Course Plan for SOFT COMPUTING (3-0-0) (BCS 422)

Class-1: Basic tools of soft Computing – Fuzzy logic, Neural Networks and Evolutionary Computing ,	Class-18: Activation functions. Adaline: its training and capabilities, weights learning,
Class-2: Approximations of Multivariate functions,	Class-19 -22: MLP : error back propagation, generalized delta rule.
Class-3 &4: Non – linear Error surface and optimization.	Class-23: Radial basis function networks and least square training algorithm,
Class-5: Fuzzy Logic Systems : Basics of fuzzy logic theory,	Class-24: Kohenen self – organizing map and learning vector quantization networks.
Class-6: Crisp and fuzzy sets.	Class_25, 26 & 27: Recurrent neural networks,
Class-7, 8 & 9: Basic set operations.	Class-28: Simulated annealing neural networks.
Class-10: Fuzzy relations and their Composition rules	Class-29 & 30: Adaptive neuro-fuzzy information systems (ANFIS),
Class-11 &12: Fuzzy inference,	Class-31 & 32: Applications to control and pattern recognition.
Class-13: Zadeh's compositional rule of inference.	Class-33**:** Genetic algorithms : Basic concepts, fitness function
Class-14: Defuzzificaiton.	Class-34-36: encoding ,
Class-15: Fuzzy logic control: Mamdani and Takagi and Sugeno architectures.	Class-37: reproduction
Class-16: Applications to pattern recognition.	Class-38:Differences of GA and traditional optimization methods.
Class-17**:** Neural networks : Single layer networks, Perceptron.	Class-39 & 40:Basic genetic programming concepts Applications

Soft Computing

Soft computing differs from conventional or hard computing in that, unlike hard computing, it is tolerant of imprecision, uncertainty, partial truth, and approximate reasoning. Conventional or hard computing requires a precisely stated analytical model and a lot of mathematical and logical computation.

Components of soft computing include:

Neural networks (NN),

Fuzzy logic (FL),

Evolutionary computation (EC), (which includes Evolutionary algorithms, Genetic algorithms, Differential evolution),

Meta-heuristics and Swarm Intelligence, (which includes Ant colony optimization, Particle swarm optimization, Firefly algorithm, Cuckoo search),

Support Vector Machines (SVM),

Probabilistic reasoning, (which includes Bayesian network), and Chaos theory.

NOTE:

- **SC is evolving rapidly**
- **New techniques and applications are constantly being proposed**

Generally speaking, soft computing techniques resemble biological processes more closely than traditional techniques, which are largely based on formal logical systems, such as sentential logic and predicate logic, or rely heavily on computer-aided numerical analysis (as in finite element analysis). Soft computing techniques are intended to complement each other.

Unlike hard computing schemes, which strive for exactness and full truth, soft computing techniques exploit the given tolerance of imprecision, partial truth, and uncertainty for a particular problem. Another common contrast comes from the observation that inductive reasoning plays a larger role in soft computing than in hard computing.

Hard Computing	**Soft computing**
Precisely stated analytical model	Tolerant to imprecision, uncertainty, partial truth, approximation
Based on binary logic, crisp systems, numerical analysis, crisp software	Fuzzy logic, neural nets, probabilistic reasoning.
Programs are to be written	Evolve their own programs
Two values logic	Multi valued logic
Exact input data	Ambiguous and noisy data
Strictly sequential	Parallel computations
Precise answers	Approximate answers

Many contemporary problems do not lend themselves to precise solutions such as Recognition problems (handwriting, speech, objects, images), forecasting, combinatorial problems etc.

Soft Computing became a formal Computer Science area of study in the early 1990's.Earlier computational approaches could model and precisely analyze only relatively simple systems. More complex systems arising in biology, medicine, the humanities, management sciences, and similar fields often remained intractable to conventional mathematical and analytical methods. That said, it should be pointed out that simplicity and complexity of systems are relative, and many conventional mathematical models have been both challenging and very productive. Soft computing deals with imprecision, uncertainty, partial truth, and approximation to achieve tractability, robustness and low solution cost.

Implications of Soft Computing

Soft computing employs NN, SVM, FL etc, in a complementary rather than a competitive way. One example of a particularly effective combination is what has come to be known as "neuro-fuzzy systems." Such systems are becoming increasingly visible as consumer products ranging from air conditioners and washing machines to photocopiers, camcorders and many industrial applications.

