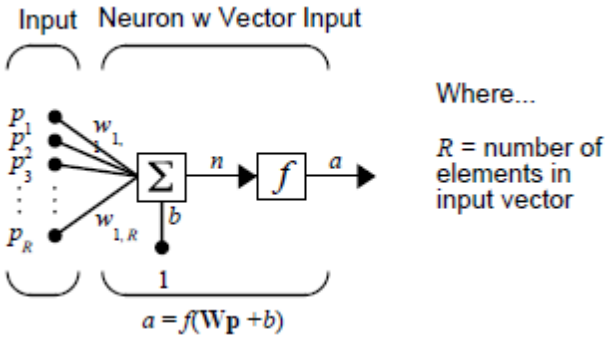
a = logsig(n)

Log-Sigmoid Transfer Function

This transfer function is commonly used in backpropagation networks, in part because it is differentiable. The symbol in the square to the right of each transfer function graph shown above represents the associated transfer function. These icons will replace the general f in the boxes of network diagrams to show the particular transfer function being used.

Neuron with Vector Input

A neuron with a single R-element input vector is shown below. Here the individual element inputs are multiplied by weights and the weighted values are fed to the summing junction. Their sum is simply **Wp**, the dot product of the (single row) matrix **W** and the vector **p**.



Where...

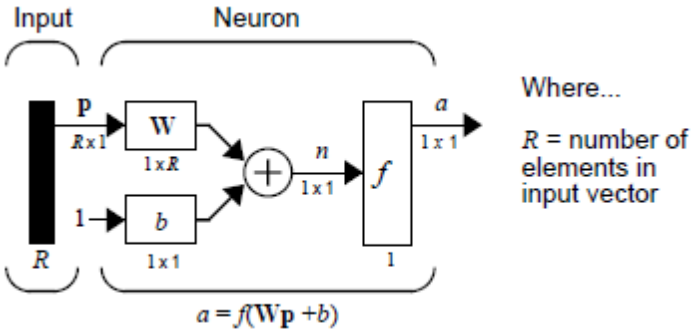
R = number of elements in input vector

The neuron has a bias b, which is summed with the weighted inputs to form the net input n. This sum, n, is the argument of the transfer function f.

$$n = w_{1,1}p_1 + w_{1,2}p_2 + \ldots + w_{1,R}p_R + b$$

The figure of a single neuron shown above contains a lot of detail. When we consider networks with many neurons and perhaps layers of many neurons, there is so much detail that the main thoughts tend to be lost. Thus, the authors have devised an abbreviated notation for an individual neuron. This notation, which will be used later in circuits of multiple neurons, is illustrated in the diagram shown below. Here the input vector **p** is represented by the solid dark vertical bar at the

left. The dimensions of **p** are shown below the symbol **p** in the figure as Rx1. (Note that we will use a capital letter, such as R in the previous sentence, when referring to the size of a vector.) Thus, **p** is a vector of R input elements. These inputs post multiply the single row, R column matrix **W**. As before, a constant 1 enters the neuron as an input and is multiplied by a scalar bias b. The net input to the transfer function f is n, the sum of the bias b and the product **Wp**. This sum is passed to the transfer function f to get the neuron's output a, which in this case is a scalar. Note that if we had more than one neuron, the network output would be a vector.



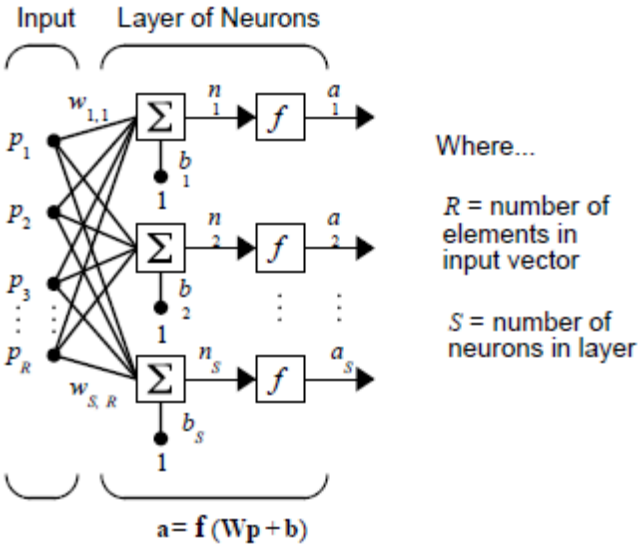
A layer of a network is defined in the figure shown above. A layer includes the combination of the weights, the multiplication and summing operation (here realized as a vector product **Wp**), the bias b, and the transfer function f. The array of inputs, vector **p**, is not included in or called a layer. Each time this abbreviated network notation is used, the size of the matrices will be shown just below their matrix variable names.

Network Architectures

Two or more of the neurons shown earlier can be combined in a layer, and a particular network could contain one or more such layers. First consider a single layer of neurons.

A Layer of Neurons

A one-layer network with R input elements and S neurons follows.

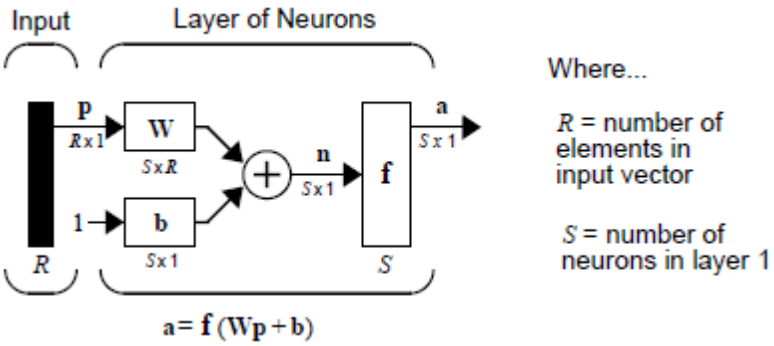


In this network, each element of the input vector **p** is connected to each neuron input through the weight matrix **W**. The ith neuron has a summer that gathers its weighted inputs and bias to form its own scalar output n(i). The various n(i) taken together form an S-element net input vector **n**. Finally, the neuron layer outputs form a column vector **a**. We show the expression for **a** at the bottom of the figure. Note that it is common for the number of inputs to a layer to be different from the number of neurons (i.e., R ¦ S). A layer is not constrained to have the number of its inputs equal to the number of its neurons.

You can create a single (composite) layer of neurons having different transfer functions simply by putting two of the networks shown earlier in parallel. Both networks would have the same inputs, and each network would create some of the outputs. The input vector elements enter the network through the weight matrix **W**.

$$\mathbf{W} = \begin{bmatrix} w_{1,1} & w_{1,2} & \cdots & w_{1,R} \\ w_{2,1} & w_{2,2} & \cdots & w_{2,R} \\ & & & \\ w_{S,1} & w_{S,2} & \cdots & w_{S,R} \end{bmatrix}$$

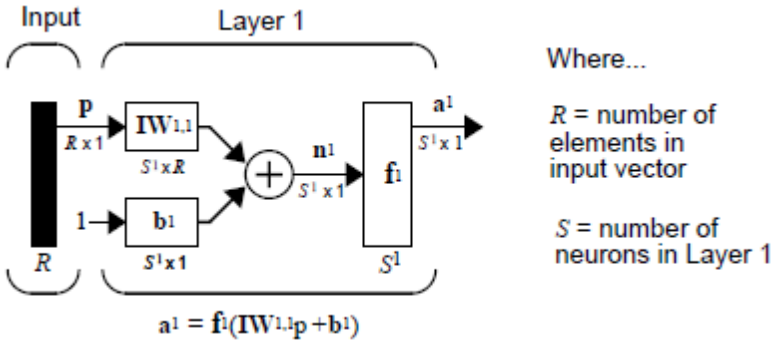
Note that the row indices on the elements of matrix **W** indicate the destination neuron of the weight, and the column indices indicate which source is the input for that weight. Thus, the indices in say that the strength of the signal from the second input element to the first (and only) neuron is $W_{1,2}$. The S neuron R input one-layer network also can be drawn in abbreviated notation.



Here **p** is an R length input vector, **W** is an SxR matrix, and **a** and **b** are S length vectors. As defined previously, the neuron layer includes the weight matrix, the multiplication operations, the bias vector **b**, the summer, and the transfer function boxes.

Inputs and Layers

We are about to discuss networks having multiple layers so we will need to extend our notation to talk about such networks. Specifically, we need to make a distinction between weight matrices that are connected to inputs and weight matrices that are connected between layers. We also need to identify the source and destination for the weight matrices. We will call weight matrices connected to inputs, input weights; and we will call weight matrices coming from layer outputs, layer weights. Further, we will use superscripts to identify the source (second index) and the destination (first index) for the various weights and other elements of the network. To illustrate, we have taken the one-layer multiple input network shown earlier and redrawn it in abbreviated form below.

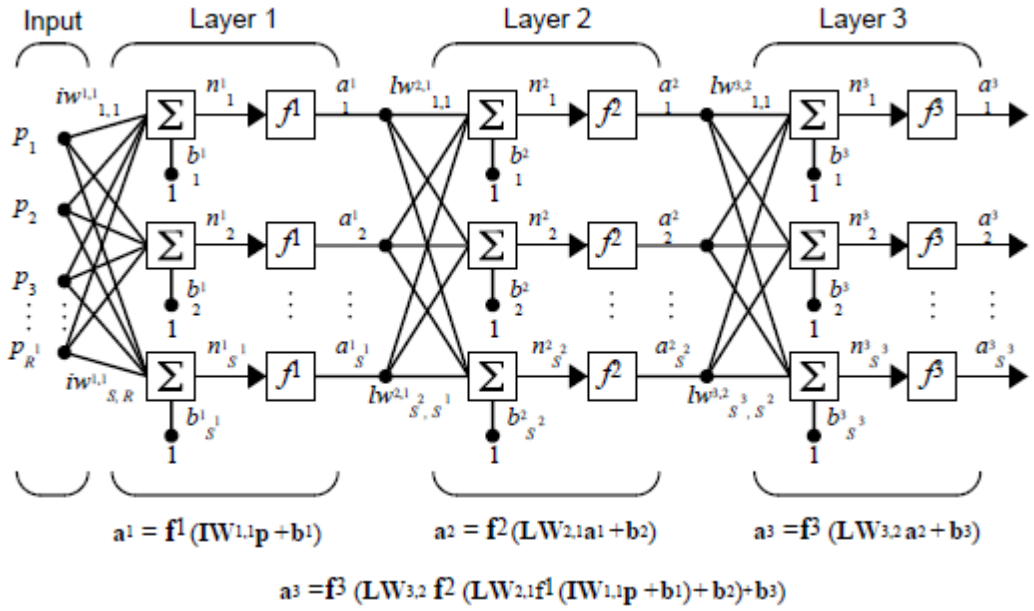


As you can see, we have labeled the weight matrix connected to the input vector **p** as an Input Weight matrix (**IW**1,1) having a source 1 (second index) and a destination 1 (first index). Also, elements of layer one, such as its bias, net input, and output have a superscript 1 to say that they are associated with the first layer.

In the next section, we will use Layer Weight (**LW**) matrices as well as Input Weight (**IW**) matrices.

Multiple Layers of Neurons

A network can have several layers. Each layer has a weight matrix **W**, a bias vector **b**, and an output vector **a**. To distinguish between the weight matrices, output vectors, etc., for each of these layers in our figures, we append the number of the layer as a superscript to the variable of interest. You can see the use of this layer notation in the three-layer network shown below, and in the equations at the bottom of the figure.

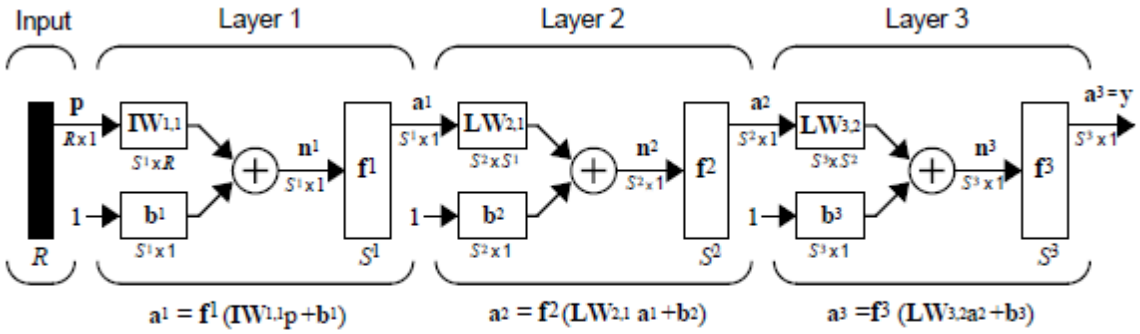


The network shown above has R1 inputs, S1 neurons in the first layer, S2 neurons in the second layer, etc. It is common for different layers to have different numbers of neurons. A constant input 1 is fed to the biases for each neuron. Note that the outputs of each intermediate layer are the inputs to the following layer. Thus layer 2 can be analyzed as a one-layer network with S1 inputs, S2 neurons, and an S2xS1 weight matrix W2. The input to layer 2 is **a**1; the output is **a**2. Now that we have identified all the vectors and matrices of layer 2, we can treat it as a single-layer network on its own. This approach can be taken with any layer of the network.

The layers of a multilayer network play different roles. A layer that produces the network output is called an output layer. All other layers are called hidden layers. The three-layer network

shown earlier has one output layer (layer 3) and two hidden layers (layer 1 and layer 2). Some authors refer to the inputs as a fourth layer. We will not use that designation.

The same three-layer network discussed previously also can be drawn using our abbreviated notation.



$$a^3 = f^3\,(LW^{3,2}\,f^2\,(LW^{2,1}f^1\,(IW^{1,1}p + b^1) + b^2) + b^3 = y$$

Multiple-layer networks are quite powerful. For instance, a network of two layers, where the first layer is sigmoid and the second layer is linear, can be trained to approximate any function (with a finite number of discontinuities) arbitrarily well. This kind of two-layer network is used extensively in Chapter 5, "Backpropagation."

Here we assume that the output of the third layer, **a**3, is the network output of interest, and we have labeled this output as **y**. We will use this notation to specify the output of multilayer networks.

Data Structures

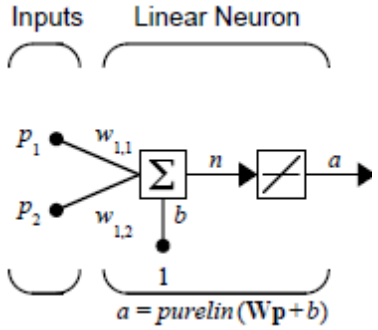
This section discusses how the format of input data structures affects the simulation of networks. We will begin with static networks, and then move to dynamic networks.

We are concerned with two basic types of input vectors: those that occur concurrently (at the same time, or in no particular time sequence), and those that occur sequentially in time. For concurrent vectors, the order is not important, and if we had a number of networks running in parallel, we could present one input vector to each of the networks. For sequential vectors, the order in which the vectors appear is important.

Simulation with Concurrent Inputs in a Static

Network

The simplest situation for simulating a network occurs when the network to be simulated is static (has no feedback or delays). In this case, we do not have to be concerned about whether or not the input vectors occur in a particular time sequence, so we can treat the inputs as concurrent. In addition, we make the problem even simpler by assuming that the network has only one input vector.

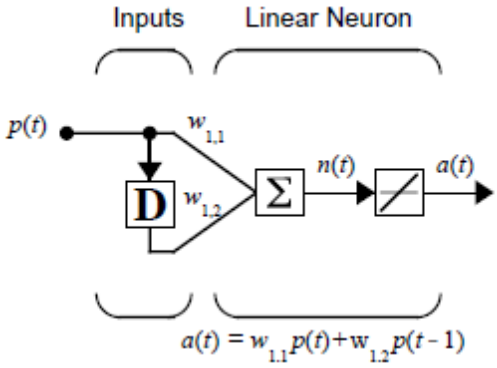


A single matrix of concurrent vectors is presented to the network and the network produces a single matrix of concurrent vectors as output. The result would be the same if there were four networks operating in parallel and each network received one of the input vectors and produced one of the outputs. The ordering of the input vectors is not important as they do not interact with each other.

Simulation with Sequential Inputs in a Dynamic

Network

When a network contains delays, the input to the network would normally be a sequence of input vectors that occur in a certain time order. To illustrate this case, we use a simple network that contains one delay.



We input a cell array containing a sequence of inputs, and the network produced a cell array containing a sequence of outputs. Note that the order of the inputs is important when they are

presented as a sequence. In this case, the current output is obtained by multiplying the current input by 1 and the preceding input by 2 and summing the result. If we were to change the order of the inputs, it would change the numbers we would obtain in the output.

Simulation with Concurrent Inputs in a Dynamic Network

If we were to apply the same inputs from the previous example as a set of concurrent inputs instead of a sequence of inputs, we would obtain a completely different response. It would be as if each input were applied concurrently to a separate parallel network.

The result is the same as if we had concurrently applied each one of the inputs to a separate network and computed one output. Note that since we did not assign any initial conditions to the network delays, they were assumed to be zero. For this case the output will simply be 1 times the input, since the weight that multiplies the current input is 1.

In certain special cases, we might want to simulate the network response to several different sequences at the same time. In this case, we would want to present the network with a concurrent set of sequences. For example, let's say

we wanted to present the following two sequences to the network:

p1 = 1 , **p**2 = 2 , **p**3 = 3 , **p**4 = 4

p1□1□□= 1 , **p**1□2□□= 2 , **p**1□3□□= 3 , **p**1□4□□= 4

p2□1□□= 4 , **p**2□2□□= 3 , **p**2□3□□= 2 , **p**2□4□□= 1

The input P should be a cell array, where each element of the array contains the two elements of the two sequences that occur at the same time:

P = {[1 4] [2 3] [3 2] [4 1]};

We can now simulate the network:

A = sim(net,P);

The resulting network output would be

A = {[1 4] [4 11] [7 8] [10 5]}

As you can see, the first column of each matrix makes up the output sequence produced by the first input sequence, which was the one we used in an earlier example. The second column of each matrix makes up the output sequence produced by the second input sequence. There is no interaction between the two concurrent sequences. It is as if they were each applied to separate networks running in parallel.

The following diagram shows the general format for the input P to the sim function when we have Q concurrent sequences of TS time steps. It covers all cases where there is a single input vector. Each element of the cell array is a matrix of concurrent vectors that correspond to the same point in time for each sequence. If there are multiple input vectors, there will be multiple rows of matrices in the cell array.

In this section, we have applied sequential and concurrent inputs to dynamic networks. In the previous section, we applied concurrent inputs to static networks. It is also possible to apply sequential inputs to static networks. It will not change the simulated response of the network, but it can affect the way in which the network is trained. This will become clear in the next section.

Module III

Neural network Models: Feed forward Neural Networks, Back propagation algorithm, Applications of Feed forward networks, Recurrent networks, Hopfield networks, Hebbian learning, Self organizing networks, unsupervised learning, competitive learning.

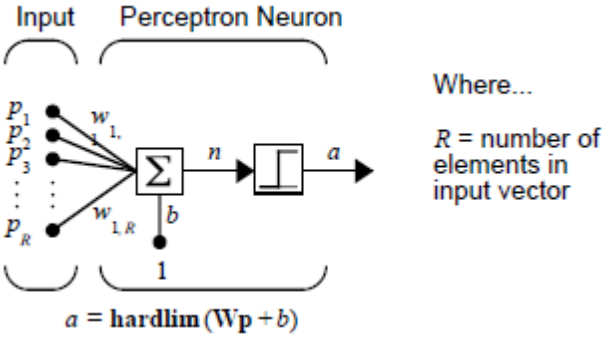
Neuro - Fuzzy Modelling: Neuro-Fuzzy inference systems, Neuro-Fuzzy control

Perceptrons

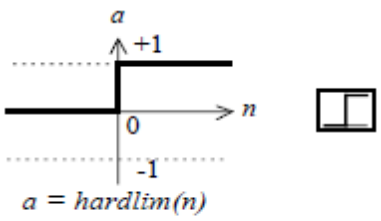
Rosenblatt [Rose61] created many variations of the perceptron. One of the simplest was a single-layer network whose weights and biases could be trained to produce a correct target vector when presented with the corresponding input vector. The training technique used is called the perceptron learning rule. The perceptron generated great interest due to its ability to generalize from its training vectors and learn from initially randomly distributed connections. Perceptrons are especially suited for simple problems in pattern classification. They are fast and reliable networks for the problems they can solve. In addition, an understanding of the operations of the perceptron provides a good basis for understanding more complex networks.

Neuron Model

A perceptron neuron, which uses the hard-limit transfer function hardlim, is shown below.

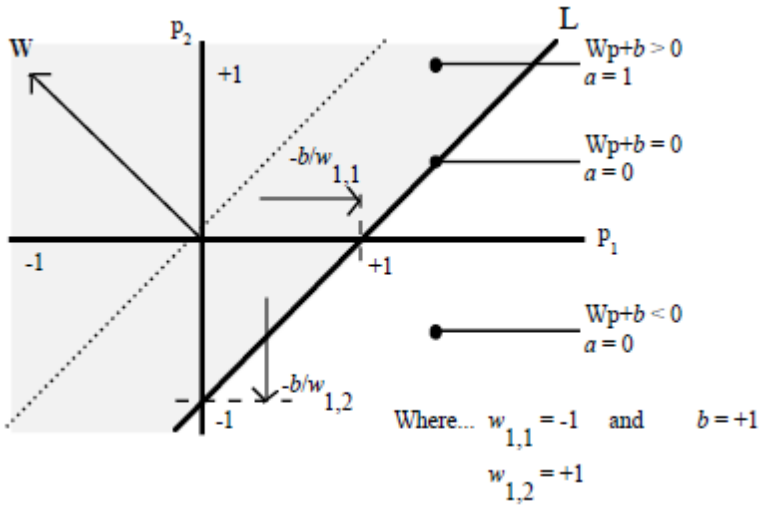


Each external input is weighted with an appropriate weight w1j, and the sum of the weighted inputs is sent to the hard-limit transfer function, which also has an input of 1 transmitted to it through the bias. The hard-limit transfer function, which returns a 0 or a 1, is shown below.