Some Applications of Soft Computing

- Application of soft computing to handwriting recognition
- Application of soft computing to automotive systems and manufacturing
- Application of soft computing to image processing and data compression
- Application of soft computing to architecture
- Application of soft computing to decision-suppor systems
- Application of soft computing to power systems
- Neurofuzzy systems
- Fuzzy logic control

Overview of Techniques in Soft Computing

- Neural networks
- Fuzzy Logic
- Genetic Algorithms in Evolutionary Computation
- Support Vector Machines

Neural Networks:

Neural Networks, which are simplified models of the biological neuron system, is a massively parallel distributed processing system made up of highly interconnected neural computing elements that have the ability to learn and thereby acquire knowledge and making it available for use. It resembles the brain in two respects: (1) Knowledge is acquired by the network through a learning process and (2) Interconnection strengths known as synaptic weights are used to store the knowledge

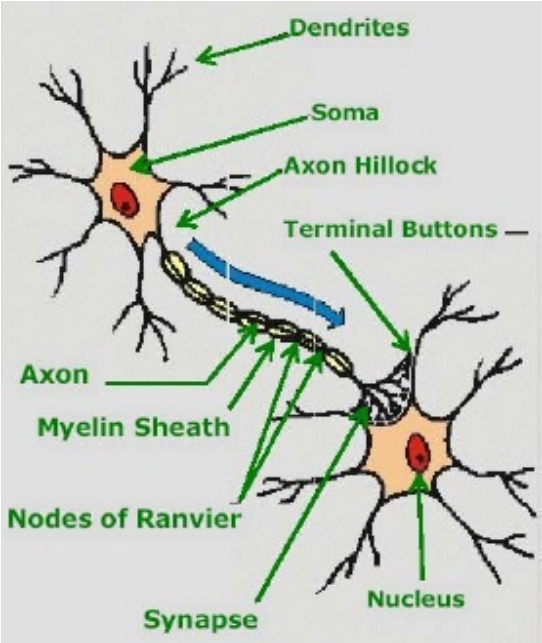


Fig. 1: A biological neuron connected to another neuron.

A neuron is composed of nucleus- a cell body known as soma. Attached to the soma are long irregularly shaped filaments called dendrites. The dendrites behave as input channels, all inputs from other neurons arrive through dendrites.

Another link to soma called Axon is electrically active and serves as an output channel. If the cumulative inputs received by the soma raise internal electric potential of the cell known as membrane potential, then the neuron fires by propagating the action potential down the axon to excite or inhibit other neurons. The axon terminates in a specialized contact called synapse that connects the axon with the dendrite links of another neuron. An artificial neuron model bears direct analogy to the actual constituents of biological neuron. This model forms basis of Artificial Neural Networks

Artificial Neural Networks (ANNs)

The ANN provides a general practical method for real-valued, discrete-valued, and vector-valued functions from examples. The back propagation algorithm which is widely used in ANNs, uses gradient descent to tune network parameters to best fit a training set of input-output pairs.

An artificial neural network (ANN) is a distributed computing scheme based on the structure of the nervous system of humans. The architecture of a neural network is formed by connecting multiple elementary processors, this being an adaptive system that has an algorithm to adjust their weights (free parameters) to achieve the performance requirements of the problem based on representative samples.

Examinations of the human's central nervous system inspired the concept of neural networks. In an Artificial Neural Network, simple artificial nodes, known as "neurons", "neurodes", processing elements" or "units", are connected together to form a network which mimics a biological neural network.

There is no single formal definition of what an artificial neural network is. However, a class of statistical models may commonly be called "Neural" if they possess the following characteristics:

1. consist of sets of adaptive weights, i.e. numerical parameters that are tuned by a learning algorithm, and
2. are capable of approximating non-linear functions of their inputs.

The adaptive weights are conceptually connection strengths between neurons, which are activated during training and prediction.

Neural networks are similar to biological neural networks in performing functions collectively and in parallel by the units, rather than there being a clear delineation of subtasks to which various units are assigned. The term "neural network" usually refers to models employed in statistics, cognitive psychology and artificial intelligence. Neural network models which emulate the central nervous system are part of theoretical neuroscience and computational neuroscience.

In modern software implementations of artificial neural networks, the approach inspired by biology has been largely abandoned for a more practical approach based on statistics and signal processing. In some of these systems, neural networks or parts of neural networks (like artificial neurons) form components in larger systems that combine both adaptive and non-adaptive elements. While the more general approach of such systems is more suitable for real-world problem solving, it has little to do with the traditional artificial intelligence connectionist models. What they do have in common, however, is the principle of non-linear, distributed, parallel and local processing and adaptation. Historically, the use of neural networks models marked a paradigm shift in the late eighties from high-level (symbolic) artificial intelligence, characterized by expert systems with knowledge embodied in *if-then* rules, to low-level (sub-symbolic) machine learning, characterized by knowledge embodied in the parameters of a dynamical system.

Biological Motivation

- Human brain is a densely interconnected network of approximately 10^{11} neurons, each connected to, on average, 10^{4} others.
- Neuron activity is *excited* or *inhibited* through connections to other neurons.
- The fastest neuron switching times are known to be on the order of 10^{-3} sec.

Genetic Algorithm:

Genetic algorithms (GAs) are search methods based on principles of natural selection and genetics. GAs were First described by John Holland in the 1960s and further developed by Holland and his students and colleagues at the University of Michigan in the 1960s and 1970s. Holland's goal was to understand the phenomenon of \adaptation" as it occurs in nature and to develop ways in which the mechanisms of natural adaptation might be imported into computer systems. Holland's 1975 book Adaptation in Natural and Artificial Systems (Holland, 1975/1992) presented the GA as an abstraction of biological evolution and gave a theoretical framework for adaptation under the GA.

The population size, which is usually a user-specified parameter, is one of the important factors affecting the scalability and performance of genetic algorithms. For example, small population sizes might lead to premature convergence and yield substandard solutions. On the other hand, large population sizes lead to unnecessary expenditure of valuable computational time. Once the problem is encoded in a chromosomal manner and a fitness measure for discriminating good solutions from bad ones has been chosen, we can start to evolve solutions to the search problem using the following steps:

1. Initialization: The initial population of candidate solutions is usually generated randomly across the search space. However, domain-specific knowledge or other information can be easily incorporated.
2. Evaluation: Once the population is initialized or an offspring population is created, the fitness values of the candidate solutions are evaluated.
3. Selection: Selection allocates more copies of those solutions with higher fitness values and thus imposes the survival-of-the-fittest mechanism on the candidate solutions. The main idea of selection is to prefer better solutions to worse ones, and many selection procedures have been proposed to accomplish this idea, including roulette-wheel selection, stochastic universal selection, ranking selection and tournament selection, some of which are described in the next section.
4. Recombination: Recombination combines parts of two or more parental solutions to create new, possibly better solutions (i.e. offspring). There are many ways of accomplishing this (some of which are discussed in the next section), and competent performance depends on a properly designed recombination mechanism. The offspring under recombination will not be identical to any particular parent and will instead combine parental traits in a novel manner
5. Mutation: While recombination operates on two or more parental chromosomes, mutation locally but randomly modifies a solution. Again, there are many variations of mutation, but it usually involves one or more changes being made to an individual’s trait or traits. In other words, mutation performs a random walk in the vicinity of a candidate solution.
6. Replacement. The offspring population created by selection, recombination, and mutation replaces the original parental population. Many replacement techniques such as elitist replacement, generation-wise replacement and steady-state replacement methods are used in GAs.
7. Repeat steps 2–6 until a terminating condition is met. Goldberg has likened GAs to mechanistic versions of certain modes of human innovation and has shown that these operators when analyzed individually are ineffective, but when combined together they can work well.

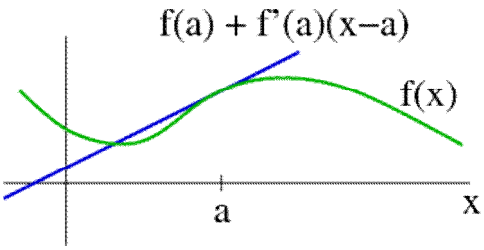
Introduction to Taylor's theorem for multivariable functions

Remember one-variable calculus Taylor's theorem. Given a one variable function $f(x)$, you can fit it with a polynomial around $x=a$.

For example, the best linear approximation for $f(x)$ is

$$f(x)\approx f(a)+f'(a)(x-a).$$

This linear approximation fits $f(x)$ (shown in green below) with a line (shown in blue) through $x=a$ that matches the slope of f at a.



We can add additional, higher-order terms, to approximate $f(x)$ better near a. The best quadratic approximation is

$$f(x)\approx f(a)+f'(a)(x-a)+12f''(a)(x-a)2$$

We could add third-order or even higher-order terms:

$$f(x)\approx f(a)+f'(a)(x-a)+12f''(a)(x-a)2+16f'''(a)(x-a)3+\cdots.$$

The important point is that this *Taylor polynomial* approximates $f(x)$ well for x near a.