Hard-Limit Transfer Function

The perceptron neuron produces a 1 if the net input into the transfer function is equal to or greater than 0; otherwise it produces a 0. The hard-limit transfer function gives a perceptron the ability to classify input vectors by dividing the input space into two regions. Specifically, outputs will be 0 if the net input n is less than 0, or 1 if the net input n is 0 or greater. The input space of a two-input hard limit neuron with the weights and a bias, is shown below.



Two classification regions are formed by the decision boundary line L at. This line is perpendicular to the weight matrix **W** and shifted according to the bias b. Input vectors above and to the left of the line L will result in a net input greater than 0; and therefore, cause the hard-limit neuron to output a 1. Input vectors below and to the right of the line L cause the neuron to output 0. The dividing line can be oriented and moved anywhere to classify the input space as desired by picking the weight and bias values.

Hard-limit neurons without a bias will always have a classification line going through the origin. Adding a bias allows the neuron to solve problems where the two sets of input vectors are not

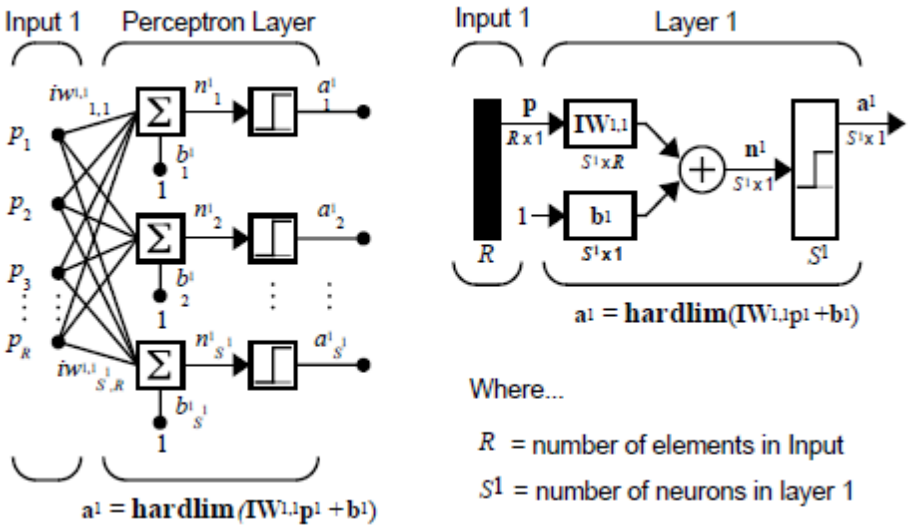
located on different sides of the origin. The bias allows the decision boundary to be shifted away from the origin as shown

in the plot above.

You may want to run the demonstration program nnd4db. With it you can move a decision boundary around, pick new inputs to classify, and see how the repeated application of the learning rule yields a network that does classify the input vectors properly.

Perceptron Architecture

The perceptron network consists of a single layer of S perceptron neurons connected to R inputs through a set of weights wi,j as shown below in two forms. As before, the network indices i and j indicate that wi,j is the strength of the connection from the jth input to the ith neuron.



The perceptron learning rule that we will describe shortly is capable of training only a single layer. Thus, here we will consider only one-layer networks. This restriction places limitations on the computation a perceptron can perform. The types of problems that perceptrons are capable of solving are discussed later in this chapter in the "Limitations and Cautions" section.

Learning Rules

We define a learning rule as a procedure for modifying the weights and biases of a network. (This procedure may also be referred to as a training algorithm.) The learning rule is applied to train the network to perform some particular task. Learning rules in this toolbox fall into two

broad categories: supervised learning, and unsupervised learning. In supervised learning, the learning rule is provided with a set of examples (the training set) of proper network behavior

$$\{\mathbf{p}_1, \mathbf{t}_1\}, \{\mathbf{p}_2, \mathbf{t}_2\}, \ldots, \{\mathbf{p}_Q, \mathbf{t}_Q\}$$

where is an input to the network, and is the corresponding correct (target) output. As the inputs are applied to the network, the network outputs are compared to the targets. The learning rule is then used to adjust the weights and biases of the network in order to move the network outputs closer to the targets. The perceptron learning rule falls in this supervised learning category. In unsupervised learning, the weights and biases are modified in response to network inputs only. There are no target outputs available. Most of these algorithms perform clustering operations. They categorize the input patterns into a finite number of classes. This is especially useful in such applications as vector quantization. As noted, the perceptron discussed in this chapter is trained with supervised learning. Hopefully, a network that produces the right output for a particular input will be obtained.

Perceptron Learning Rule (learnp)

Perceptrons are trained on examples of desired behavior. The desired behavior can be summarized by a set of input, output pairs where **p** is an input to the network and **t** is the corresponding correct (target) output. The objective is to reduce the error **e**, which is the difference between the neuron response **a,** and the target vector **t**. The perceptron learning rule learnp calculates desired changes to the perceptron's weights and biases given an input vector **p**, and the associated error **e**. The target vector **t** must contain values of either 0 or 1, as perceptrons (with hardlim transfer functions) can only output such values.

Each time learnp is executed, the perceptron has a better chance of producing the correct outputs. The perceptron rule is proven to converge on a solution in a finite number of iterations if a solution exists.

If a bias is not used, learnp works to find a solution by altering only the weight vector **w** to point toward input vectors to be classified as 1, and away from vectors to be classified as 0. This results in a decision boundary that is perpendicular to **w**, and which properly classifies the input vectors.

There are three conditions that can occur for a single neuron once an input vector **p** is presented and the network's response **a** is calculated:

CASE 1. If an input vector is presented and the output of the neuron is correct (**a** = **t**, and **e** = **t** – **a** = 0), then the weight vector **w** is not altered.

CASE 2. If the neuron output is 0 and should have been 1 (**a** = 0 and **t** = 1, and **e** = **t** – **a** = 1), the input vector **p** is added to the weight vector **w**. This makes the weight vector point closer to the input vector, increasing the chance that the input vector will be classified as a 1 in the future.

CASE 3. If the neuron output is 1 and should have been 0 (**a** = 1and **t** = 0, and **e** = **t** – **a** = –1), the input vector **p** is subtracted from the weight vector **w**. This makes the weight vector point farther away from the input vector, increasing the chance that the input vector is classified as a 0 in the future.

The perceptron learning rule can be written more succinctly in terms of the error **e** = **t** – **a**, and the change to be made to the weight vector Δ**w**:

CASE 1. If **e** = 0, then make a change □**w** equal to 0.

CASE 2. If **e** = 1, then make a change □**w** equal to **p**T.

CASE 3. If **e** = –1, then make a change □**w** equal to –**p**T.

All three cases can then be written with a single expression:

$$\Delta \mathbf{w} = (t-a)\mathbf{p}^{\mathrm{T}} = e\mathbf{p}^{\mathrm{T}}$$

We can get the expression for changes in a neuron's bias by noting that the bias is simply a weight that always has an input of 1:

$$\Delta b = (t-a)(1) = e$$

For the case of a layer of neurons we have:

$$\Delta \mathbf{W} = (\mathbf{t}-\mathbf{a})(\mathbf{p})^{\mathrm{T}} = \mathbf{e}(\mathbf{p})^{\mathrm{T}} \text{ and}$$

$$\Delta \mathbf{b} = (\mathbf{t}-\mathbf{a}) = \mathbf{E}$$

The Perceptron Learning Rule can be summarized as follows

$$\mathbf{W}^{new} = \mathbf{W}^{old} + \mathbf{e}\mathbf{p}^{T} \text{ and}$$

$$\mathbf{b}^{new} = \mathbf{b}^{old} + \mathbf{e}$$

where

$$\mathbf{e} = \mathbf{t} - \mathbf{a}.$$

The process of finding new weights (and biases) can be repeated until there are no errors. Note that the perceptron learning rule is guaranteed to converge in a finite number of steps for all

problems that can be solved by a perceptron. These include all classification problems that are "linearly separable." The objects to be classified in such cases can be separated by a single line.

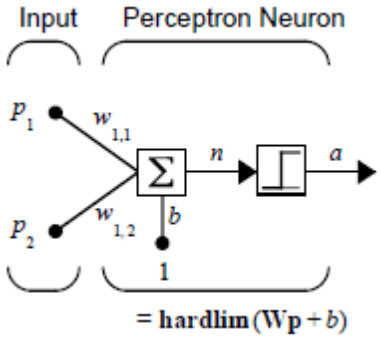
Training (train)

If sim and learnp are used repeatedly to present inputs to a perceptron, and to change the perceptron weights and biases according to the error, the perceptron will eventually find weight and bias values that solve the problem, given that the perceptron can solve it. Each traverse through all of the training input and target vectors is called a pass. The function train carries out such a loop of calculation. In each pass the function train proceeds through the specified sequence of inputs, calculating the output, error and network adjustment for each input vector in the sequence as the inputs are presented. Note that train does not guarantee that the resulting network does its job.

The new values of **W** and **b** must be checked by computing the network output for each input vector to see if all targets are reached. If a network does not perform successfully it can be trained further by again calling train with the new weights and biases for more training passes, or the problem can be analyzed to see if it is a suitable problem for the perceptron. Problems which are not solvable by the perceptron network are discussed in the "Limitations and Cautions" section.

To illustrate the training procedure, we will work through a simple problem.

Consider a one neuron perceptron with a single vector input having two elements.



Let us suppose we have the following classification problem and would like to solve it with our single vector input, two-element perceptron network.

$$\left\{\mathbf{p}_1 = \begin{bmatrix}2\\2\end{bmatrix}, t_1 = 0\right\} \left\{\mathbf{p}_2 = \begin{bmatrix}1\\-2\end{bmatrix}, t_2 = 1\right\} \left\{\mathbf{p}_3 = \begin{bmatrix}-2\\2\end{bmatrix}, t_3 = 0\right\} \left\{\mathbf{p}_4 = \begin{bmatrix}-1\\1\end{bmatrix}, t_4 = 1\right\}$$

Use the initial weights and bias. We denote the variables at each step of this calculation by using a number in parentheses after the variable. Thus, above, we have the initial values, **W**(0) and b(0).

$$\mathbf{W}(0) = \begin{bmatrix} 0 & 0 \end{bmatrix} \qquad b(0) = 0$$

We start by calculating the perceptron's output a for the first input vector **p1**, using the initial weights and bias.

$$a = hardlim(\mathbf{W}(0)\mathbf{p}_1 + b(0))$$

$$= hardlim\left(\begin{bmatrix} 0 & 0 \end{bmatrix}\begin{bmatrix} 2 \\ 2 \end{bmatrix} + 0\right) = hardlim(0) = 1$$

The output a does not equal the target value t1, so we use the perceptron rule to find the incremental changes to the weights and biases based on the error.

$$e = t_1 - a = 0 - 1 = -1$$
$$\Delta\mathbf{W} = e\mathbf{p}_1^T = (-1)\begin{bmatrix} 2 & 2 \end{bmatrix} = \begin{bmatrix} -2 & -2 \end{bmatrix}$$
$$\Delta b = e = (-1) = -1$$

You can calculate the new weights and bias using the Perceptron update rules shown previously.

$$\mathbf{W}^{new} = \mathbf{W}^{old} + \mathbf{e}\mathbf{p}^T = \begin{bmatrix} 0 & 0 \end{bmatrix} + \begin{bmatrix} -2 & -2 \end{bmatrix} = \begin{bmatrix} -2 & -2 \end{bmatrix} = \mathbf{W}(1)$$

$$b^{new} = b^{old} + e = 0 + (-1) = -1 = b(1)$$

Now present the next input vector, **p2.** The output is calculated below.

$$a = hardlim(\mathbf{W}(1)\mathbf{p}_2 + b(1))$$

$$= hardlim\left(\begin{bmatrix} -2 & -2 \end{bmatrix}\begin{bmatrix} -2 \\ -2 \end{bmatrix} - 1\right) = hardlim(1) = 1$$

On this occasion, the target is 1, so the error is zero. Thus there are no changes in weights or bias, so $\mathbf{W}(2) = \mathbf{W}(1) = \begin{bmatrix} -2 & -2 \end{bmatrix}$ and $b(2) = b(1) = -1$

We can continue in this fashion, presenting **p3** next, calculating an output and the error, and making changes in the weights and bias, etc. After making one pass through all of the four inputs, you get the values: and. To determine if we obtained a satisfactory solution, we must make one pass through all input vectors to see if they all produce the desired target values. This is not true for the 4th input, but the algorithm does converge on the 6th presentation of an input. The final values are:

$\mathbf{W}(6) = \begin{bmatrix} -2 & -3 \end{bmatrix}$ and $b(6) = 1$

LMS Algorithm (learnwh)

The LMS algorithm or Widrow-Hoff learning algorithm, is based on an approximate steepest descent procedure. Here again, linear networks are trained on examples of correct behavior.

Widrow and Hoff had the insight that they could estimate the mean square error by using the squared error at each iteration. If we take the partial derivative of the squared error with respect to the weights and biases at the kth iteration we have

$$\frac{\partial e^2(k)}{\partial w_{1,j}} = 2e(k)\frac{\partial e(k)}{\partial w_{1,j}}$$

for $j = 1, 2, \ldots, R$ and

$$\frac{\partial e^2(k)}{\partial b} = 2e(k)\frac{\partial e(k)}{\partial b}$$

Next look at the partial derivative with respect to the error.

$$\frac{\partial e(k)}{\partial w_{1,j}} = \frac{\partial [t(k) - a(k)]}{\partial w_{1,j}} = \frac{\partial}{\partial w_{1,j}}[t(k) - (\mathbf{W}\mathbf{p}(k) + b)] \text{ or}$$

$$\frac{\partial e(k)}{\partial w_{1,j}} = \frac{\partial}{\partial w_{1,j}} \left[t(k) - \left(\sum_{i=1}^{R} w_{1,i} p_i(k) + b \right) \right]$$

Here pi(k) is the ith element of the input vector at the kth iteration.

Similarly,

$$\frac{\partial e(k)}{\partial w_{1,j}} = -p_j(k)$$

This can be simplified to:

$$\frac{\partial e(k)}{\partial w_{1,j}} = -p_j(k) \text{ and}$$

$$\frac{\partial e(k)}{\partial b} = -1$$

Finally, the change to the weight matrix and the bias will be

$$2\alpha e(k)\mathbf{p}(k) \text{ and } 2\alpha e(k).$$

These two equations form the basis of the Widrow-Hoff (LMS) learning algorithm.

These results can be extended to the case of multiple neurons, and written in matrix form as

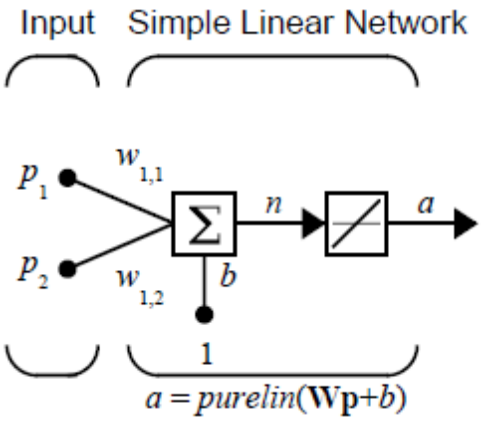
$$\mathbf{W}(k+1) = \mathbf{W}(k) + 2\alpha \mathbf{e}(k)\mathbf{p}^{T}(k)$$

$$\mathbf{b}(k+1) = \mathbf{b}(k) + 2\alpha \mathbf{e}(k)$$

Here the error **e** and the bias **b** are vectors and is a learning rate. If is large, learning occurs quickly, but if it is too large it may lead to instability and errors may even increase. To ensure stable learning, the learning rate must be less than the reciprocal of the largest eigenvalue of the correlation matrix of the input vectors.

Linear Classification (train)

Linear networks can be trained to perform linear classification with the function train. This function applies each vector of a set of input vectors and calculates the network weight and bias increments due to each of the inputs according to learnp. Then the network is adjusted with the sum of all these corrections. We will call each pass through the input vectors an epoch. Finally, train applies the inputs to the new network, calculates the outputs, compares them to the associated targets, and calculates a mean square error. If the error goal is met, or if the maximum number of epochs is reached, the training is stopped and train returns the new network and a training record. Otherwise train goes through another epoch. Fortunately, the LMS algorithm converges when this procedure is executed.



MULTllAYER FEED-FORWARD NETWORKS

Figure 7 shows a typical three-layer perceptron. In general, a standard L-layer feed-forward network (we adopt the convention that the input nodes are not counted as a layer) consists of an input stage, (L-1) hidden layers, and an output layer of units successively connected (fully or locally) in a feed-forward fashion with no connections between units in the same layer and no feedback connections between layers.

Multilayer perceptron

The most popular class of multilayer feed-forward networks is multilayer perceptrons in which each computational unit employs either the thresholding function or the sigmoid function. Multilayer perceptrons can form arbitrarily complex decision boundaries and represent any Boolean function.6 The development of the back-propagation learning algorithm for determining weights in a multilayer perceptron has made these networks the most popular among researchers and users of neural networks. We denote w,I(l) as the weight on the connection between the ith unit in layer **(2-1)** to jth unit in layer 1. Let **{(x(l)d, c1))(, x(2)d, @))., . . , (xb), d(p)) }** be a set ofp training patterns (input-output pairs), where **x(l) E** R" is the input vector in the n-dimensional pattern space, and **d(l)** E [0, **l]",** an m-dimensional hypercube. For classification purposes, m is the number of classes. The squared error cost function most frequently used in the ANN literature is defined as

$$E = \frac{1}{2}\sum_{i=1}^{p} \left\| \mathbf{y}^{(i)} - \mathbf{d}^{(i)} \right\|^2$$

The back-propagation algorithm9 is a gradient-descent method to minimize the squared-error cost function in Equation 2 (see "Back-propagation algorithm" sidebar). A geometric interpretation (adopted and modified from Lippmann") shown in Figure 8 can help explicate the role of hidden units (with the threshold activation function). Each unit in the first hidden layer forms a hyper plane in the pattern space; boundaries between pattern classes can be approximated by hyper planes. A unit in the second hidden layer forms a hyper region from the outputs of the first-layer units; a decision region is obtained by performing an AND operation on the hyper planes. The output-layer units combine the decision regions made by the units in the second hidden layer by performing logical OR operations. Remember that this scenario is depicted only to explain the role of hidden units. Their actual behavior, after the network is trained, could differ. A two-layer network can form more complex decision boundaries than those shown in Figure 8. Moreover, multilayer perceptrons with sigmoid activation functions can form smooth decision boundaries rather than piecewise linear boundaries.

4. The Backpropagation Algorithm

The backpropagation algorithm (Rumelhart and McClelland, 1986) is used in layered feed-forward ANNs. This means that the artificial neurons are organized in layers, and send their signals “forward”, and then the errors are propagated backwards. The network receives inputs by neurons in the input layer, and the output of the network is given by the neurons on an output layer. There may be one or more intermediate hidden layers.

The backpropagation algorithm uses supervised learning, which means that we provide the algorithm with examples of the inputs and outputs we want the network to compute, and then the error (difference between actual and expected results) is calculated. The idea of the backpropagation algorithm is to reduce this error, until the ANN learns the training data. The training begins with random weights, and the goal is to adjust them so that the error will be minimal.

The activation function of the artificial neurons in ANNs implementing the backpropagation algorithm is a weighted sum (the sum of the inputs xi multiplied by their respective weights w ji):

$$A_j(\overline{x}, \overline{w}) = \sum_{i=0}^{n} x_i w_{ji} \qquad (1)$$

We can see that the activation depends only on the inputs and the weights.

If the output function would be the identity (output=activation), then the neuron would be called linear. But these have severe limitations. The most common output function is the sigmoidal function:

$$O_j(\overline{x}, \overline{w}) = \frac{1}{1 + e^{A_j(\overline{x}, \overline{w})}} \qquad (2)$$

The sigmoidal function is very close to one for large positive numbers, 0.5 at zero, and very close to zero for large negative numbers. This allows a smooth transition between the low and high output of the neuron (close to zero or close to one). We can see that the output depends only in the activation, which in turn depends on the values of the inputs and their respective weights.

Now, the goal of the training process is to obtain a desired output when certain inputs are given. Since the error is the difference between the actual and the desired output, the error depends on the weights, and we need to adjust the weights in order to minimize the error. We can define the error function for the output of each neuron:

$$E_j(\overline{x}, \overline{w}, d) = \left(O_j(\overline{x}, \overline{w}) - d_j\right)^2 \qquad (3)$$

We take the square of the difference between the output and the desired target because it will be always positive, and because it will be greater if the difference is big, and lesser if the difference is small. The error of the network will simply be the sum of the errors of all the neurons in the output layer:

$$E(\overline{x}, \overline{w}, \overline{d}) = \sum_j \left(O_j(\overline{x}, \overline{w}) - d_j\right)^2 \qquad (4)$$

The backpropagation algorithm now calculates how the error depends on the output, inputs, and weights. After we find this, we can adjust the weights using the method of gradient descendent:

$$\Delta w_{ji} = -\eta \frac{\partial E}{\partial w_{ji}} \qquad (5)$$

This formula can be interpreted in the following way: the adjustment of each weight (Δw ji) will be the negative of a constant eta (η) multiplied by the dependance of the previous weight on the error of the network, which is the derivative of E in respect to w i .