We want to generalize the Taylor polynomial to (scalar-valued) functions of multiple variables:

$$f(\mathbf{x})=f(x1,x2,\ldots,xn).$$

We already know the best linear approximation to f. It involves the derivative,

$$f(\mathbf{x})\approx f(\mathbf{a})+Df(\mathbf{a})(\mathbf{x}-\mathbf{a}).$$

where $Df(\mathbf{a})$ is the matrix of partial derivatives. The linear approximation is the first-order Taylor polynomial.

What about the second-order Taylor polynomial? To find a quadratic approximation, we need to add quadratic terms to our linear approximation. For a function of one-variable $f(x)$, the quadratic term was

$$12f''(a)(x-a)2.$$

For a function of multiple variables $f(\mathbf{x})$, what is analogous to the second derivative?

Since $f(\mathbf{x})$ is scalar, the first derivative is $Df(\mathbf{x})$, a $1\times n$ matrix, which we can view as an n-dimensional vector-valued function of the n-dimensional vector $\mathbf{x}$. For the second derivative of $f(\mathbf{x})$, we can take the matrix of partial derivatives of the function $Df(\mathbf{x})$. We could write it as $DDf(\mathbf{x})$ for the moment. This second derivative matrix is an $n\times n$ matrix called the **Hessian matrix** of f. We'll denote it by $Hf(\mathbf{x})$,

$$Hf(\mathbf{x})=DDf(\mathbf{x}).$$

When f is a function of multiple variables, the second derivative term in the Taylor series will use the Hessian $Hf(\mathbf{a})$. For the single-variable case, we could rewrite the quadratic expression as

$$12(x-a)f''(a)(x-a).$$

The analog of this expression for the multivariable case is

$$12(\mathbf{x}-\mathbf{a})THf(\mathbf{a})(\mathbf{x}-\mathbf{a}).$$

We can add the above expression to our first-order Taylor polynomial to obtain the second-order Taylor polynomial for functions of multiple variables:

$$f(\mathbf{x})\approx f(\mathbf{a})+Df(\mathbf{a})(\mathbf{x}-\mathbf{a})+12(\mathbf{x}-\mathbf{a})THf(\mathbf{a})(\mathbf{x}-\mathbf{a}).$$

The second-order Taylor polynomial is a better approximation of $f(\mathbf{x})$ near $\mathbf{x}=\mathbf{a}$ than is the linear approximation (which is the same as the first-order Taylor polynomial). We'll be able to use it for things such as finding a local minimum or local maximum of the function $f(\mathbf{x})$.

INTRODUCTION TO THE CONCEPT OF F FUZZY LOGIC

In many cases in our day-to-day work we do not do exact measurements like washing a cloth, driving a vehicle, brushing the teeth, adding salt or sugar to cooked food, etc. So also in many cases of our day-to-day conversation we use statements like more or less tall, too fat, very thin, slightly dwarf, etc which do not convey exact information, but conveniently tackle the situations. In these situations it is not critical to be precise in data rather a rough answer solves the problem. Fuzzy logic is all about this relative importance of precision. Hence fuzzy logics are used in cases of imprecision and imprecise human reasoning and human perception. In cases of high precision problems where exact mathematical models are already developed there the fuzzy logics are not used. Still fuzzy logics are used to drill printed circuit boards under LASER to an appreciate precision with low cost Hence for cost effectiveness we may prefer fuzzy than exact mathematical model in many cases.

Applications of Fuzzy Logic:

Some already developed areas of applications of Fuzzy Logic are given below.

1) Fuzzy focusing and image stabilization.
2) Fuzzy air conditioner that controls temperature according to comfort index.
3) Fuzzy washing machine that washes clothes with help of smart sensors.
4) Fuzzy controlled sub-way systems.
5) Fuzzy controlled toasters, rice cookers, vacuum cleaners, etc.

Advantages of Fuzzy Logic

1) Fuzzy logic is conceptually easy to understand.
2) Fuzzy logic is flexible that can add more functionality on top of it without starting afresh.
3) Fuzzy logic is tolerant of imprecise data.
4) Fuzzy logic can model non-linear functions of arbitrary complexity.
5) Fuzzy logic can be added to conventional control system easily.
6) Fuzzy logic can be built on the top of the experience of experts.
7) Fuzzy logic is based on natural language.

FUZZY SET THEORY

Classical set: A classical or crisp set has a clear and unambiguous boundary that separates the members of a classical set from members not belonging to the set. For example a classical set of positive integers defined as:

$A_c = \{ x \mid x < 5 \}$ or $A_c = \{1, 2, 3, 4\}$

Here the whole numbers less than 5 belongs to the set A_c and the number 5 and more than 5 do not belong to the set A_c. Hence number 5 is acting as a boundary.

These classical sets are suitable for various applications and have proven to be an important tool for mathematics and computer science, but they do not reflect the nature of human concepts and thoughts, which tends to be abstract and imprecise, as an example of a set of young men.

Classical Set vs Fuzzy set with example of making a set of tall persons

No	Name	Height (cm)	Degree of Membership of "tall men"	
			Crisp	Fuzzy
1	Boy	206	1	1
2	Martin	190	1	1
3	Dewanto	175	0	0.8
4	Joko	160	0	0.7
5	Kom	155	0	0.4

- What is fuzzy thinking
  - Experts rely on common sense when they solve the problems
  - How can we represent expert knowledge that uses vague and ambiguous terms in a computer
  - Fuzzy logic is not logic that is fuzzy but logic that is used to describe the fuzziness. Fuzzy logic is the theory of fuzzy sets, set that calibrate the vagueness.
  - Fuzzy logic is based on the idea that all things admit of degrees. Temperature, height, speed, distance, beauty – all come on a sliding scale.
    Jim is *tall* guy
    It is really *very* hot today

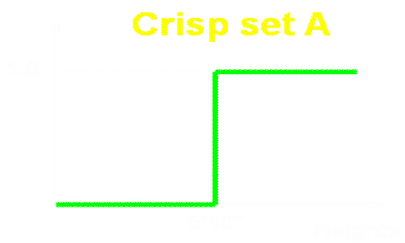
Fuzzy Set: It is a set without crisp or sharp boundary. Here the transition from "belonging to a set" to "not belonging to a set" is gradual. This is done by assigning a membership function to the elements or members of the set. The members of a fuzzy set get a membership value within 0 and 1. Here it is obvious that the membership value 0 indicates that the element is not a member of the fuzzy set.

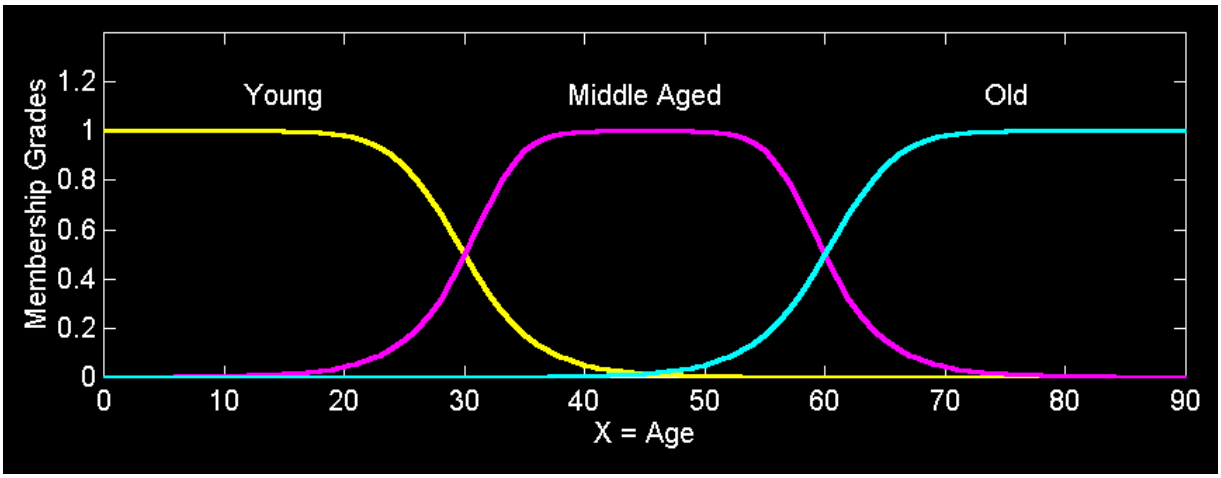
Definition: If X is a collection of objects denoted generically by x, then fuzzy set A_f in X is defined as a set of ordered pairs:

$$A_f = \{ (x, \mu_A(x)) \mid x \in X \}$$

where $\mu_A(x)$ is called the membership function for the fuzzy set A_f.

The membership function (MF)maps each element of X to a membership grade or membership value between 0 and 1.