The size of the adjustment will depend on 0, and on the contribution of the weight to the error of the function. This is, if the weight contributes a lot to the error, the adjustment will be greater than if it contributes in a smaller amount. (5) is used until we find appropriate weights (the error is minimal). If you do not know derivatives, don't worry, you can see them now as functions that we will replace right away with algebraic expressions. If you understand derivatives, derive the expressions yourself and compare your results with the ones presented here. If you are searching for a mathematical proof of the backpropagation algorithm, you are advised to check it in the suggested reading, since this is out of the scope of this material.

So, we "only" need to find the derivative of E in respect to w ji . This is the goal of the backpropagation algorithm, since we need to achieve this backwards. First, we need to calculate how much the error depends on the output, which is the derivative of E in respect to O j (from (3)).

$$\frac{\partial E}{\partial O_j} = 2\left(O_j - d_j\right) \qquad (6)$$

And then, how much the output depends on the activation, which in turn depends on the weights (from (1) and (2)):

$$\frac{\partial O_j}{\partial w_{ji}} = \frac{\partial O_j}{\partial A_j}\frac{\partial A_j}{\partial w_{ji}} = O_j(1 - O_j)x_i \qquad (7)$$

And we can see that (from (6) and (7)):

$$\frac{\partial E}{\partial w_{ji}} = \frac{\partial E}{\partial O_j}\frac{\partial O_j}{\partial w_{ji}} = 2\left(O_j - d_j\right)O_j(1-O_j)x_i \qquad (8)$$

And so, the adjustment to each weight will be (from (5) and (8)):

$$\Delta w_{ji} = -2\eta\left(O_j - d_j\right)O_j(1-O_j)x_i \qquad (9)$$

We can use (9) as it is for training an ANN with two layers. Now, for training the network with one more layer we need to make some considerations. If we want to adjust the weights (let's call them v ik) of a previous layer, we need first to calculate how the error depends not on the weight, but in the input from the previous layer. This is easy, we would just need to change x i with w ji in (7), (8), and (9). But we also need to see how the error of
the network depends on the adjustment of v ik . So:

$$\Delta v_{ik} = -\eta\frac{\partial E}{\partial v_{ik}} = -\eta\frac{\partial E}{\partial x_i}\frac{\partial x_i}{\partial v_{ik}} \qquad (10)$$

Where:

$$\frac{\partial E}{\partial w_{ji}} = 2\left(O_j - d_j\right)O_j(1-O_j)w_{ji} \qquad (11)$$

And, assuming that there are inputs uk into the neuron with v ik (from (7)):

$$\frac{\partial x_i}{\partial v_{ik}} = x_i(1-x_i)v_{ik} \qquad (12)$$

If we want to add yet another layer, we can do the same, calculating how the error depends on the inputs and weights of the first layer. We should just be careful with the indexes, since each layer can have a different number of neurons, and we should not confuse them.

For practical reasons, ANNs implementing the backpropagation algorithm do not have too many layers, since the time for training the networks grows exponentially. Also, there are refinements to the backpropagation algorithm which allow a faster learning.

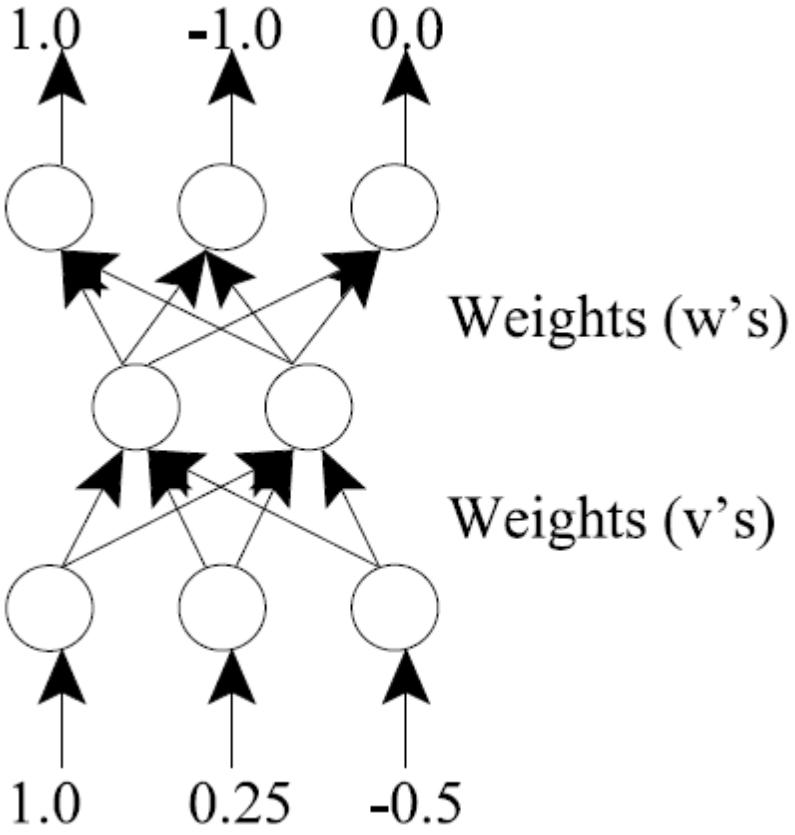
4.1. Exercise

If you know how to program, implement the backpropagation algorithm, that at least will train the following network. If you can do a general implementation of the
backpropagation algorithm, go ahead (for any number of neurons per layer, training sets, and even layers).

If you do not know how to program, but know how to use a mathematical assistant (such as Matlab or Mathematica), find weights which will suit the following network after defining functions which will ease your task.

If you do not have any computing experience, find the weights by hand.

The network for this exercise has three neurons in the input layer, two neurons in a hidden layer, and three neurons in the output layer. Usually networks are trained with large training sets, but for this exercise, we will only use one training example. When the inputs are (1, 0.25, -0.5), the outputs should be (1,-1,0). Remember you start with random weights.



Backpropagation

Backpropagation was created by generalizing the Widrow-Hoff learning rule to multiple-layer networks and nonlinear differentiable transfer functions. Input vectors and the corresponding target vectors are used to train a network until it can approximate a function, associate input vectors with specific output vectors, or classify input vectors in an appropriate way as defined by you. Networks with biases, a sigmoid layer, and a linear output layer are capable of approximating any function with a finite number of discontinuities. Standard backpropagation is a gradient descent algorithm, as is the Widrow-Hoff learning rule, in which the network weights are moved along the negative of the gradient of the performance function. The term backpropagation refers to the manner in which the gradient is computed for nonlinear multilayer networks. There are a number of variations on the basic algorithm that are based on other standard optimization techniques, such as conjugate gradient and Newton methods. The Neural Network Toolbox implements a number of these variations. This chapter explains how to use each of these routines and discusses the advantages and disadvantages of each. Properly trained backpropagation networks tend to give reasonable answers when presented with inputs that they have never seen. Typically, a new input leads to an output similar to the correct output for input vectors used in training that are similar to the new input being presented. This generalization property makes it possible to train a network on a representative set of input/ target pairs and get good results without training the network on all possible input/output pairs. There are two features of the Neural Network Toolbox that are designed to improve network generalization - regularization and early stopping. These features and their use are discussed later in this chapter.

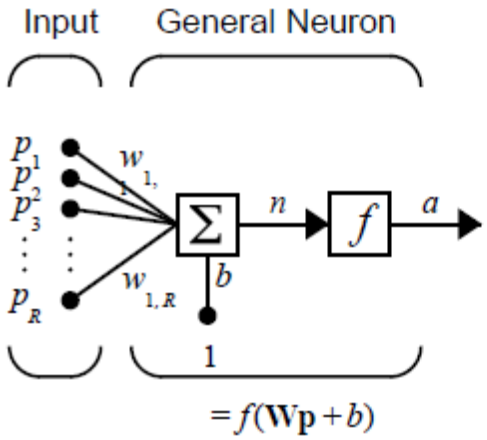
Fundamentals

Architecture

This section presents the architecture of the network that is most commonly used with the backpropagation algorithm - the multilayer feedforward network. The routines in the Neural Network Toolbox can be used to train more general networks; some of these will be briefly discussed in later chapters.

Neuron Model (tansig, logsig, purelin)

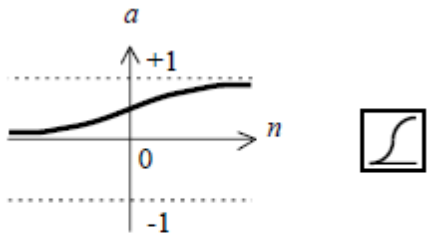
An elementary neuron with R inputs is shown below. Each input is weighted with an appropriate w. The sum of the weighted inputs and the bias forms the input to the transfer function f. Neurons may use any differentiable transfer function f to generate their output.



Where...

R = Number of elements in input vector

Multilayer networks often use the log-sigmoid transfer function logsig.

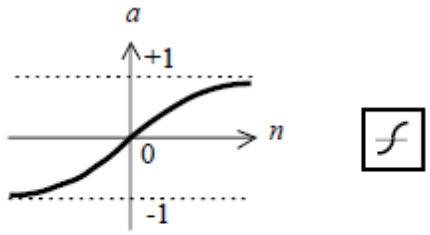


$a = logsig(n)$

Log-Sigmoid Transfer Function

The function logsig generates outputs between 0 and 1 as the neuron's net input goes from negative to positive infinity.

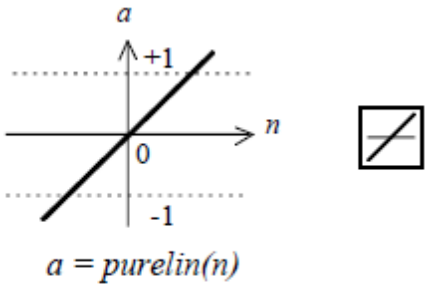
Alternatively, multilayer networks may use the tan-sigmoid transfer function tansig.



$a = tansig(n)$

Tan-Sigmoid Transfer Function

Occasionally, the linear transfer function purelin is used in backpropagation networks.



$a = purelin(n)$

Linear Transfer Function

If the last layer of a multilayer network has sigmoid neurons, then the outputs of the network are limited to a small range. If linear output neurons are used the network outputs can take on any value.

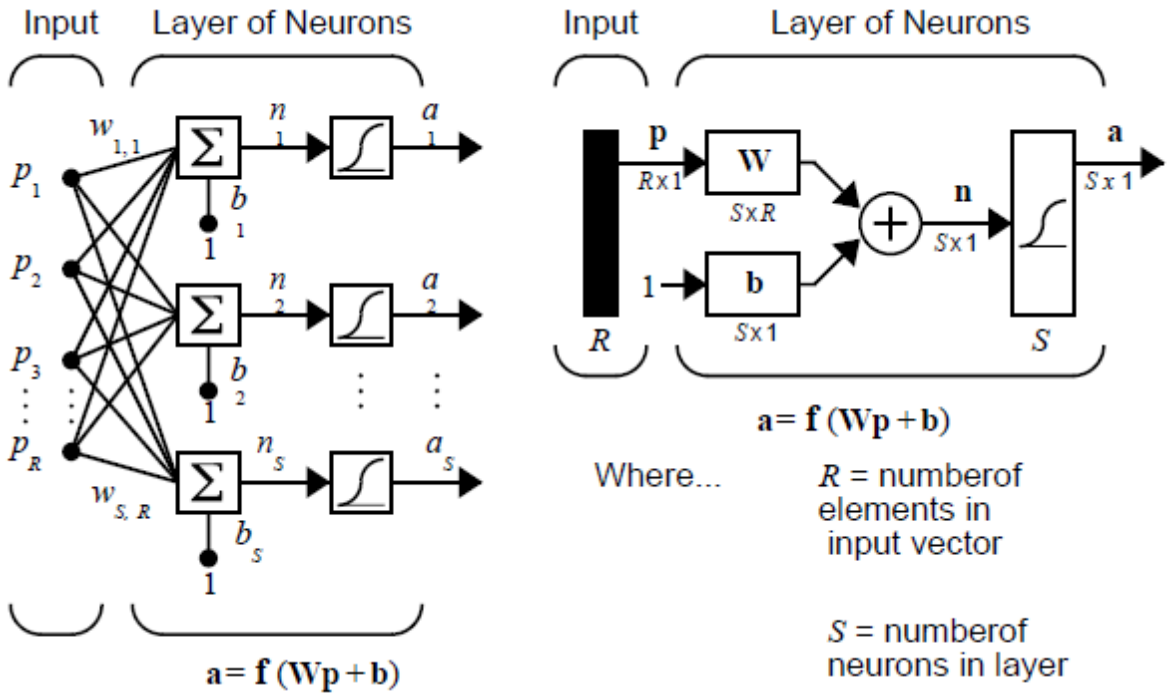
In backpropagation it is important to be able to calculate the derivatives of any transfer functions used. Each of the transfer functions above, tansig, logsig, and purelin, have a corresponding derivative function: dtansig, dlogsig, and

dpurelin. To get the name of a transfer function's associated derivative function, call the transfer function with the string 'deriv'.

The three transfer functions described here are the most commonly used transfer functions for backpropagation, but other differentiable transfer functions can be created and used with backpropagation if desired.

Feedforward Network

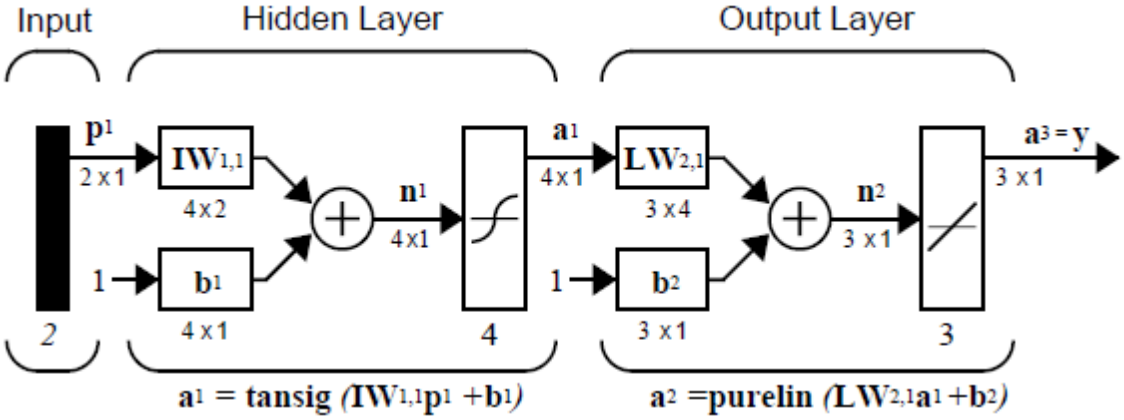
A single-layer network of S logsig neurons having R inputs is shown below in full detail on the left and with a layer diagram on the right.



Feedforward networks often have one or more hidden layers of sigmoid neurons followed by an output layer of linear neurons. Multiple layers of neurons with nonlinear transfer functions allow the network to learn nonlinear and linear relationships between input and output vectors. The linear output layer lets the network produce values outside the range –1 to +1.

On the other hand, if you want to constrain the outputs of a network (such as between 0 and 1), then the output layer should use a sigmoid transfer function (such as logsig).

For multiple-layer networks we use the number of the layers to determine the superscript on the weight matrices. The appropriate notation is used in the two-layer tansig/purelin network shown next.



This network can be used as a general function approximator. It can approximate any function with a finite number of discontinuities, arbitrarily well, given sufficient neurons in the hidden layer.

Backpropagation Algorithm

There are many variations of the backpropagation algorithm, several of which we discuss in this chapter. The simplest implementation of backpropagation learning updates the network weights and biases in the direction in which the performance function decreases most rapidly - the negative of the gradient. One iteration of this algorithm can be written

$$\mathbf{x}_{k+1} = \mathbf{x}_k - \alpha_k \mathbf{g}_k$$

where is a vector of current weights and biases, is the current gradient, and is the learning rate.

There are two different ways in which this gradient descent algorithm can be implemented: incremental mode and batch mode. In the incremental mode, the gradient is computed and the weights are updated after each input is applied to the network. In the batch mode all of the inputs are applied to the network before the weights are updated. The next section describes the batch mode of training; incremental training will be discussed in a later chapter.

Batch Training (train). In batch mode the weights and biases of the network are updated only after the entire training set has been applied to the network. The gradients calculated at each training example are added together to determine the change in the weights and biases.

Batch Gradient Descent (traingd). The batch steepest descent training function is traingd. The weights and biases are updated in the direction of the negative gradient of the performance function. If you want to train a network using batch steepest descent, you should set the network trainFcn to traingd, and then call the function train. There is only one training function associated with a given network.

There are seven training parameters associated with traingd: epochs, show, goal, time, min_grad, max_fail, and lr. The learning rate lr is multiplied times the negative of the gradient to determine the changes to the weights and biases. The larger the learning rate, the bigger the step. If the learning rate is made too large, the algorithm becomes unstable. If the learning rate is set too small, the algorithm takes a long time to converge.

The training status is displayed for every show iteration of the algorithm. The other parameters determine when the training stops. The training stops if the number of iterations exceeds epochs,

if the performance function drops below goal, if the magnitude of the gradient is less than mingrad, or if the training time is longer than time seconds. We discuss max_fail, which is associated with the early stopping technique, in the section on improving generalization.

Batch Gradient Descent with Momentum (traingdm). In addition to traingd, there is another batch algorithm for feedforward networks that often provides faster convergence - traingdm, steepest descent with momentum. Momentum allows a network to respond not only to the local gradient, but also to recent trends in the error surface. Acting like a low-pass filter, momentum allows the network to ignore small features in the error surface. Without momentum a network may get stuck in a shallow local minimum. With momentum a network can slide through such a minimum.

Momentum can be added to backpropagation learning by making weight changes equal to the sum of a fraction of the last weight change and the new change suggested by the backpropagation rule. The magnitude of the effect that the last weight change is allowed to have is mediated by a momentum constant, mc, which can be any number between 0 and 1. When the momentum constant is 0, a weight change is based solely on the gradient. When the momentum constant is 1, the new weight change is set to equal the last weight change and the gradient is simply ignored. The gradient is computed by summing the gradients calculated at each training example, and the weights and biases are only updated after all training examples have been presented.

If the new performance function on a given iteration exceeds the performance function on a previous iteration by more than a predefined ratio max_perf_inc (typically 1.04), the new weights and biases are discarded, and the momentum coefficient mc is set to zero.

The batch form of gradient descent with momentum is invoked using the training function traingdm. The traingdm function is invoked using the same steps shown above for the traingd function, except that the mc, lr and

max_perf_inc learning parameters can all be set.

Radial Basis Neworks

Radial basis networks may require more neurons than standard feed-forward backpropagation networks, but often they can be designed in a fraction of the time it takes to train standard feed-forward networks. They work best when many training vectors are available.

Radial Basis Function network

The Radial Basis Function (RBF) network,3 which has two layers, is a special class of multilayer feed-forward networks. Each unit in the hidden layer employs a radial basis function, such as a Gaussian kernel, as the activation function. The radial basis function (or kernel function) is centered at the point specified by the weight vector associated with the unit. Both the positions and the widths of these kernels must be learned from training patterns. There are usually many fewer kernels in the RBF network than there are training patterns. Each output unit implements a linear combination of these radial basis functions. From the point of view of function approximation, the hidden units provide a set of functions that constitute a basis set for representing input patterns in the space spanned by the hidden units.

There are a variety of learning algorithms for the RBFnetwork.3 The basic one employs a two-step learning strategy, or hybrid learning. It estimates kernel positions and kernel widths using an unsupervised clustering algorithm, followed by a supervised least mean square (LMS) algorithm to determine the connection weights between the hidden layer and the output layer. Because the output units are linear, a non-iterative algorithm can be used. After this initial solution is obtained, a supervised gradient-based algorithm can be used to refine the network parameters.

This hybrid learning algorithm for training the RBF network converges much faster than the back-propagation algorithm for training multilayer perceptrons. However, for many problems, the RBF network often involves a larger number of hidden units. This implies that the runtime (after training) speed of the RBF network is often slower than the runtime speed of a multilayer perceptron.

The efficiencies (error versus network size) of the RBF network and the multilayer perceptron are, however, problem-dependent. It has been shown that the RBF network has the same asymptotic approximation power as a multilayer perceptron.

Radial Basis Functions

Neuron Model

Here is a radial basis network with R inputs. works. Each unit in the hidden layer employs a radial basis function, such as a Gaussian kernel, as the activation function. The radial basis function (or kernel function) is centered at the point specified by the weight vector associated with the unit. Both the positions and the widths of these kernels must be learned from training patterns. There are usually many fewer kernels in the RBF network than there are training patterns. Each output unit implements a linear combination of these radial basis functions. From the point of view of function approximation, the hidden units provide a set of functions that constitute a basis set for representing input patterns in the space spanned by the hidden units.

There are a variety of learning algorithms for the RBF network.3 The basic one employs a two-step learning strategy, or hybrid learning. It estimates kernel positions and kernel widths using an unsupervised clustering algorithm, followed by a supervised least mean square (LMS) algorithm to determine the connection weights between the hidden layer and the output layer. Because the output units are linear, a non-iterative algorithm can be used. After this initial solution is obtained, a supervised gradient-based algorithm can be used to refine the network parameters.

This hybrid learning algorithm for training the RBF network converges much faster than the back-propagation algorithm for training multilayer perceptrons. However, for many problems, the RBF network often involves a larger number of hidden units. This implies that the runtime (after training) speed of the RBF network is often slower than the runtime speed of a multilayer perceptron. The efficiencies (error versus network size) of the RBF network and the multilayer perceptron are, however, problem- dependent. It has been shown that the RBF network has the same asymptotic approximation power as a multilayer perceptron.

Issues

There are many issues in designing feed-forward networks, including

How many layers are needed for a given task,

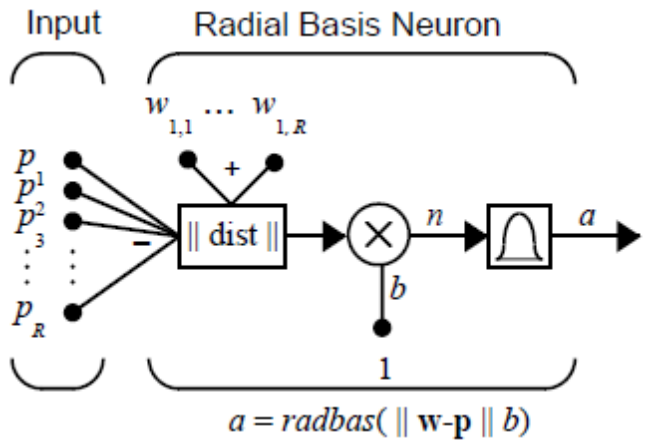
How many units are needed per layer,

How will the network perform on data not included in the training set (generalization ability), and

How large the training set should be for “good” generalization.

Although multilayer feed-forward networks using back propagation have been widely employed for classification and function approximation,2 many design parameters still must be determined

by trial and error. Existing theoretical results provide only very loose guidelines for selecting these parameters in practice.

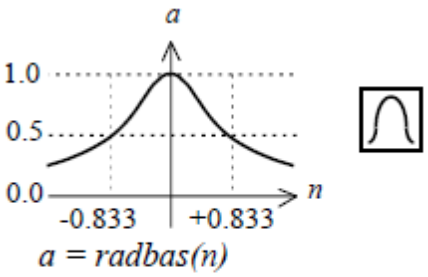


Notice that the expression for the net input of a radbas neuron is different from that of neurons in previous chapters. Here the net input to the radbas transfer function is the vector distance between its weight vector **w** and the input vector **p**, multiplied by the bias b. (The box in this figure accepts the input vector **p** and the single row input weight matrix, and produces the dot product of the two.)

The transfer function for a radial basis neuron is:

$$radbas(n) = e^{-n^2}$$

Here is a plot of the radbas transfer function.



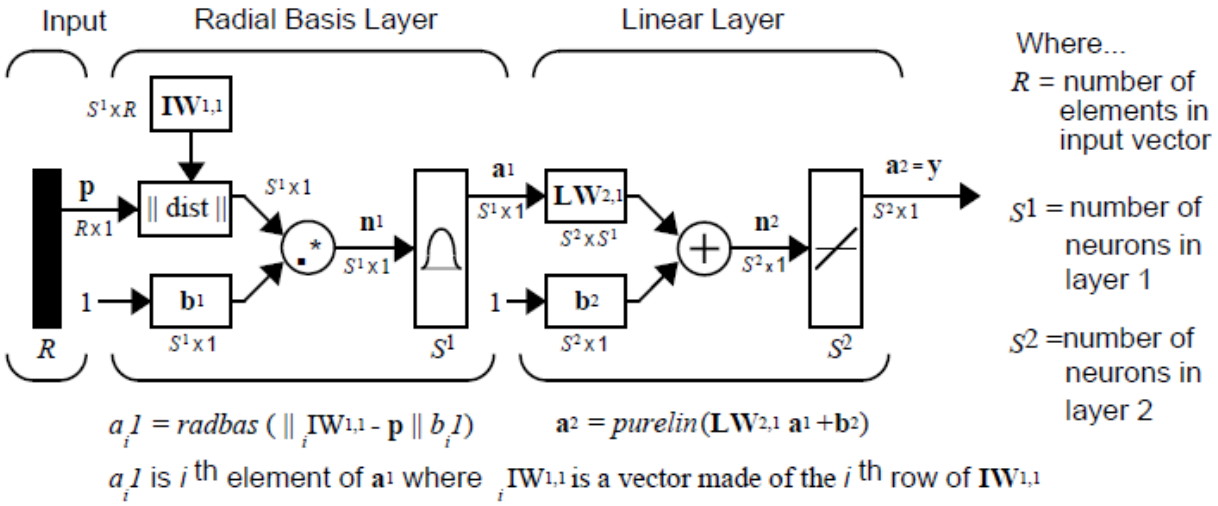
Radial Basis Function

The radial basis function has a maximum of 1 when its input is 0. As the distance between **w** and **p** decreases, the output increases. Thus, a radial basis neuron acts as a detector that produces 1 whenever the input **p** is identical to its weight vector **p**.

The bias b allows the sensitivity of the radbas neuron to be adjusted. For example, if a neuron had a bias of 0.1 it would output 0.5 for any input vector **p** at vector distance of 8.326 (0.8326/b) from its weight vector **w**.

Network Architecture

Radial basis networks consist of two layers: a hidden radial basis layer of S1 neurons, and an output linear layer of S2 neurons.



The box in this figure accepts the input vector **p** and the input weight matrix **IW**1,1, and produces a vector having S1 elements. The elements are the distances between the input vector and vectors i**IW**1,1 formed from the rows of the input weight matrix.

We can understand how this network behaves by following an input vector **p** through the network to the output **a**2. If we present an input vector to such a network, each neuron in the radial basis layer will output a value according to how close the input vector is to each neuron's weight vector.

Thus, radial basis neurons with weight vectors quite different from the input vector **p** have outputs near zero. These small outputs have only a negligible effect on the linear output neurons.

In contrast, a radial basis neuron with a weight vector close to the input vector **p** produces a value near 1. If a neuron has an output of 1 its output weights in the second layer pass their values to the linear neurons in the second layer.

In fact, if only one radial basis neuron had an output of 1, and all others had outputs of 0's (or very close to 0), the output of the linear layer would be the active neuron's output weights. This would, however, be an extreme case.

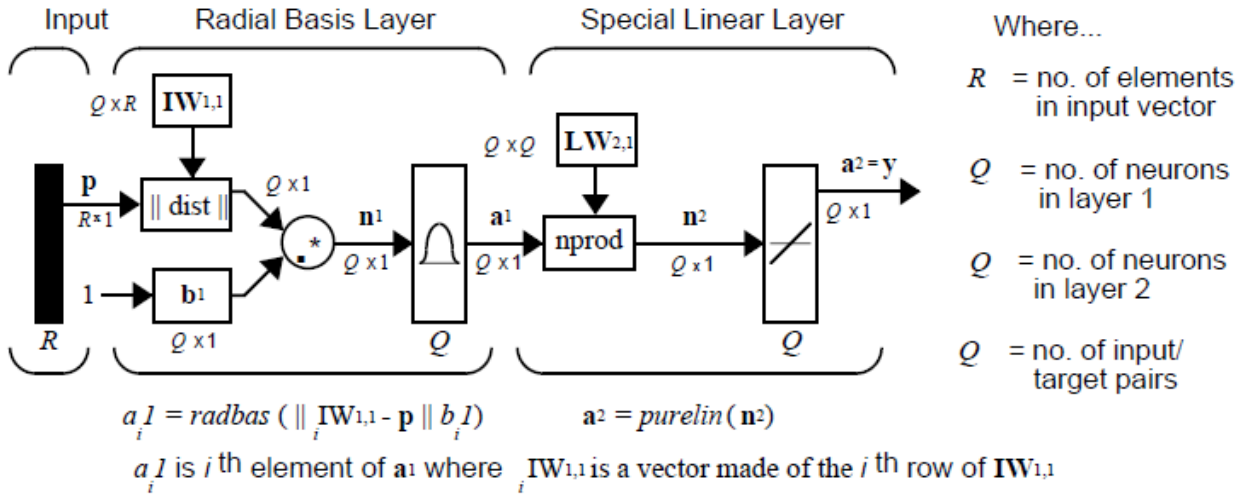
Typically several neurons are always firing, to varying degrees.

Generalized Regression Networks

A generalized regression neural network (GRNN) is often used for function approximation. As discussed below, it has a radial basis layer and a special linear layer.

Network Architecture

The architecture for the GRNN is shown below. It is similar to the radial basis network, but has a slightly different second layer.



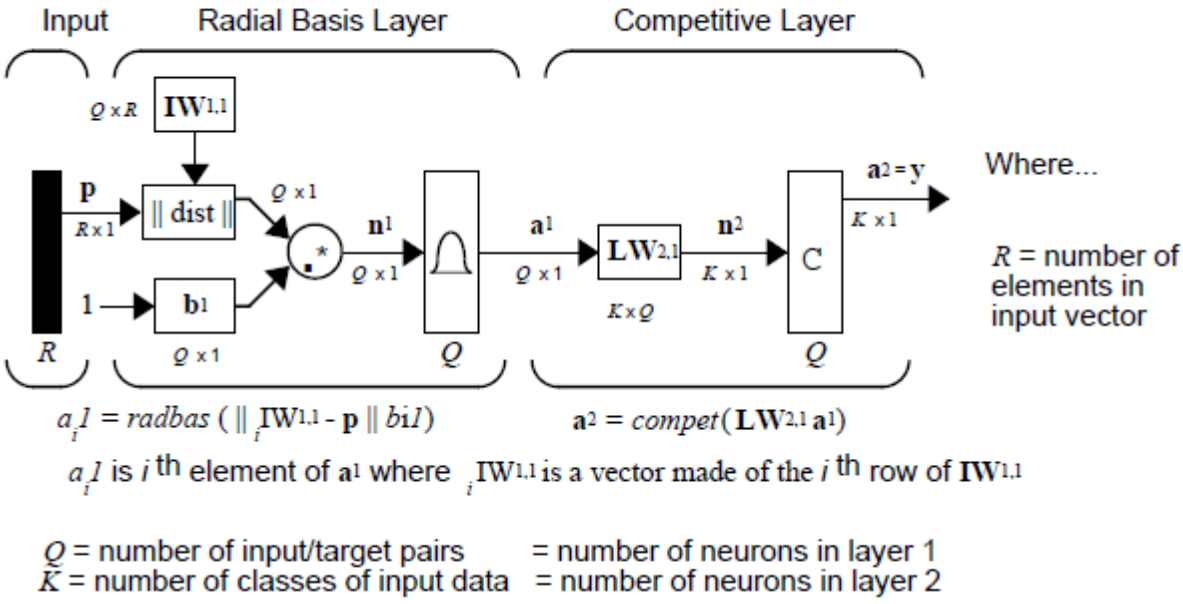
Here the **nprod** box shown above (code function normprod) produces S2 elements in vector **n**2. Each element is the dot product of a row of LW2,1 and the input vector **a**1, all normalized by the sum of the elements of **a**1.

Probabilistic Neural Networks

Probabilistic neural networks can be used for classification problems. When an input is presented, the first layer computes distances from the input vector to the training input vectors, and produces a vector whose elements indicate how close the input is to a training input. The second layer sums these contributions for each class of inputs to produce as its net output a vector of probabilities.

Finally, a compete transfer function on the output of the second layer picks the maximum of these probabilities, and produces a 1 for that class and a 0 for the other classes. The architecture for this system is shown below.

Network Architecture



It is assumed that there are Q input vector/target vector pairs. Each target vector has K elements. One of these elements is 1 and the rest is 0. Thus, each input vector is associated with one of K classes.

Self Organising Maps

Introduction

Self-organizing in networks is one of the most fascinating topics in the neural network field. Such networks can learn to detect regularities and correlations in their input and adapt their future responses to that input accordingly. The neurons of competitive networks learn to recognize groups of similar input vectors. Self-organizing maps learn to recognize groups of similar input vectors in such a way that neurons physically near each other in the neuron layer respond to similar input vectors.

Learning vector quantization (LVQ) is a method for training competitive layers in a supervised manner. A competitive layer automatically learns to classify input vectors. However, the classes that the competitive layer finds are dependent only on the distance between input vectors. If two input vectors are very similar, the competitive layer probably will put them in the same class. There is no mechanism in a strictly competitive layer design to say whether or not any two input vectors are in the same class or different classes.

COMPETITIVE W I N G RULES. Unlike Hebbian learning (in which multiple output units can be fired simultaneously), competitive-learning output units compete among themselves for activation. As a result, only one output unit is active at any given time. This phenomenon is known as winner-take-all. Competitive learning has been found to exist in biological neural network^.^ Competitive learning often clusters or categorizes the input data. Similar patterns are grouped by the network and represented by a single unit. This grouping is done automatically based on data correlations. The simplest competitive learning network consists of a single layer of output units as shown in Figure 4. Each output unit i in the network connects to all the input units (+) via weights, w,, j= **1,2,** . . . , n. Each output unit also connects to all other output units via inhibitory weights but has a self-feedback with an excitatory weight. As a result of competition, only the unit **i'** with the largest (or the smallest) net input becomes the winner, that is, **w,*x.** 2 w,. **x,** Vi, or 11 **w; - x** /I < /I w, - **x** 11, Vi. When all the weight vectors are normalized, these two inequalities are equivalent.. When all the weight vectors are normalized, these two inequalities are equivalent. **A** simple competitive learning rule can be stated as

$$\Delta w_{ij} = \begin{cases} \eta(x_j^u - w_{i^*j}), & i = i^*, \\ 0, & i \neq i^*. \end{cases} \qquad (1)$$

Note that only the weights of the winner unit get updated. The effect of this learning rule is to move the stored pattern in the winner unit (weights) a little bit closer to the input pattern. Figure 6 demonstrates a geometric interpretation of competitive learning. In this example, we assume that all input vectors have been normalized to have unit length. They are depicted as black dots in Figure 6. The weight vectors of the three units are randomly initialized. Their initial and final positions on the sphere after competitive learning are marked as Xs in Figures 6a and 6b, respectively. In Figure 6, each of the three natural groups (clusters) of patterns has been discovered by an output unit whose weight vector points to the center of gravity of the discovered group.

You can see from the competitive learning rule that the network will not stop learning (updating weights) unless the learning rate q is 0. A particular input pattern can fire different output units at different iterations during learning.

This brings up the stability issue of a learning system.

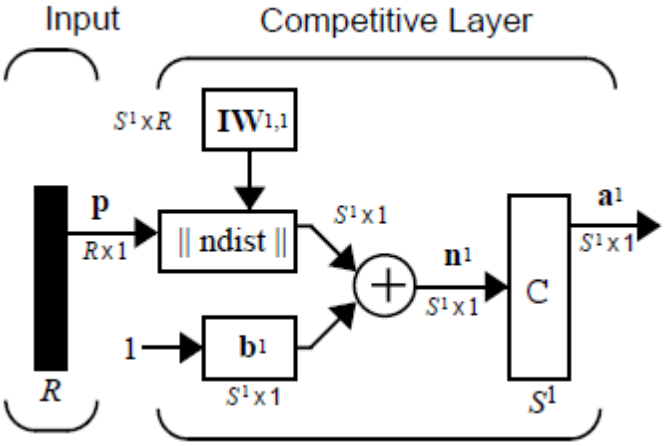
The system is said to be stable if no pattern in the training data changes its category after a finite number of learning iterations. One way to achieve stability is to force the learning rate to decrease gradually as the learning process proceeds towards 0. However, this artificial freezing of learning causes another problem termed plasticity, which is the ability to adapt to new data. This is known as Grossberg's stability- plasticity dilemma in competitive learning. The most well-known example of competitive learning is vector quantization for data compression. It has been widely used in speech and image processing for efficient storage, transmission, and modeling. Its goal is to represent a set or distribution of input vectors with a relatively small number of prototype vectors (weight vectors), or a codebook. Once a codebook has been constructed and agreed upon by both the transmitter and the receiver, you need only transmit or store the index of the corresponding prototype to the input vector. Given an input vector, its corresponding prototype can be found by searching for the nearest prototype in the codebook.

Competitive Learning

The neurons in a competitive layer distribute themselves to recognize frequently presented input vectors.

Architecture

The architecture for a competitive network is shown below.



The box in this figure accepts the input vector **p** and the input weight matrix **IW**1,1, and produces a vector having S1 elements. The elements are the negative of the distances between the input vector and vectors i**IW**1,1 formed from the rows of the input weight matrix.

The net input **n**1 of a competitive layer is computed by finding the negative distance between input vector **p** and the weight vectors and adding the biases

b. If all biases are zero, the maximum net input a neuron can have is 0. This occurs when the input vector **p** equals that neuron's weight vector.

The competitive transfer function accepts a net input vector for a layer and returns neuron outputs of 0 for all neurons except for the winner, the neuron associated with the most positive element of net input **n**1. The winner's output is 1. If all biases are 0, then the neuron whose weight vector is closest to the input vector has the least negative net input and, therefore, wins the competition to output a 1.

Reasons for using biases with competitive layers are introduced in a later section on training.

KOHONEN'S SELF-ORGANIZING MAPS

The self-organizing map (SOM)I6 has the desirable property of topology preservation, which captures an important aspect of the feature maps in the cortex of highly developed animal brains. In a topology-preserving mapping, nearby input patterns should activate nearby output units on the map. Figure 4 shows the basic network architecture of Kohonen's SOM. It basically consists of a two dimensional array of units, each connected to all n input nodes. Let **wg** denote the n-dimensional vector associated with the unit at location (i, j) of the 2D array. Each neuron computes the Euclidean distance between the input vector **x** and the stored weight vector wy. This SOM is a special type of competitive learning network that defines a spatial neighborhood for each output unit. The shape of the local neighborhood can be square, rectangular, or circular. Initial neighborhood size is often set to one half to two thirds of the network size and shrinks over time according to a schedule (for example, an exponentially decreasing function). During competitive learning, all the weight vectors associated with the winner and its neighboring units are updated (see the "SOM learning algorithm" sidebar). Kohonen's SOM can be used for projection of multivariate data, density approximation, and clustering. It has been successfully applied in the areas of speech recognition, image processing, robotics, and process control.2 The design parameters include the dimensionality of the neuron array, the number of neurons in each dimension, the shape of the neighborhood, the shrinking schedule of the neighborhood, and the learning rate.

Kohonen Learning Rule (learnk)

The weights of the winning neuron (a row of the input weight matrix) are adjusted with the Kohonen learning rule. Supposing that the ith neuron wins, the elements of the ith row of the input weight matrix are adjusted as shown
below.

$$ {}_i\mathbf{IW}^{1,1}(q) = {}_i\mathbf{IW}^{1,1}(q-1) + \alpha(\mathbf{p}(q) - {}_i\mathbf{IW}^{1,1}(q-1)) $$

The Kohonen rule allows the weights of a neuron to learn an input vector, and because of this it is useful in recognition applications.

Thus, the neuron whose weight vector was closest to the input vector is updated to be even closer. The result is that the winning neuron is more likely to win the competition the next time a similar vector is presented, and less likely to win when a very different input vector is presented. As more and more inputs are presented, each neuron in the layer closest to a group of input vectors soon adjusts its weight vector toward those input vectors. Eventually, if there are enough neurons, every cluster of similar input vectors will have a neuron that outputs 1 when a vector in the cluster is presented, while outputting a 0 at all other times. Thus, the competitive network learns to categorize the input vectors it sees.

Self-Organizing Maps

Self-organizing feature maps (SOFM) learn to classify input vectors according to how they are grouped in the input space. They differ from competitive layers in that neighboring neurons in the self-organizing map learn to recognize neighboring sections of the input space. Thus, self-organizing maps learn both the distribution (as do competitive layers) and topology of the input vectors they are trained on.

The neurons in the layer of an SOFM are arranged originally in physical positions according to a topology function. The functions gridtop, hextop or randtop can arrange the neurons in a grid, hexagonal, or random topology.

Distances between neurons are calculated from their positions with a distance function. There are four distance functions, dist, boxdist, linkdist and mandist. Link distance is the most common. These topology and distance functions are described in detail later in this section.

Here a self-organizing feature map network identifies a winning neuron using the same procedure as employed by a competitive layer. However, instead of updating only the winning

neuron, all neurons within a certain neighborhood of the winning neuron are updated using the Kohonen rule. Specifically, we adjust all such neurons as follows.

$$_i\mathbf{w}(q) = {}_i\mathbf{w}(q-1) + \alpha(\mathbf{p}(q) - {}_i\mathbf{w}(q-1)) \text{ or}$$

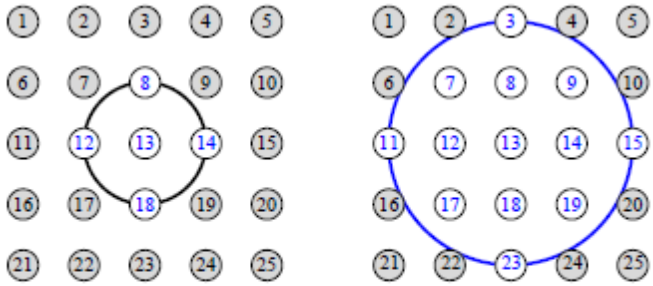
$$_i\mathbf{w}(q) = (1-\alpha){}_i\mathbf{w}(q-1) + \alpha\mathbf{p}(q)$$

Here the neighborhood contains the indices for all of the neurons that lie within a radius of the winning neuron.

$$N_i(d) = \{j, d_{ij} \le d\}$$

Thus, when a vector p is presented, the weights of the winning neuron and its close neighbors move toward p . Consequently, after many presentations, neighboring neurons will have learned vectors similar to each other.

To illustrate the concept of neighborhoods, consider the figure given below. The left diagram shows a two-dimensional neighborhood of radius d=1around neuron 13. The right diagram shows a neighborhood of radius =2.