- **Boolean logic**
  - **Uses sharp distinctions. It forces us to draw a line between a members of class and non members.**
- **Fuzzy logic**
  - **Reflects how people think. It attempt to model our senses of words, our decision making and our common sense -> more human and intelligent systems**

Several types of Fuzzy sets and their notations

(1) A Fuzzy set with discrete non-ordered Universe: Let X = {Kolkata, Delhi, Chennai, Mumbai, Bangalore} be the set of cities one may choose to live in. The fuzzy set A = "desirable city to live in" may be defined as (using comma notation):

A = { (Mumbai, 0.5), (Delhi, 0.6), (Chennai, 0.3), (Bangalore, 0.7), (Kolkata, 0.8) }

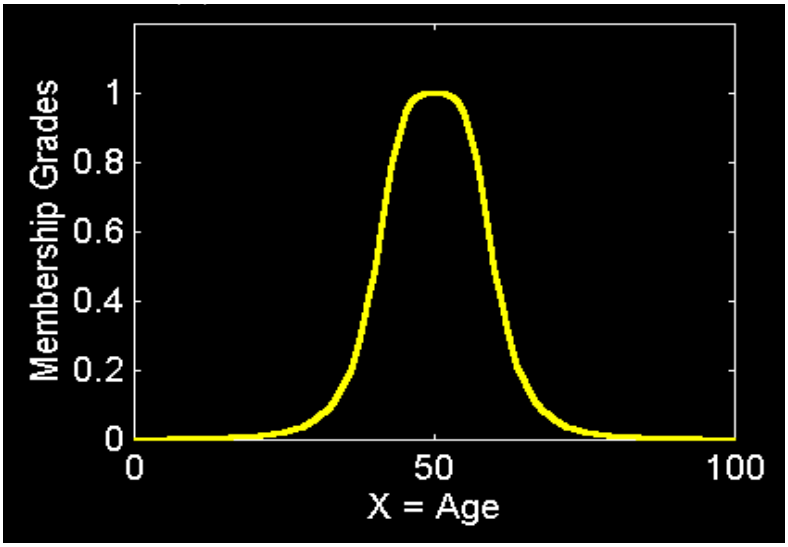
(2) A Fuzzy set with discrete ordered Universe: Let 'U' be a number of children from 1 to 5, that is, U = {1,2, 3, 4, 5}. Now the fuzzy set B = "number of children a couple will wish to have " may be defined as (using summation notation):

B = (0.5/1 + 0.9/2 + 0.4/3 + 0.2/4 + 0.1/5)

(3) A Fuzzy set with continuous Universe: Let X = R^+ be the set of possible age of human beings. Then the fuzzy set C = "a 50 year old man" may be defined as:

$C = \{ (x, \mu_C (x) \mid x \in X \}$

where $\mu_C (x) = 1/ (1 + (x – 50/10)^{10})$



We observed that finding or designing an appropriate membership function has importance on solution, but there is neither a unique solution nor a unique membership function, to characterize a particular description. Once a fuzzy set is being defined by a membership function, then the membership function are not fuzzy, rather they are precise mathematical functions. Thus by having a fuzzy description with a membership function, we essentially defuzzify the fuzzy description. In other words, fuzzy set theory does not fuzzify the world rather they defuzzify the world by assigning a membership value to a fuzzy set elements.

Relation to probability theory: When we devise a fuzzy set, then there must be a unique membership function associated with it, which expresses the abstract concept of the fuzzy problem. Thus the membership functions are non-random in nature. Hence fuzzy should not be compared with probability theory which mostly deals with random phenomena. Fuzziness describes the ambiguity of an event where as randomness describes the uncertainty in the occurrence of the event.

Some nomenclature used in Fuzzy sets:

Consider three fuzzy sets and their membership functions (MFs) corresponding to "young", "middle aged", and "old" peoples which are plotted aside. Now we will define some nomenclature used in fuzzy logic.

(1) **Support:** The support of a fuzzy set A, denoted as support(A) or supp(A), is the crisp set of X whose elements all have non-zero membership values in A. Thus we can write

$\text{Support}(A) = \{ x \mid \mu_A(x) > 0 \}$

(2) **Core:** The core of a fuzzy set A is the set of all points x in X such that $\mu_A(x) = 1$. Thus we can write

$$\text{Core}(A) = \{x \mid \mu_A(x) = 1`\}$$

(3) **Normal fuzzy set and Sub normal fuzzy set:** A fuzzy set A is called normal if there exits an $x \in X$ such that $\mu_A(x) = 1$, that is, $\text{core}(A) \neq \varphi$. Otherwise the set is called sub-normal fuzzy set.