These neighborhoods could be written as:

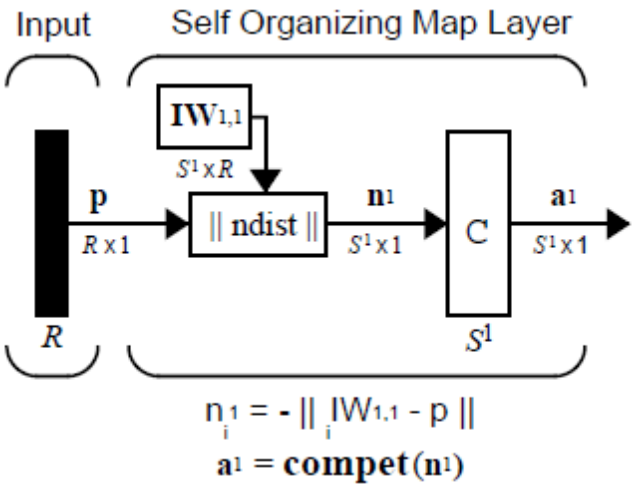
$$N_{13}(1) = \{8, 12, 13, 14, 18\} \text{ and}$$

$$N_{13}(2) = \{3, 7, 8, 9, 11, 12, 13, 14, 15, 17, 18, 19, 23\}$$

Note that the neurons in an SOFM do not have to be arranged in a two-dimensional pattern. You can use a one-dimensional arrangement, or even three or more dimensions. For a one-dimensional SOFM, a neuron has only two neighbors within a radius of 1 (or a single neighbor if the neuron is at the end of the line).You can also define distance in different ways, for instance, by using rectangular and hexagonal arrangements of neurons and neighborhoods. The performance of the network is not sensitive to the exact shape of the neighborhoods.

Architecture

The architecture for this SOFM is shown below.



This architecture is like that of a competitive network, except no bias is used here. The competitive transfer function produces a 1 for output element **a**1icorresponding to ,the winning neuron. All other output elements in **a**1 are 0.

Now, however, as described above, neurons close to the winning neuron are updated along with the winning neuron. As described previously, one can chose from various topologies of neurons. Similarly, one can choose from various distance expressions to calculate neurons that are close to the winning neuron.

Recurrent Network

Recurrent networks are a topic of considerable interest. This chapter covers two recurrent networks: Elman, and Hopfield networks.

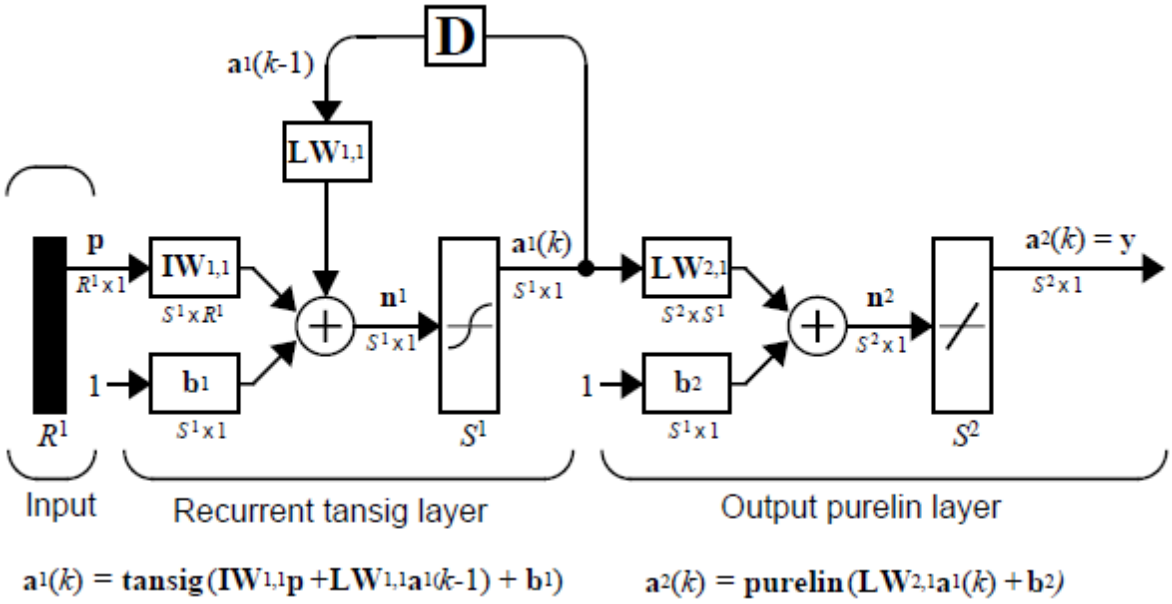
Elman networks are two-layer backpropagation networks, with the addition of a feedback connection from the output of the hidden layer to its input. This feedback path allows Elman networks to learn to recognize and generate temporal patterns, as well as spatial patterns.

The Hopfield network is used to store one or more stable target vectors. These stable vectors can be viewed as memories that the network recalls when provided with similar vectors that act as a cue to the network memory.

Elman Networks

Architecture

The Elman network commonly is a two-layer network with feedback from the first-layer output to the first layer input. This recurrent connection allows the Elman network to both detect and generate time-varying patterns. A two-layer Elman network is shown below.



$$\mathbf{a}^1(k) = \mathbf{tansig}(\mathbf{IW}^{1,1}\mathbf{p} + \mathbf{LW}^{1,1}\mathbf{a}^1(k\text{-}1) + \mathbf{b}^1) \qquad \mathbf{a}^2(k) = \mathbf{purelin}(\mathbf{LW}^{2,1}\mathbf{a}^1(k) + \mathbf{b}^2)$$

The Elman network has tansig neurons in its hidden (recurrent) layer, and purelin neurons in its output layer. This combination is special in that two-layer networks with these transfer functions can approximate any function (with a finite number of discontinuities) with arbitrary accuracy. The only requirement is that the hidden layer must have enough neurons. More hidden neurons are needed as the function being fit increases in complexity.

Note that the Elman network differs from conventional two-layer networks in that the first layer has a recurrent connection. The delay in this connection stores values from the previous time step, which can be used in the current time step.

Thus, even if two Elman networks, with the same weights and biases, are given identical inputs at a given time step, their outputs can be different due to different feedback states. Because the network can store information for future reference, it is able to learn temporal patterns as well as spatial patterns. The Elman network can be trained to respond to, and to generate, both kinds of patterns.

HOPFIELD NETWORK

Hopfield used a network energy function as a tool for designing recurrent networks and for understanding their dynamic behavior.' Hopfield's formulation made explicit the principle of storing information as dynamically stable attractors and popularized the use of recurrent networks for associative memory and for solving combinatorial optimization problems.

A Hopfield network with n units has two versions: binary and continuously valued. Let **v,** be the state or output of the ith unit. For binary networks, v, is either + 1 or -1, but for continuous networks, v, can be any value between 0 and 1. Let w, be the synapse weight on the connection from units i to j. In Hopfield networks, w, = w,~, Vi, j (symmetric networks), and w,, = 0, Vi (no self-feedback connections). The network dynamics for the binary Hopfield network are

$$v_i = Sgn\left(\sum_j w_{ij} v_j - \theta_i\right)$$

The dynamic update of network states in Equation 4 can be carried out in at least two ways: synchronously and synchronously. In a synchronous updating scheme, all units are updated simultaneously at each time step. A central: lock must synchronize the process. An asynchronous updating scheme selects one unit at a time and updates its state. The unit for updating can be randomly chosen. The energy function of the binary Hopfield network in a state **v** = (vl, v2, . . . , vJT is given by

$$E = -\frac{1}{2}\sum_i \sum_j w_{ij} v_i v_j$$

The central property of the energy function is that as network state evolves according to the network dynamics (Equation 4), the network energy always decreases and eventually reaches a local minimum point (attractor) where the network stays with a constant energy.

Associative memory

When a set of patterns is stored in these network amactors, it can be used as an associative memory. Any pattern present in the basin of attraction of a stored pattern can be used as an index to retrieve it. An associative memory usually operates in two phases: storage and retrieval. In the storage phase, the weights in the network are determined so that the attractors of the network

memorize a set ofp n-dimensional patterns **{XI, x2,** . . . , **xp}** to be stored. A generalization of the Hebbian learning rule can be used for setting connection weights wli. In the retrieval phase, the input pattern is used as the initial state of the network, and the network evolves according to its dynamics. A pattern is produced (or retrieved) when the network reaches equilibrium. How many patterns can be stored in a network with n binary units? In other words, what is the memory capacity of a network? It is finite because a network with n binary units has a maximum of 2“ distinct states, and not all of them are attractors. Moreover, not all amactors (stable states) can store useful patterns. Spurious attractors can also store patterns different from those in the training set.2

It has been shown that the maximum number of random patterns that a Hopfield network can store is P,, = 0.15W~h en the number of stored patternsp < 0.15n, a nearly perfect recall can be achieved. When memorypatterns are orthogonal vectors instead of random patterns, more patterns can be stored. But the number of spurious attractors increases asp reaches capacity. Several learning rules have been proposed for increasing the memory capacity of Hopfield networks.2 Note that we require n2 connections in the network to storep n-bit patterns.

Energy minimization

Hopfield networks always evolve in the direction that leads to lower network energy. This implies that if a combinatorial optimization problem can be formulated as minimizing this energy, the Hopfield network can be used to find the optimal (or suboptimal) solution by letting the network evolve freely. In fact, any quadratic objective function can be rewritten in the form of Hopfield network energy. For example, the classic Traveling Salesman Problem can be formulated as such a problem.

Hopfield Network

Fundamentals

The goal here is to design a network that stores a specific set of equilibrium points such that, when an initial condition is provided, the network eventually comes to rest at such a design point. The network is recursive in that the output is fed back as the input, once the network is in operation. Hopefully, the network output will settle on one of the original design points

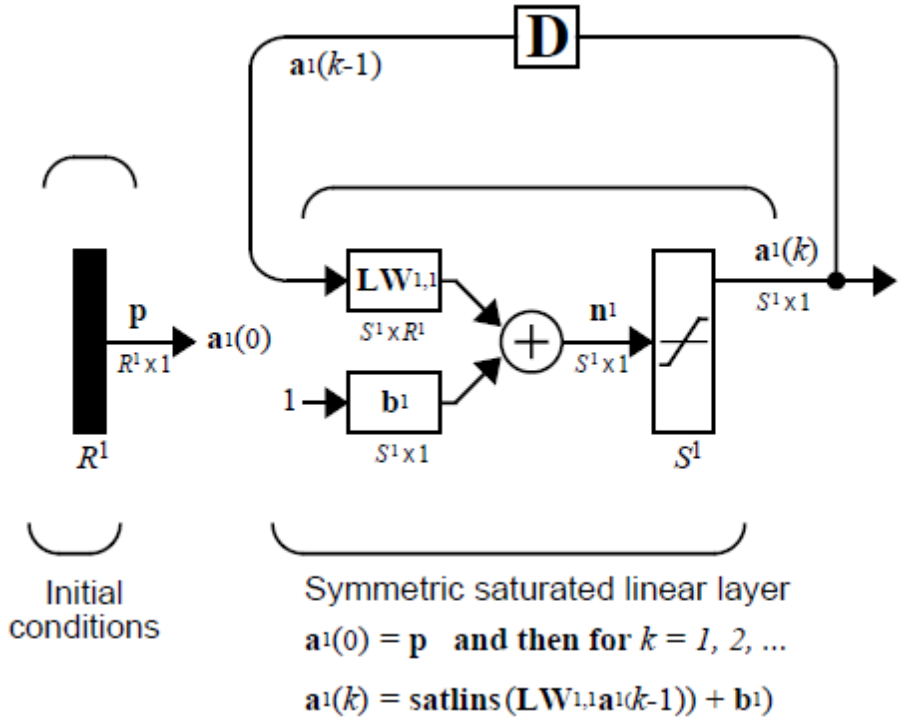
The design method that we present is not perfect in that the designed network may have undesired spurious equilibrium points in addition to the desired ones. However, the number of these undesired points is made as small as possible by the design method. Further, the domain of attraction of the designed equilibrium points is as large as possible.

The design method is based on a system of first-order linear ordinary differential equations that are defined on a closed hypercube of the state space.

The solutions exist on the boundary of the hypercube. These systems have the basic structure of the Hopfield model, but are easier to understand and design than the Hopfield model.

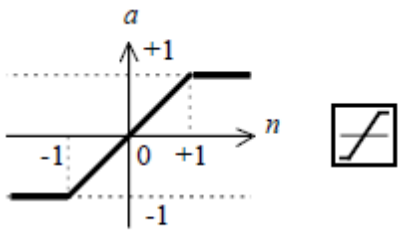
Architecture

The architecture of the network that we are using follows.



As noted, the input **p** to this network merely supplies the initial conditions.

The Hopfield network uses the saturated linear transfer function satlins.



$a = satlins(n)$

Satlins Transfer Function

For inputs less than -1 satlins produces -1. For inputs in the range -1 to +1 it simply returns the input value. For inputs greater than +1 it produces +1.

This network can be tested with one or more input vectors which are presented as initial conditions to the network. After the initial conditions are given, the network produces an output which is then fed back to become the input. This process is repeated over and over until the output stabilizes. Hopefully, each output vector eventually converges to one of the design equilibrium point vectors that is closest to the input that provoked it.

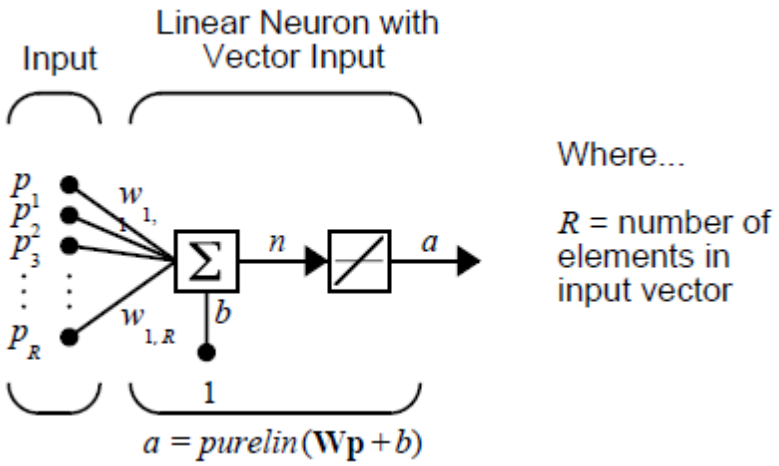
Introduction

The ADALINE (Adaptive Linear Neuron networks) networks discussed in this chapter are similar to the perceptron, but their transfer function is linear rather than hard-limiting. This allows their outputs to take on any value, whereas the perceptron output is limited to either 0 or 1. Both the ADALINE and the perceptron can only solve linearly separable problems. However, here we will make use of the LMS (Least Mean Squares) learning rule, which is much more powerful than the perceptron learning rule. The LMS or Widrow-Hoff learning rule minimizes the mean square error and, thus, moves the decision boundaries as far as it can from the training patterns.

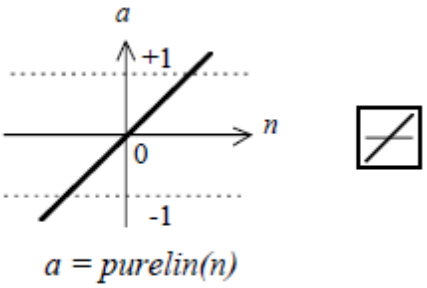
In this chapter, we design an adaptive linear system that responds to changes in its environment as it is operating. Linear networks that are adjusted at each time step based on new input and target vectors can find weights and biases that minimize the network's sum-squared error for recent input and target vectors. Networks of this sort are often used in error cancellation, signal processing, and control systems.

Linear Neuron Model

A linear neuron with R inputs is shown below.



This network has the same basic structure as the perceptron. The only difference is that the linear neuron uses a linear transfer function, which we name purelin.



Linear Transfer Function

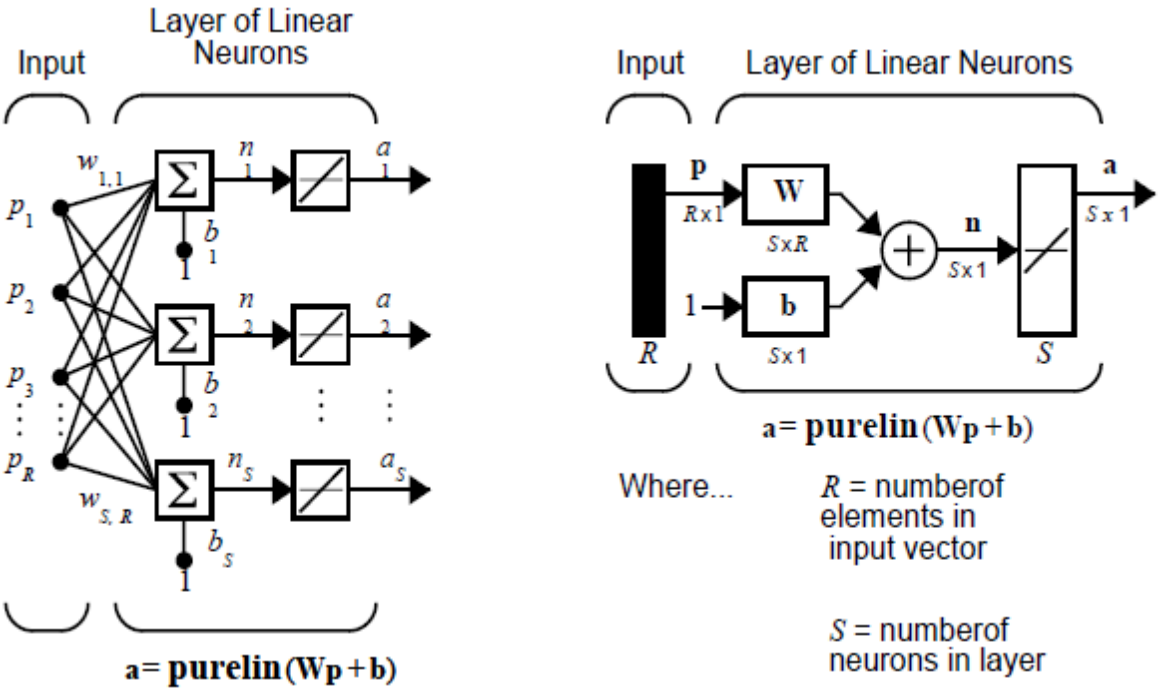
The linear transfer function calculates the neuron's output by simply returning the value passed to it.

$$a = purelin(n) = purelin(\mathbf{W}\mathbf{p} + b) = \mathbf{W}\mathbf{p} + b$$

This neuron can be trained to learn an affine function of its inputs, or to find a linear approximation to a nonlinear function. A linear network cannot, of course, be made to perform a nonlinear computation.

Adaptive Linear Network Architecture

The ADALINE network shown below has one layer of S neurons connected to R inputs through a matrix of weights **W**.

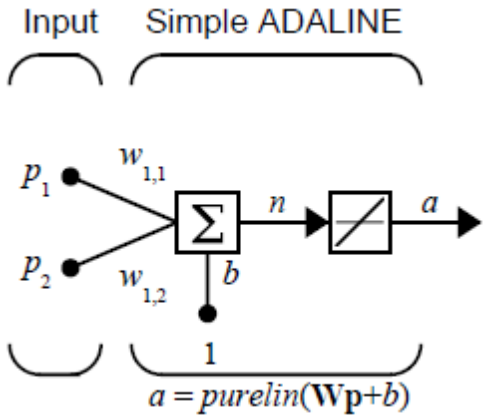


This network is sometimes called a MADALINE for Many ADALINES. Note that the figure on the right defines an S-length output vector **a**.

The Widrow-Hoff rule can only train single-layer linear networks. This is not much of a disadvantage, however, as single-layer linear networks are just as capable as multilayer linear networks. For every multilayer linear network, there is an equivalent single-layer linear network.

Single ADALINE (newlin)

Consider a single ADALINE with two inputs. The diagram for this network is shown below.

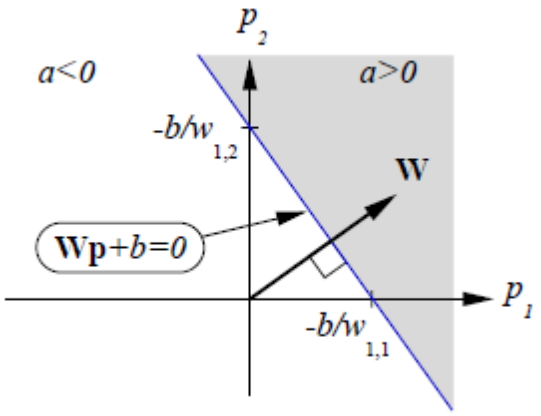


The weight matrix **W** in this case has only one row. The network output is:

$$a = purelin(n) = purelin(\mathbf{W}\mathbf{p} + b) = \mathbf{W}\mathbf{p} + b \qquad \text{or}$$

$$a = w_{1,1}p_1 + w_{1,2}p_2 + b$$

Like the perceptron, the ADALINE has a decision boundary that is determined by the input vectors for which the net input n is zero. For the equation specifies such a decision boundary as shown below.



Input vectors in the upper right gray area lead to an output greater than 0.

Input vectors in the lower left white area lead to an output less than 0. Thus, the ADALINE can be used to classify objects into two categories.

FUSION OF FUZZY SYSTEM AND NEURAL NETWORKS

In this chapter, we study the fusion of fuzzy systems and neural networks.