(4) Crossover Points: A crossover point of a fuzzy set A is a point $x \in X$ at which $\mu_A(x) = 0.5$. Thus we can write

$$\text{Crossover}(A) = \{ x \mid \mu_A(x) = 0.5 \}$$

(5) Fuzzy Singleton: A fuzzy set whose support is a single point in X with $\mu_A(x) = 1$, is called a fuzzy singleton.

(6) α- cut: The α- cut or α- level set of a fuzzy set A is a crisp set A_α or $[A]^\alpha$ that contains all the elements x in X such that $\mu_A(x) \geq \alpha$. Thus we can write

$$A_\alpha \text{ or } [A]^\alpha = \{ x \mid \mu_A(x) \geq \alpha\}$$

(7) Strong α- cut: Strong α- cut or strong α- level set A'_α is defined as

$$A_\alpha' = \{ x \mid \mu_A(x) \geq \alpha\}$$

Now we can write $\text{support}(A) = A'_0$ and $\text{core}(A) = A_1$

(8) Convex fuzzy set: A convex fuzzy set A is described by a membership function whose membership values are strictly monotonically increasing or whose membership values are strictly monotonically decreasing or whose membership values are strictly monotonically increasing and then strictly monotonically decreasing with increasing values of the elements in the Universe. Thus for any elements x_1, x_2 and x_3 with relation $x_1 < x_2 < x_3$ implies that

$$\mu_A(x_2) \geq \min \{ \mu_A(x_1), \mu_A(x_3) \}$$

(9) Fuzzy Numbers: If a is a convex continuous point normal fuzzy set defined on the real line (R), then A is often termed a fuzzy number.

(10) Quasi fuzzy number: A quasi fuzzy number A is a fuzzy set of real line with a normal fuzzy convex and continuous membership function satisfying the limit condition $\lim \mu_A(x) = 0$ as $x \to \infty$.

(11) Bandwidth: Band width of a normal and convex fuzzy set is defined as the distance between two unique crossover points. That is