The two methods are complementary. The neural networks can learn from data while the fuzzy systems cannot; the fuzzy systems are easy to comprehend because they use linguistic terms but the neural networks are not. Many researches have been devoted to the fusion of them in order to take their advantages.

11.1 Neural Networks

11.1.1 Basic Concepts of Neural Networks

Neural networks(NN) are a computational model of the operation of human brain. A neural network is composed of a number of nodes connected by links. Each link has a numeric weight associated with it. Weights are the primary means of long-term storage in neural networks.

One of the major features of the neural network is its learning capability. They can adjust the weights to improve the performance for a given task. Learning usually takes place by updating the weights.

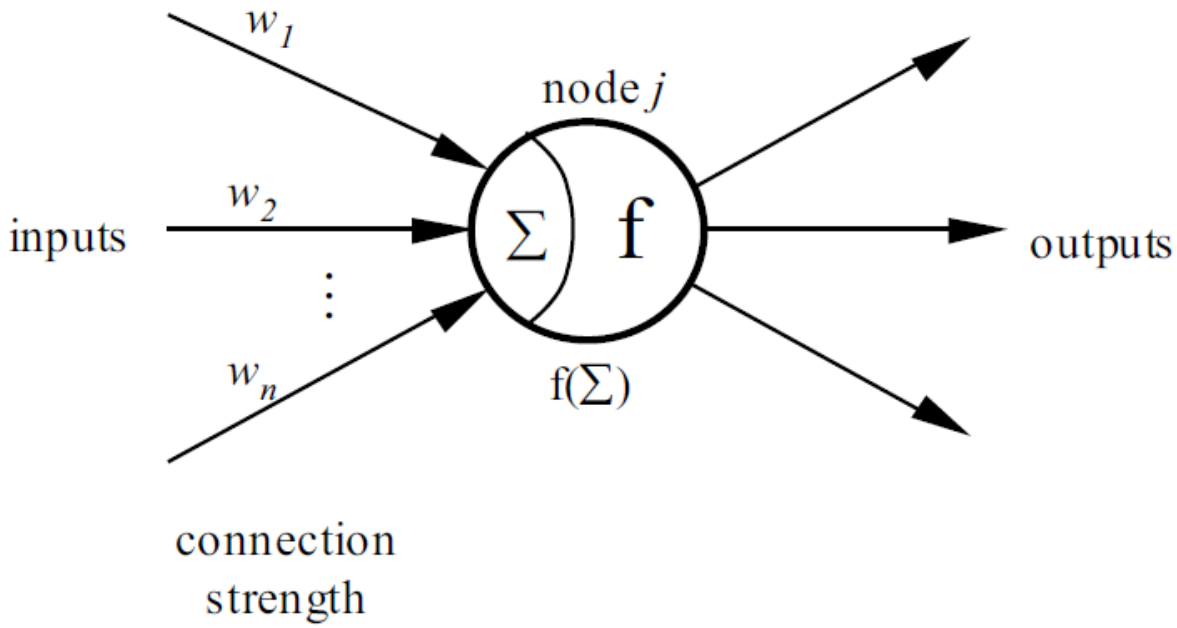
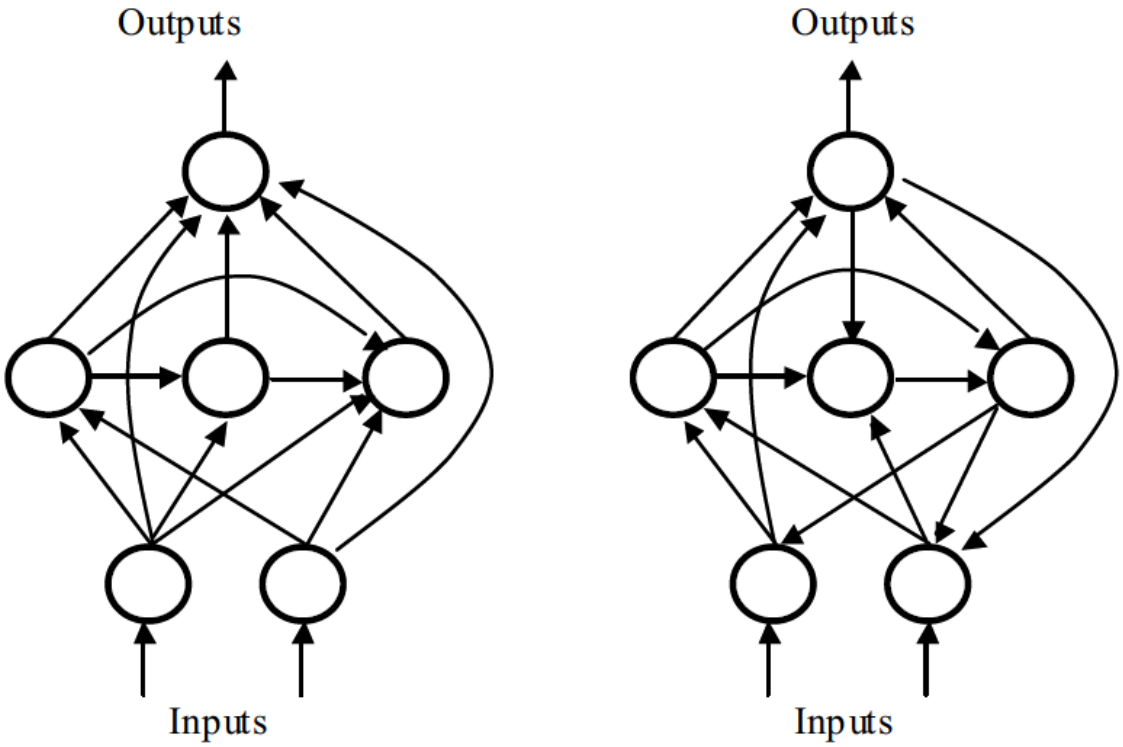


Fig. 11.1. Node of neural networks



(a) a feedforward neural network (b) a feedback neural network

Fig. 11.2. Neural Networks

A node has a set of input links from other nodes, a set of output links, and a nonlinear function called activation function. The output of a node is generated from the inputs, the weight of links, and activation function (Fig 11.1.). So, the output of node *j* are as follows:

$$o_j = f(net_j)$$

where

$$net_j = \sum_i w_{ij} o_i$$

is the output of node i and an input to node *j*, *wji* is the weight of the link between nodes *j* and *i*, and *f* is the activation function. Any nonlinear functions can be used as the activation function, but sigmoidal function is often used.

11.1.2 Learning Algorithms

Neural networks can be categorized into feedforward and feedback. The feedforward neural networks have only feedforward links, i.e. neural networks which do not have feedback cycle. The output of a node will not directly or indirectly be used as an input of that node. To the contrary, the feedback neural networks have feedback cycle. The output of a node may be used as an input of a node. (Figs 11.2(a), (b)) show the feedforward and feedback neural networks, respectively. In the case of the feedback neural network, there is no guarantee that the networks become stable because of the feedback cycle. Some of them converge to a stable point, some may have limit-cycle, or become chaotic or divergent. These are common characteristics of non-linear systems which have feedback.

There are two types of learning algorithms in the neural networks. The first type is supervised learning. It uses a set of training data which consist of pairs of input and output. During learning, the weights of a neural network are changed so that the input-output mapping becomes more and more close to the training data. The second type is unsupervised learning. While the supervised learning presents target answers for each input to a neural network, the unsupervised learning has the target answers. It adjusts the weights of a neural network in response to only input patterns without the target answers. In the unsupervised learning, the network usually classifies the input patterns into similarity categories.

11.1.3 Multilayer Perceptrons and Error Backpropagation Learning

Among the neural networks and learning algorithms, multiplayer perceptron network and its learning algorithm are widely used. This learning algorithm is called error backpropagation method. Multilayer perceptrons are layered feedforward neural networks. They consist of several layers. A layer is a set of nodes which do not have inter-connection links, i.e. the nodes in the same layer are not connected to each other. One more characteristic is that the nodes in a layer are connected to only the nodes in the neighboring layer. (Fig 11.3) shows a three layer perceptron network. The first layer is the input layer, the second is the hidden layer and the third is the output layer.

11.2 Fusion with Neural Networks

Neural networks and fuzzy systems are two complementary technologies.

Neural networks can learn from data, but the knowledge represented by the neural networks is difficult to understand. In contrast, fuzzy systems are easy to comprehend because they use linguistic terms and if-then rules, but it does not have learning algorithms. So, many researches have been devoted to fusion of them. The researches on fusion of neural networks and fuzzy systems can be classified into four categories:

(1) Modifying fuzzy systems with supervised neural network learning,

(2) Building neural networks using fuzzy systems,

(3) Making membership functions with neural networks,

(4) Concatenating neural networks and fuzzy systems.

11.2.1 Modifying Fuzzy Systems with Supervised Neural Network Learning

Research in this category represents fuzzy systems with neural networks.

These systems are called neuro fuzzy systems, and the neural networks are used to improve the performance of fuzzy systems. Neuro fuzzy systems have characteristics of both neural networks and fuzzy systems; they have learning capability like neural networks and can perform inference like fuzzy systems.

In the ordinary neural networks, nodes have the same functionality and are fully connected to the nodes in the neighboring layers. But in a neuro fuzzy system, nodes have different functionalities and are not fully connected to the nodes in the neighboring layers. These differences come from the fact that the nodes and links in a neuro fuzzy system usually correspond to a specific component in a fuzzy system. That is, some nodes represent the linguistic terms of input

variables, some nodes are for those of output variables, and some nodes and links are used for representing fuzzy rules.

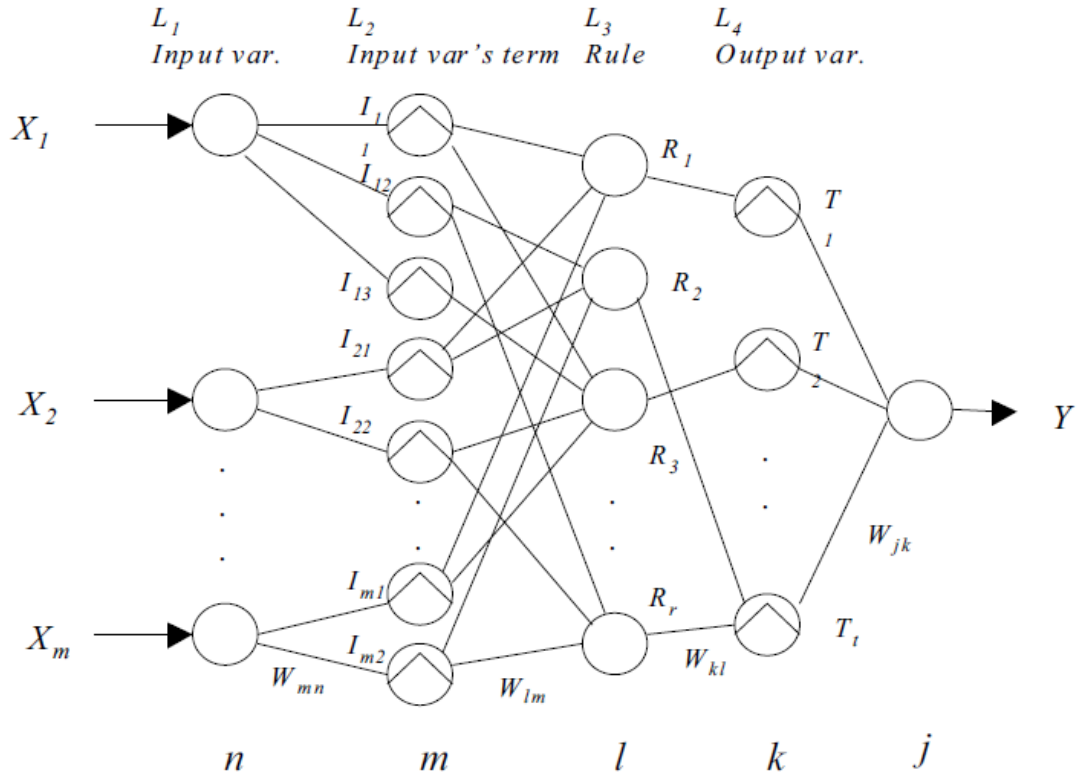


Fig. 11.4. Structure of the neuro fuzzy system proposed by Kwak

One merit of the neuro fuzzy systems over the ordinary neural networks is the easiness of adding the expert knowledge before learning. The neuro fuzzy systems are usually built from the given fuzzy systems which are based on the expert or prior knowledge. Thus, the neuro fuzzy systems can embed the knowledge at the beginning. For this reason, convergence to local minimum may not be so much serious as that in the ordinary neural networks.

For example, the neuro fuzzy system proposed by Kwak, Lee and Lee- Kwang consists of five layers as shown in (Fig 11.4.). In the followings, the function of a node *f* is presented.

(1) First layer of the network(Inputs). The nodes in the first layer take inputs and just pass them to the second layer.

$$f_j^1(x) = x$$

(2) Second layer of the network(Input linguistic terms). A node in this layer represents a linguistic term of an input variable. It has parameters which represent the membership function of linguistic term. For example, in (Fig 11.4.). the input variable X1 is connected to three nodes in the second layer and X2 is connected to two nodes. It means that there are three linguistic terms defined on X1 and two linguistic terms on X2. The node in the second layer outputs the members degree of input.

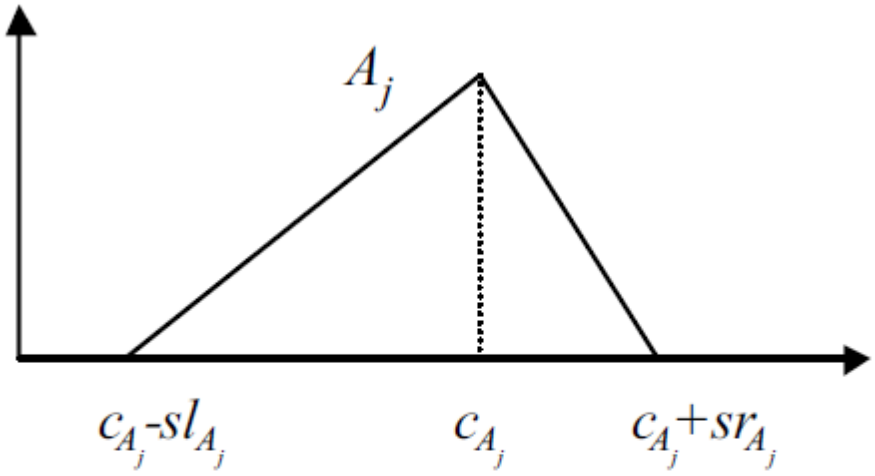


Fig. 11.5. A triangular fuzzy number

(3) Third layer of the network(Antecedent parts). This layer corresponds to the antecedent parts of fuzzy rules. For example, in Fig. 11. the inputs of *R*1 are the outputs of *I*11, *I*21 and *Im*1. It represents the antecedent part of the rule such as :

if is and is and and is then "

The output of the node is the matching degree of given inputs to the antecedent part. When evaluating the matching degrees, the minimum or product operators can be used. The connection weights between the second and third layers are fixed to 1.0.

$$f_i^3(x_1, x_2, \cdots, x_p) = \begin{cases} \min_{i=1}^{p}(x_i) & \text{if minimum used} \\ \prod_{i=1}^{p}(x_i) & \text{if product used} \end{cases}$$

(4) Fourth layer of the network(Consequent parts). This layer represents the consequent parts of fuzzy rules. Like in the second layer, a node in this layer represents a linguistic term of output variable. For example, node *Tt* has two inputs from *R*2 and *Rr*.

The output of the node is the maximum matching degree of an input to the rules which are represented by the node. For example, the output of the node *Tt* is the maximum output of nodes *R*2 and *Rr*. The weights between the third and fourth layers are used as the importance degree of rules, or fixed to 1.00.

$$f_j^4(x_1, x_2, \cdots, x_q) = \max_{i=1}^{q} \{w_{ji} x_i\}$$

where *wji* is the weight between node *j* in the fourth layer and node *i* in the third.

(5) Fifth layer of the network(Defuzzification). A node in this layer gathers the outputs of all rules and defuzzifies them. A defuzzification method similar to the center of gravity method is used.

The weight of links between it and the fourth layer is 1.00.

$$f_j^5(x_1, x_2, \cdots, x_t) = \frac{\sum_i^t Centroid(B_i, x_i) Area(B_i, x_i)}{\sum_i^t Area(B_i, x_i)}$$

where Bi is the fuzzy set represented by the node j in the fourth layer and xi is the output of the node. If the output variable y is quantized with level n, Centroid(Bi, xi) and Area(Bi, xi) are defined as follows:

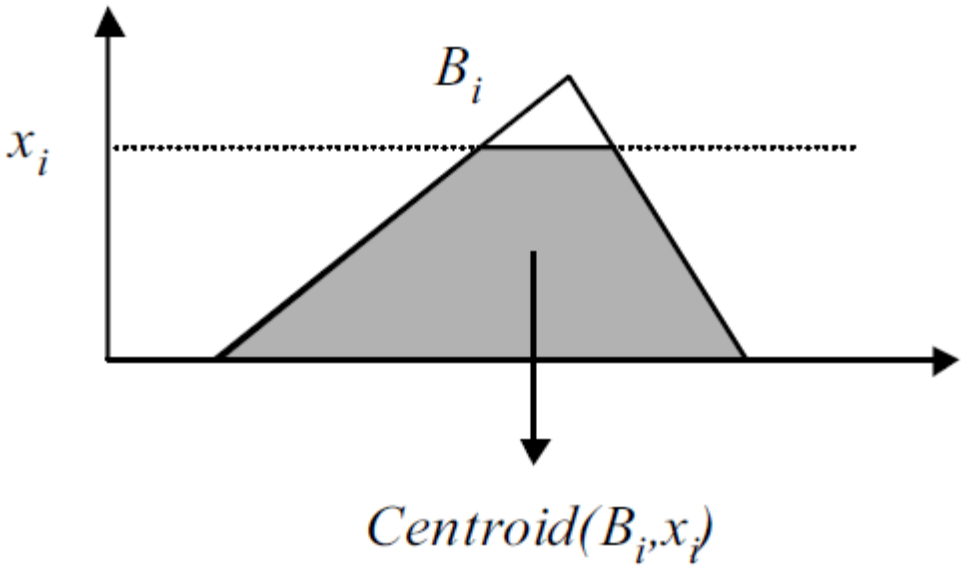


Fig. 11.6. Definition of $Centroid(B_i, x_i)$

The difference is the how many times the overlapped areas are evaluated. The center of gravity method evaluates those areas only one time, but the Kwak's method may evaluate the overlapped area several times. One merit of the Kwak's method is that the defuzzification may be done with simpler operations than the center of gravity.

(6) Learning algorithm. The learning algorithm of this model is based on the error backpropagation. During the learning process, the weights between the third and fourth layers, and the parameters representing membership functions in the nodes of the second and fourth layers are modified based on the error backpropagation method.

11.2.2 Building Neural Networks using Fuzzy Systems

In the neuro fuzzy systems, neural networks were used to improve the performance of fuzzy systems. But the methods in this category use fuzzy system or fuzzy-rule structure to design neural networks. This model is a kind of divide and conquer approaches. Instead of training a neural network for the whole given input-output data, this model builds several networks:

(1) Builds a fuzzy classifier which clusters the given input-output data into several classes,

(2) Builds a neural network per class,

(3) Trains the neural networks with the input-output data in the corresponding class.

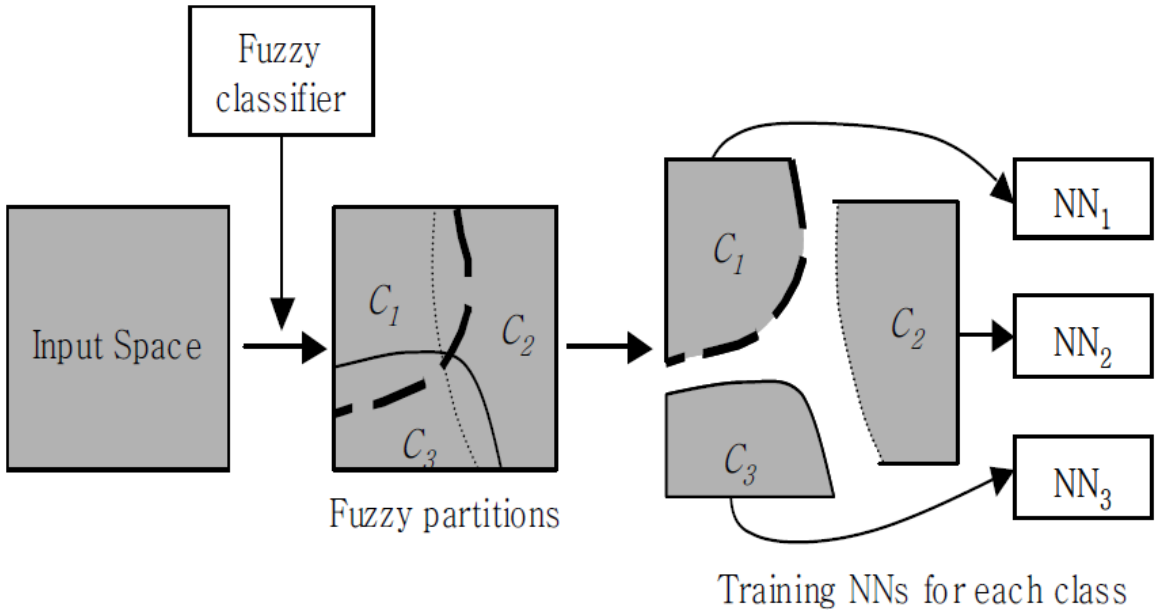


Fig. 11.9. Building neural networks using fuzzy systems

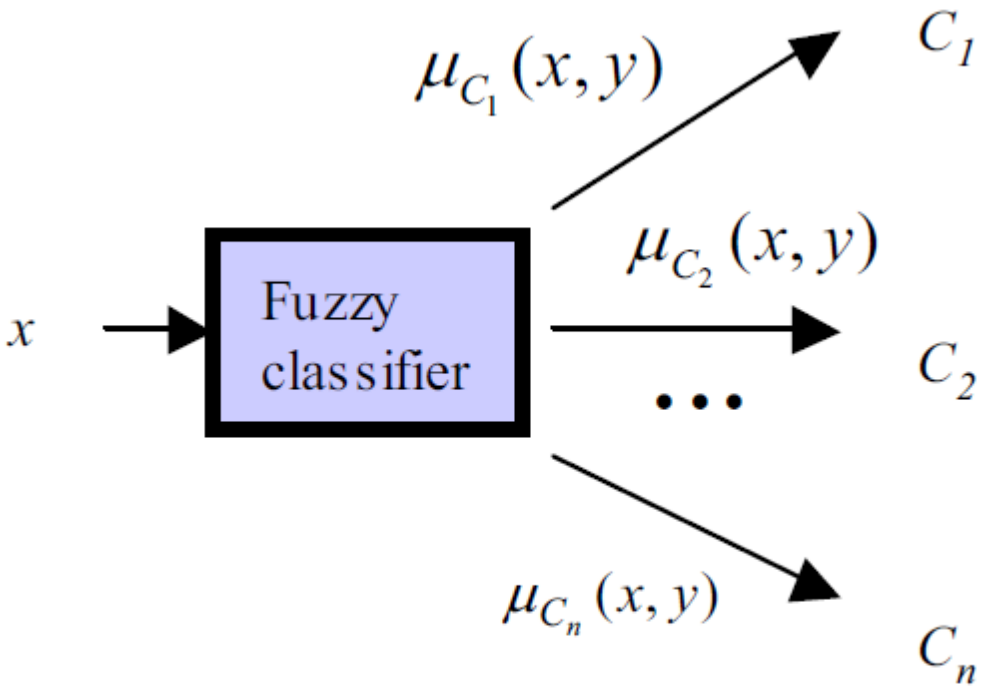


Fig. 11.9. A fuzzy classifier

(Fig.11.8.) shows the schematic diagram of this model. A fuzzy classifier divides the given data into several classes whose boundary is fuzzy. For building fuzzy classifier, many kinds of methods can be used such as fuzzy c-means, neural networks, fuzzy rules, and so on. (Fig 11.9.) shows a fuzzy classifier of which outputs are the membership degrees of input to the classes. That is, a class can correspond to a rule. The membership degree of x to class Ci is calculated by the fuzzy classifier instead of membership functions.

$Ci(x)$ is the membership degree of x to class Ci, which is evaluated by a fuzzy classifier, and $NNi(x)$ is the output of neural network NNi for input x. The complexity in each fuzzily partitioned input space is much less than that of whole given task. This approach is suitable to the problems which can be divided into small sub-problems. For example, developing recognizer of characters of multiple fonts can be achieved by developing several recognizers of each font. If there is a font classifier, we can classify characters into each font group and then recognize the characters in each font group with the recognizer specialized to the corresponding font.

With this method, we can use priori knowledge of the task to reduce the complexity and increase the performance of a system. The fuzzy rules describe the priori knowledge of the given task, which can be obtained by analyzing training data. An example of this model was proposed by Takagi and Hayashi. They used a neural network to build a fuzzy classifier, and proposed a method which could reduce the number of rules or redundant input variables. Their method is briefly summarized:

(1) Data preparation

The input/output data (i , i) **x** y are divided into the training set and the

ts n , respectively. The training set is used for building a neural network and the testing set is used for reducing the number of input variables.

(2) Crisp partition of input space

The data in the training set are grouped into r classes, by a crisp clustering method. In the training set, data belonging to Cs .

(3) Development of fuzzy classifier

This step builds a fuzzy classifier by using a neural network. The classifier will be denoted by *mem NN* because it is a neural network which evaluates the membership degree of an input to each class.

(4) Development of neural networks

(5) Reduction of input variables

This step is for reducing the number of used input variables. Then, among the *m* input variables, an input variable *xp* is arbitrarily eliminated, and *s NN* is trained by the data without *xp* in the training set as in the step 4.

(Fig 11.10.) shows the fuzzy-rule-structured neural network proposed by Takagi and Hayashi. The extended method was proposed in Takagi, Suzuki, Kouda and Kojima.

11.2.3 Making Membership Functions with Neural Networks

This approach is very similar to the previous one. In the previous approach, fuzzy-rule structure was used to build neural network. But in this approach, a neural network is used to generate compact fuzzy rules.

(1) Fuzzy partition of data

An important element of fuzzy systems is the fuzzy partition of input space. So, if there are *k* inputs, the fuzzy rules generate *k*-dimensional fuzzy hypercubes in the input space. Even though we can easily understand these fuzzy hypercubes, it is not easy to get a flexible partition for nonlinear hypersurfaces. Since neural networks are proper to approximate nonlinear functions, they can be applied to construct a fuzzy partition.

The idea of this model is to build a neural network whose outputs are degrees that an input belongs to each class. These degrees can be considered as the membership degrees to each class. That is, the neural network takes the role of membership functions. Thus the membership functions of this model can be non-linear and multi-dimensional unlike the conventional fuzzy systems.

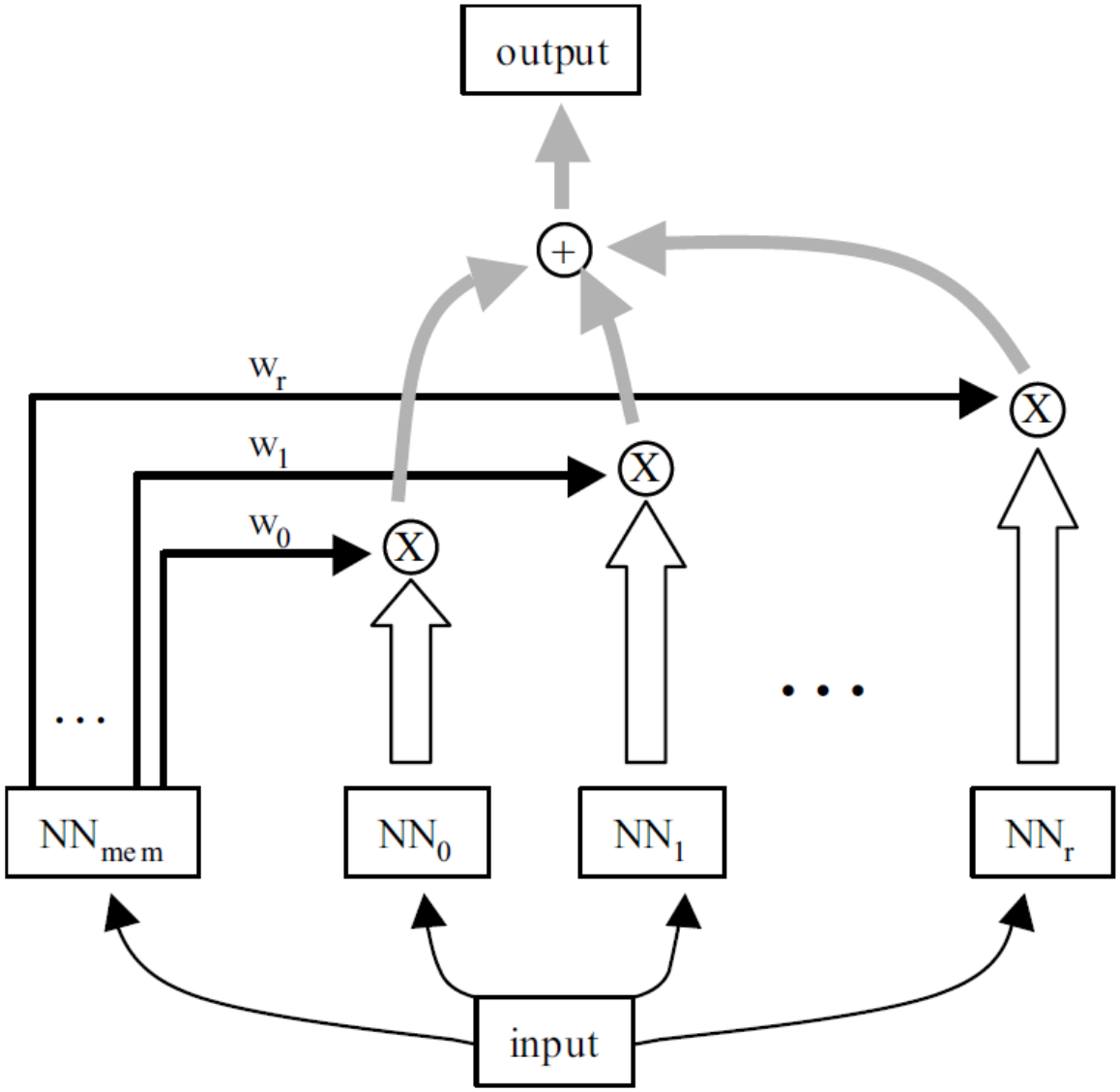


Fig. 11.10. Neural networks built in a fuzzy-rule structure

(2) Partition in a hyperspace

For example, we have two input variables *x* and *y*, and we know that the input space can be partitioned into three fuzzy classes: *C*1, *C*2 and *C*3 as shown in Fig. 11.11. The gray boundary means that it is not clear so the points on the boundary could belong to all classes adjacent to the boundary.

Let us suppose that we can easily make fuzzy rules as long as we can partition the input space into *C*1, *C*2 and *C*3. In this case, if we use fuzzy hypercubes to partition the input space into the

three classes, we need lots of fuzzy hypercubes and thus the number of fuzzy rules may increase. So, we need other method for partitioning. The neural network is a good candidate because they are proper to approximate nonlinear functions.

That is, we need only to construct a neural network that takes input variables of a fuzzy system as input and generates the degrees the input data belong to fuzzy regions. As shown in (Fig 11.11.) the fuzzy regions are fuzzy hypersurfaces. Due to the flexibility of these fuzzy hypersurfaces, the number of fuzzy rules in a fuzzy model can be reduced.

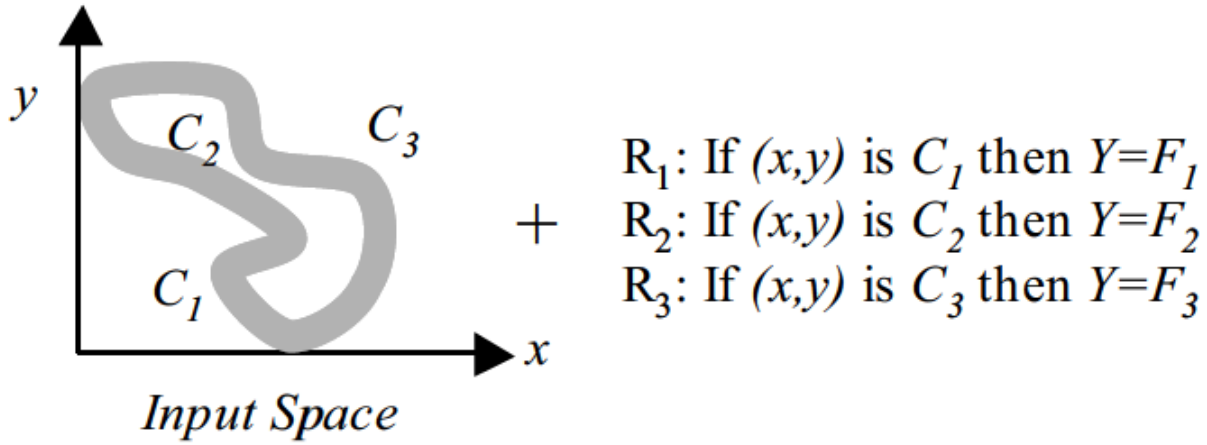


Fig 11.11. Fuzzy rules on fuzzy hyperspaces

(3) Example of the rolling mill control

This model has been used in the control of rolling mill. The purpose of the rolling mill is to make plates of iron, stainless, or aluminum by controlling 20 rolls. The controller will change system parameters if the surface of plate is not flat. How to change the parameters is determined by the produced surface shape of plates. To do this, first fuzzy rules are generated for only 20 standard template surface shapes. Then, a neural network is constructed which generates the similarity degrees to which an arbitrary surface shape belongs to each standard template shape.

The similarity degrees produced by the neural network are used as the matching degrees to the antecedent part of each rule. Since the antecedent parts of fuzzy control rules are standard template surface patterns, the output of the neural network corresponds to how much input surface pattern matches to each fuzzy rule. (Fig 11.12.) shows the structure of the system.

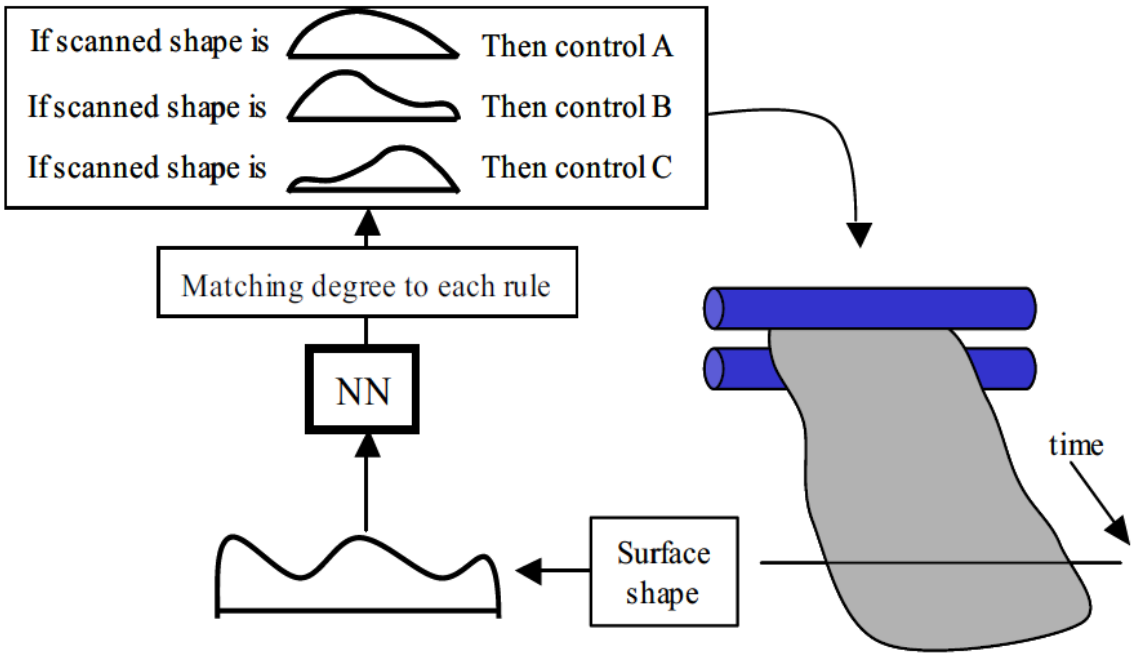


Fig. 11.12. Fuzzy rules with fuzzy hyperspace

11.2.4 Evaluation of Fuzzy Systems with Neural Networks

When a fuzzy system is developed, we have to evaluate the system in order to tune it or know its performance. When it is not easy to apply the developed system to the real system, neural networks can be used as a simulator of the real system.

Jung, Im, and Lee-Kwang developed a fuzzy control algorithm for shape control in cold rolling of a steel work. They modeled strip shapes and constructed a fuzzy rule base producing control actions for each irregular strip shape.

In order to evaluate the fuzzy rule base system before implement the system in the real field, they developed an emulator simulating the real cold rolling machine by using a neural network(Fig 11.14.). By using the neural network, they tuned the developed fuzzy system and then obtained desired results.

11.2.5 Concatenating Neural Networks and Fuzzy Systems

This category includes the methods equally using fuzzy systems and neural networks to improve system performance.

(1) Parallel combination

This combination is for correction of the output of a fuzzy system with the output of a neural network to increase the precision of the final system output. Fig. 11.(a) shows this combination.

If a fuzzy system exists and an input-output data set is available, this model can be used to improve the performance without modifying the existing fuzzy systems.

For example, there is a fuzzy system working, but to improve its performance we need to incorporate more inputs to the fuzzy system.

Usually designing a new fuzzy system is very costly. So, rather than redesigning the entire fuzzy systems to handle the extra inputs, a neural network is added that it manages the new input and corrects the output.

Examples of this method can be found in consumer products. The consumer product companies usually upgrade their products by adding some functionality. Consequently, the controller of the upgraded product usually requires more inputs to deal with such new functions while it performs almost the same functions as the controller of the old model.

In this case, modifying old one can be better than developing a totally new controller. Redesigning the fuzzy system is thus avoided, which causes a substantial saving in development time and cost, because redesigning the membership functions becomes more difficult as the number of inputs increases. (Fig 11.16.) shows the schematic underlying a Japanese washing machine. The fuzzy system is a part of the first model. Later in order to improve the performance, more inputs are added and a neural network is built to correct the output with the new inputs. The additional input fed only to the neural network is electrical conductivity, which is used to determine washing time, rinsing time and spinning time. To train the neural network, the desired correction is used. This value is the difference between the desired output and what the fuzzy system outputs.

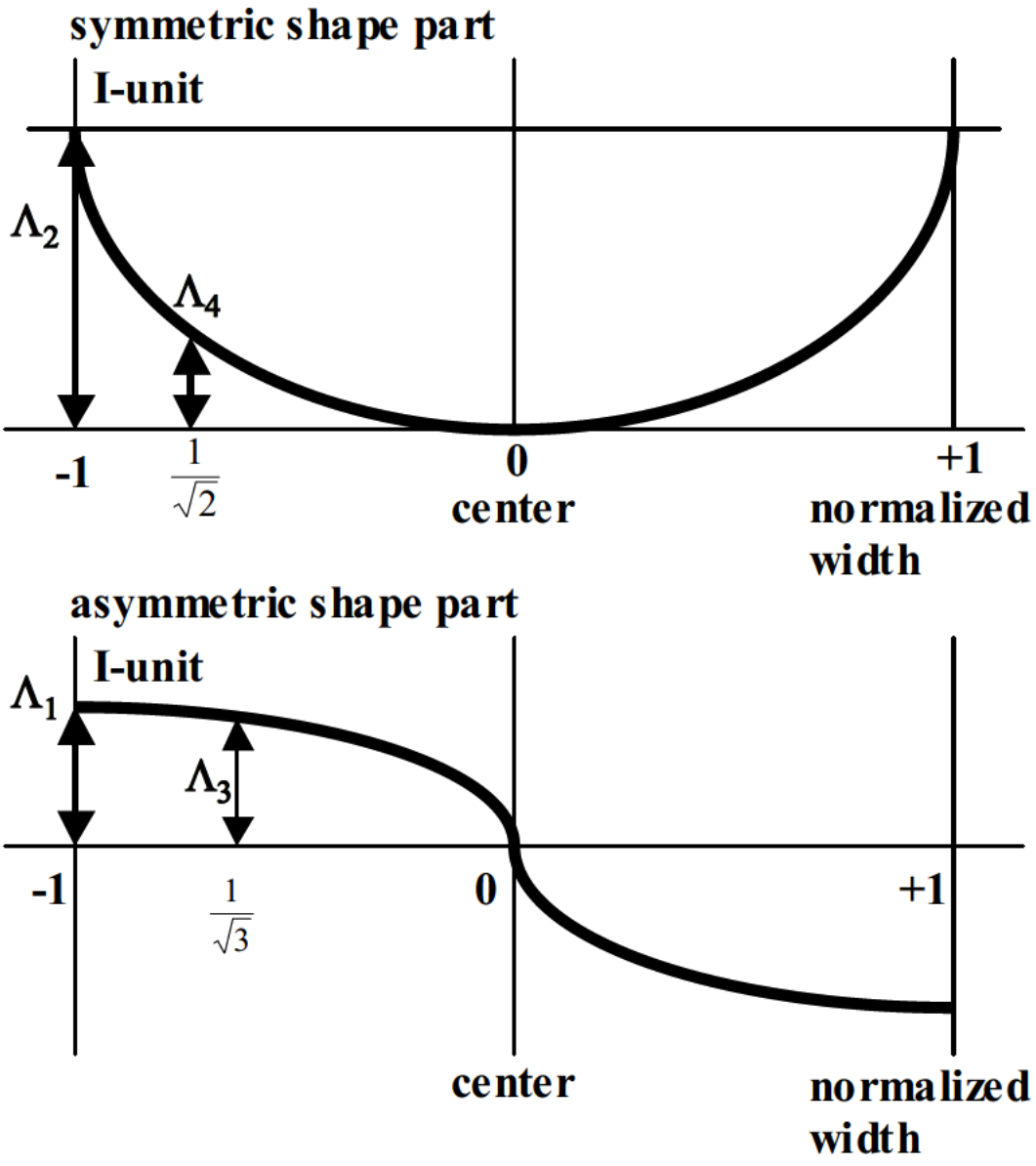
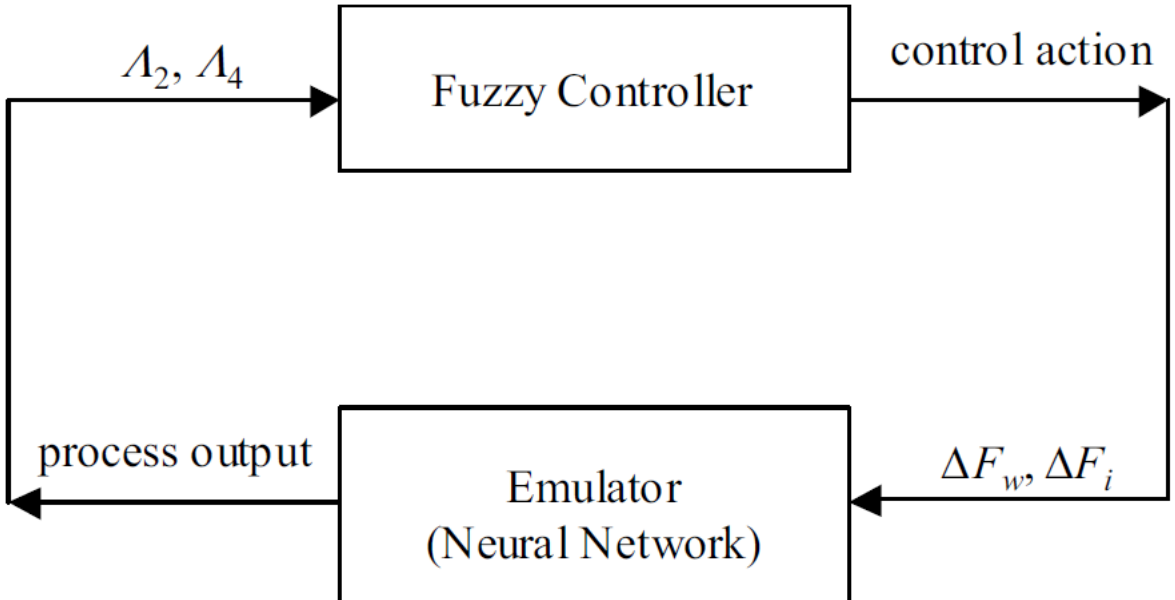


Fig. 11.13. Representation of the shape parameters



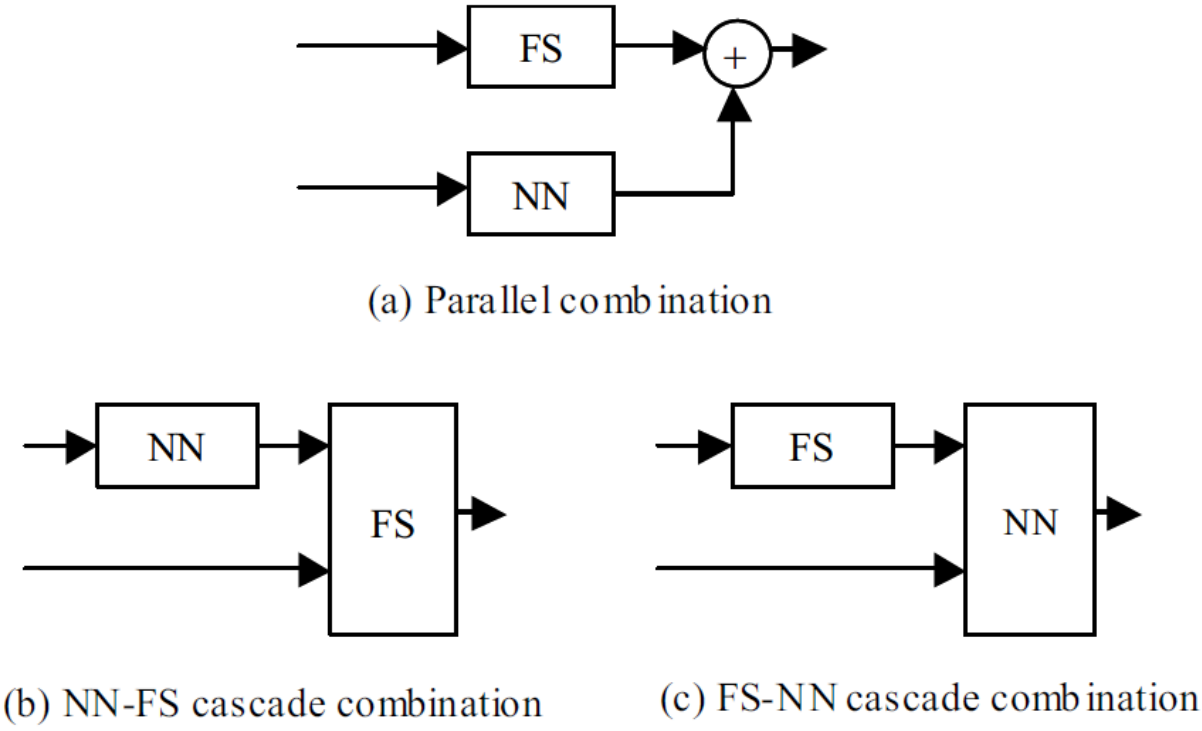


Fig. 11.15. Possible concatenations of NNs and FSs

(2) Cascade combination

The second is the cascade combination. It is a system where the output of the fuzzy system or neural network becomes the input of the other.

(b) cascade combination of NN and FS

Value corresponding to distance (Fig 11.15.(b)) shows the model where the output of a fuzzy system is inputted to a neural network and (Fig 11.15.(c)) shows the reverse model. An example of (Fig 11.15.(c)) can be found in Japanese electric fan. One of the requirement of it is to rotate toward the user by detecting the signal of remote controller. For the detection of the signal, three sensors are equipped on the fan body as shown in (Fig 11.17.(a)). In order to estimate the angle from the center of fan to the user, □ , a neural network and a fuzzy system are used. The fuzzy system estimates the distance from the fan to the user by using the sensors.

Module IV

Applications of Neuro-Fuzzy computing: Hydrologic Modelling time series Analysis and modeling, Remote sensing, Environmental Modelling, Construction Management, Fault detection and rehabilitation of structures, Water Management, Prediction of Pile capacity, Transportation/ Traffic planning.

In the past decades, vast literature had drawn attention to the Box-Jenkins method for modelling of hydrologic events. This method in the class of empirical models opposes physical models to avoid considering complicated physical processes among the hydrological variables for modelling a system. Empirical or black box models deal with the undertaken system as a black box which the input to and output from the box are solely taken into account. Acceptable forecasting results have been reported for the application of Box-Jenkins or commonly said autoregressive integrated moving average (ARIMA) models in modelling of water resources systems (Ahmed and Sarma, 2007). However, a major problem with such models is that they are linear and stationary assumption based so that the modelling of a non-linear hydrological phenomenon will probably produce less accurate and reliable forecasts. Recently, artificial intelligent (AI) models that emulate human thinking ability and brain structure are increasingly used in hydrological forecasting context. Construction of these models depends on the data and there is no need of prior knowledge of the system under consideration of the so called data driven models. However, there has been little discussion on computer paradigms, which are often highly data dependant and their performance relies on model specification and ability to cope with dynamic changes of events. Model specification is based on the existing knowledge of hydrological system and data dependency is referred to the availability of data. One of the most significant findings in dealing with the aforementioned debate is soft computing technique. This technique offers a more flexible, less assumption dependent, and potentially self adaptive approach for modelling of water-level and other dynamic and nonlinear hydrological processes (See and Openshaw, 1999). The artificial neural networks (ANN) and neuro-fuzzy (N-F) model that is a hybrid method by preserving the learning ability of ANNs and fuzzy reasoning are of the two important AI models in which the latter one is considered under soft computing class of models. ANN in the class of black box models became popular among the hydrologists in the recent past. A large number of processing elements with their inter-connections constitutes the

artificial neural networks. ANNs, or shortly neural network (NN), follow the cognition process of the human brain. Neural networks are helpful and efficient to cope with the systems that the characteristics of the processes are difficult to be described by deterministic or stochastic equations. It has been demonstrated that neural networks do not require either a priori detailed understanding of physical characteristics of the catchments or extensive data preprocessing. Moreover, NNs can handle incomplete, noisy and ambiguous data (Singh and Deo, 2007; Thirumalaiah and Deo, 1998). In recent years, there has been an increasing amount of literature on the application of ANN models in water resources engineering and managements and many other aspects of hydrology. Moreover, existing literature on rainfall-runoff modelling (Tokar and Johnson, 1999), flood forecasting (Toth et al., 2000), water quality assessment (Maier et al., 2004), evapotranspiration study (Kumar et al., 2002), groundwater prediction (Daliakopoulos et al., 2005), soil water evaporation (Han and Felker, 1997), and prediction of sediment volume (Kisi, 2007) strongly support the potential of neural networks in water resources modelling. To have a comprehensive review of this technique in water resources applications, readers are referred to the article of Maier and Dandy (2000). On the other hand, N-F models are rapidly becoming conventional in either academic or industrial application when compared to other nonlinear identification techniques. The forte of N-F systems, which has made it unique, is that the semantic transparency of rule-based fuzzy systems is combined with the learning ability of neural networks (Zounemat-Kermani and Teshnehlab, 2008). Thanks to preserving ANNs ability and fuzzy logic, N-F models can be regarded as a gray box technique (Babuška and Verbruggen, 2003). The method of applying the learning ability of artificial neural networks to the fuzzy models or fuzzy inference systems (FIS) is called N-F modelling (Jang et al., 1997). Moreover, N-F models describe the systems using fuzzy if–then rules, represented in an adaptive network structure that is trained by a NN learning algorithm. The gray box models, N-F models, are more comprehendible to users than completely black-box models, such as ANNs. The N-F models or generally called adaptive neuro-fuzzy inference systems (ANFIS) (Jang, 1993), benefits from less training time because of their smaller dimensions and the network initialization with parameters relating to the problem domain. Such results put emphasis on the advantages of combination of fuzzy logic and neural network technology as it provides an accurate Galavi and Shui 2113 initialization of the network in terms of the parameters of the fuzzy reasoning system (Aqil et al., 2007; Yurdusev and Firat, 2009). The N-F approach and particularly ANFIS, as a

multilayer feed forward network with the ability to combine the verbal power of a fuzzy system with numeric power of a neural system is becoming a powerful alternative in modelling numerous processes (Chang and Chang, 2006). More recently, literature has found the application of ANFIS in many fields, such as, regional electricity loads (Ying and Pan, 2008), ophthalmology (Güler and Übeyli, 2005), reservoir operation (Dubrovin et al., 2002), wind speed (Sfetsos, 2000), evaporation (Kisi, 2006), river flow (Firat, 2008) prediction, etc. Soft computing by combining several different computing paradigms, such as ANN, fuzzy-logic (FL) and genetic algorithm (GA) tries to find a new well-suited model to the system (Zadeh, 1994). Several attempts have been made to compare the ANFIS capabilities as a soft computing method with model driven methods, such as ARIMA models that have been conventionally used in water resources forecasting context. Although, literature has demonstrated the superiority of ANFIS models against ARIMA models (Firat, 2008), its superiority is influenced by the case study conditions and in some cases it does not outperform the ARIMA models (Altunkaynak et al., 2005). Moreover, there is no standard model building to cope with all possible case study conditions. However, extensive literature has suggested that N-F approach can be an effective alternative to the real problems when compared with conventionally used models (Altunkaynak and Sen, 2007). ANFIS models performance in river flow forecasting context has shown significant improvement in terms of computational speed, forecast errors, efficiency and peak flow estimation against ARIMA and ANN models (Shu and Ouarda, 2007). ANFIS models beside preserving reasonable forecasts accuracy in stream flow prediction have shown the capability to estimate peak flows more effectively than the low flows (Swain and Umamahesh, 2004). This investigation was done using two different membership functions in fuzzification step of model building for peak flows and low flows. In contrary to the aforementioned findings, consistent underestimation of peak flows is also reported, while low and medium flows forecasts have been more accurate (Aqil et al., 2007). This contradiction in prediction result implies that ANFIS models are case-specific and result may be different at various case studies. The effectiveness of human knowledge interference in ANFIS modelling was investigated by Chang and Chang (2006) in a study on prediction of water level at Shihmen reservoir, Taiwan. Two different N-F models were constructed based on the knowledge of case study. One was developed to forecast the water level solely based on upstream flow pattern of the reservoir, while human knowledge interfered model additionally entered current outflow of the reservoir

into the model. Using the same ANFIS architecture for both models, results demonstrated that expert's knowledge based model produces consistently superior precision than the other one.

Reservoirs are the most important and effective water storage facilities in modifying uneven distribution of water both in space and time. They not only provide water, hydroelectric energy and irrigation, but also smooth out extreme inflows to mitigate floods or droughts. To make the best use of the available water, the optimal operation of reservoirs in a system is undoubtedly very important. Reservoir operation requires a series of decisions that determine the accumulation and release of water over time. In the face of natural uncertainty, forecasts of future reservoir inflow can be helpful in making efficient operating decisions. With the high mountains and steep slopes of Taiwan, heavy rainfall, especially in a typhoon event, could cause downstream flooding within a few hours. Nevertheless, the typhoon events, usually coupled with abundant rainfall, are the most valuable yearly water resource. Accurate prediction of inflow rates is crucial not only for optimizing the management of water resources but for sustaining the safety of a reservoir. For decades, a wide variety of approaches have been proposed for streamflow forecasting including statistical (black-box) and physical (conceptual) models (e.g. [9,11,32]). Owing to the strongly non-linear, high degree of uncertainty, and time-varying characteristics of the hydrosystem, none of them can be considered as a single superior model [34]. An accurate site-specific prediction remains a difficult task. Recently, artificial neural networks (ANNs) have been accepted as a potentially useful tool for modeling complex non-linear systems and widely used for prediction [16]. In the hydrological forecasting context, ANNs have also proven to be an efficient alternative to traditional methods for rainfall forecasting [13,15,26], streamflow forecasting [2–4,6,10,18,22,36,39], groundwater modeling [24,37], and reservoir operation [17,19,30]. The ASCE Task Committee report [40,41] did a comprehensive review of the application of ANNs to hydrology, as did Maier and Dandy [27], and also Govindaraju and Rao [14] in a specialized publication. In this study, we present a novel neuro-fuzzy approach, namely adaptive neuro-fuzzy inference system (ANFIS), in forecasting 1–3 hours-ahead water level of a reservoir during flood periods. To verify its applicability, the Shihmen reservoir, Taiwan, was chosen as the study area. Because the reservoir water level is a control system, its variability cannot be solely determined by meteorological effects. Human knowledge and its operating decisions could significantly change the status of water level within

a short period; consequently, a suitable prediction model should include upstream conditions and human decisions. To demonstrate that the neuro-fuzzy network has the ability to deal with human knowledge and enhance the model performance, we developed two ANFIS models for water level forecasting, one with a human decision of reservoir outflow as input variable, another without.

The identification of suitable models for forecasting future monthly inflows to hydropower reservoirs is a significant precondition for effective reservoir management and scheduling. The results, especially in long-term prediction, are useful in many water resources applications such as environment protection, drought management, operation of water supply utilities, optimal reservoir operation involving multiple objectives of irrigation, hydropower generation, and sustainable development of water resources. As such, hydrologic time series forecasting has always been of particular interest in operational hydrology. It has received tremendous attention of researchers in last few decades and many models for hydrologic time series forecasting have been proposed to improve the hydrology forecasting. These models can be broadly divided into three groups: regression based methods, time series models and AI-based methods. For autoregressive moving-average models (ARMA) proposed by Box and Jenkins (1970), it is assumed that the times series is stationaryand follows the normal distribution. Since Carlson et al. (1970) presented significant developments in the form of ARMA models of the hydrologic times series, ARMA has been one of the most popular hydrologic times series models for reservoir design and optimization. ARMA is also applied to monthly hydrologic time series (Hipel and Mcleod, 1994). Extensive application and reviews of the several classes of such models proposed for the modeling of water resources time series were reported (Chen and Rao, 2002; Salas, 1993; Srikanthan and McMahon, 2001; Toth et al., 2000; Arena et al., 2006; Komornik et al., 2006). In recent years, AI technique, being capable of analyzing longseries and large-scale data, has become increasingly popular in hydrology and water resources among researchers and practicing engineers. Since the 1990s, artificial neural networks (ANNs), based on the understanding of the brain and nervous systems, was gradually used in hydrological prediction. An extensive review of their use in the hydrological field is given by ASCE Task Committee on Application of Artificial Neural Networks in Hydrology (ASCE, 2000a,b). Silverman and Dracup (2000) predicted the pattern of rainfall in California using an ANN model.

Olsson et al. (2004) used the ANN technique to determine rainfall occurrence and rainfall intensity during rainy periods. Freiwan and Cigizoglu(2005) investigated the potential of feed-forward ANN technique in prediction of monthly precipitation amount. ANNs have been shown to give useful results in many fields of hydrology and water resources research (Alvisi et al., 2006; Campolo et al., 2003; Chau, 2006; Chang et al., 2002; Muttil and Chau, 2006; Sudheer et al., 2002; Sudheer and Jain, 2004; Wang et al., 2008). The adaptive neural-based fuzzy inference system (ANFIS) model and its principles, first developed by Jang (1993), have been applied to study many problems and also in hydrology field as well. Chang and Chang (2001) studied the integration of a neural network and fuzzy arithmetic for real-time stream flow forecasting and reported that ANFIS helps to ensure more efficient reservoir operation than the classical models based on rule curve. Loukas (2001) developed an adaptive neuro-fuzzy inference system (ANFIS) for obtaining sufficient quantitative structure–activity relationships (QSAR) with high accuracy. Bazartseren et al. (2003) used neuro-fuzzy and neural network models for shortterm water level prediction. Nayak et al. (2004) evaluated the potential of neuro-fuzzy technique in forecasting river flow time series. Nayak et al. (2005) used a neuro-fuzzy model for short-term flood forecasting. Dixon (2005) examined the sensitivity of neurofuzzy models used to predict ground-water vulnerability in a spatial context by integrating GIS and neuro-fuzzy techniques. Shu and Ouarda (2008) used ANFIS for flood quantile estimation at ungauged sites. Other researchers reported good results in applying ANFIS in hydrological prediction (Cheng et al., 2005; Firat and Gungor,2008; Keskin et al., 2006; Zounemat-Kermaniand and Teshnehlab, 2008). Genetic Programming (GP), an extension of the well known field of genetic algorithms (GA) belonging to the family of evolutionary computation, is an automatic programming technique for evolving computer programs to solve problems (Koza, 1992). GP model was used to emulate the rainfall–runoff process (Whigam and Crapper, 2001) and was evaluated in terms of root mean square error and correlation coefficient (Liong et al., 2002; Whigam and Crapper, 2001). It was shown to be a viable alternative to traditional rainfall–runoff models. The GP approach was also employed by Johari et al. (2006) to predict the soil–water characteristic curve of soils. GP is employed for modeling and prediction of algal blooms in Tolo Harbour, Hong Kong (Muttil and Chau, 2006) and the results indicated good predictions of long-term trends in algal biomass. The Darwinian theory-based GP approach was suggested for improving fortnightly flow forecast for a short time series (Sivapragasam et al., 2007). Guven et al. (2008)

used GP-based empirical model for daily reference evapotranspiration estimation. The support vector machine (SVM) is based on structural risk minimization (SRM) principle and is an approximation implementation of the method of SRM with a good generalization capability (Vapnik, 1998). Although SVM has been used in applications for a relatively short time, this learning machine has been proven to be a robust and competent algorithm for both classification and regression in many disciplines. Recently, the use of the SVM in water resources engineering has attracted much attention. Dibike et al. (2001) demonstrated its use in rainfall–runoff modeling. Liong and Sivapragasam (2002) applied SVM to flood stage forecasting in Dhaka, Bangladesh and concluded that the accuracy of SVM exceeded that of ANN in one-lead-day to seven-lead-day forecasting. Yu et al. (2006) successfully explored the usefulness of SVM based modeling technique for predicting of real-time flood stage forecasting on Lan-Yang river in Taiwan 1–6 h ahead. Khan and Coulibaly (2006) demonstrated the application of SVM to time series modeling in water resources engineering for lake water level prediction. Wu et al. (2008) used a distributed support vector regression for river stage prediction. The SVM method has also been employed for stream flow predictions (Asefa et al., 2006; Lin et al., 2006). A number of publications have addressed hydrological time series (Carlson et al., 1970; Chang et al., 2002; Chen and Rao, 2002, 2003; Cheng et al., 2005; Firat and Gungor, 2008; Hu et al., 2001; Keskin et al., 2006; Lin et al., 2006; Nayak et al., 2004; Salas, 1993; Sivapragasam et al., 2007). Komornik et al. (2006) compared the forecasting performance of several nonlinear time series models with respect to their capabilities of forecasting monthly and seasonal flows in the Tatry region. Lin et al. (2006) used the support vector machine for hydrological prediction. Jain and Kumar (2007) presented the use of combining ANNs and traditional time series approaches for achieving improved accuracies in hydrological time series forecasting. Cheng et al. (2005), Zounemat-Kermani and Teshnehlab (2008) used adaptive neuro-fuzzy inference system for hydrological time series prediction. But none of the previous publications provide comparison for using these techniques. Therefore, the major objectives of the study presented in this paper are to investigate several AI techniques for modeling monthly discharge time series, which include ANN approaches, ANFIS techniques, GP models and SVM method, and to compare their performance with other traditional time series modeling techniques such as ARMA. Four quantitative standard statistical performance evaluation measures, i.e., coefficient of correlation (R), Nash–Sutcliffe efficiency coefficient (E), root mean squared error (RMSE), mean absolute

percentage error (MAPE), are employed to validate all models. Brief introduction and model development of these AI methods are also described before discussing the results and making concluding remarks. The performances of various models developed are demonstrated by forecasting monthly river flow discharges in Manwan Hydropower and Hongjiadu Hydropower.