$$\text{Width}(A) = |x_2 - x_1| \text{ where } \mu_A(x_1) = \mu_A(x_2) = 0.5$$

(12) Symmetricity

(13) Open left

(14) Open Right

(15) Closed

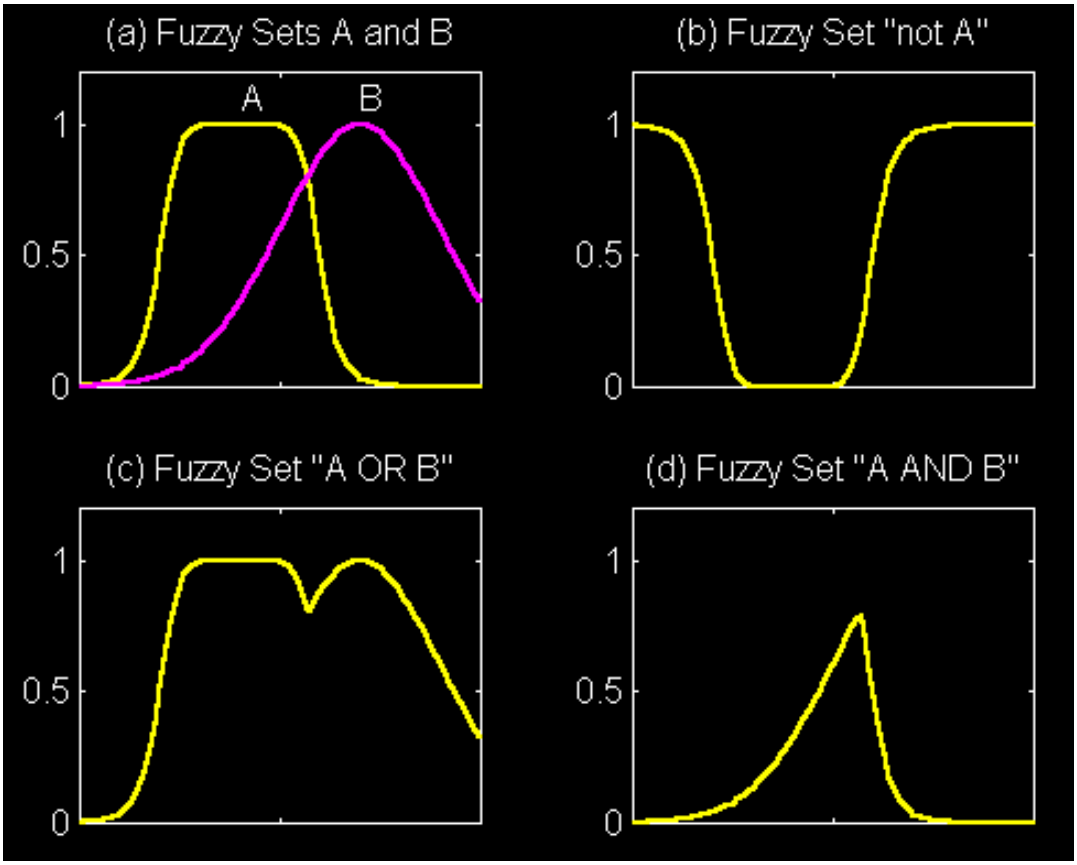
Set-Theoretic Operations

Subset: $A \subseteq B \Leftrightarrow \mu_A \leq \mu_B$

Complement: $\overline{A} = X - A \Leftrightarrow \mu_{\overline{A}}(x) = 1 - \mu_A(x)$

Union: $C = A \cup B \Leftrightarrow \mu_c(x) = \max(\mu_A(x), \mu_B(x)) = \mu_A(x) \vee \mu_B(x)$

Intersection: $C = A \cap B \Leftrightarrow \mu_c(x) = \min(\mu_A(x), \mu_B(x)) = \mu_A(x) \wedge \mu_B(x)$



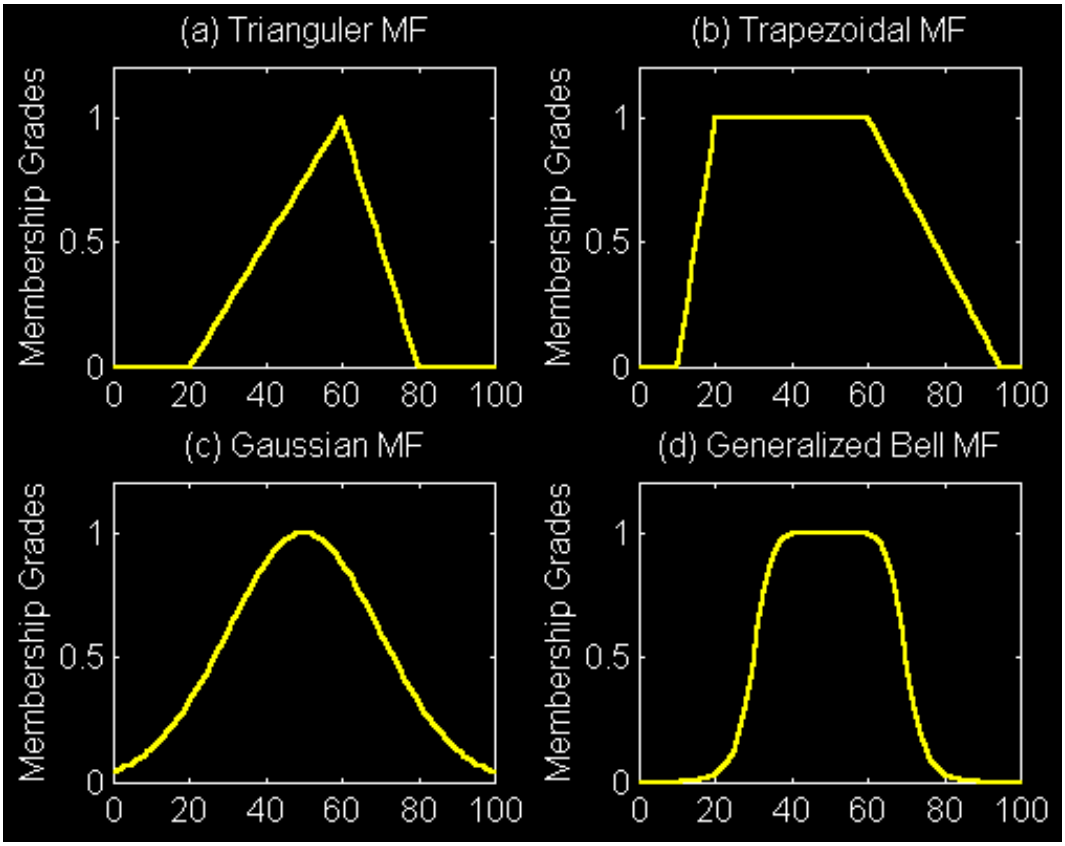
MF Formulation

Triangular MF: $trimf(x;a,b,c) = \max\left(\min\left(\frac{x-a}{b-a}, \frac{c-x}{c-b}\right), 0\right)$

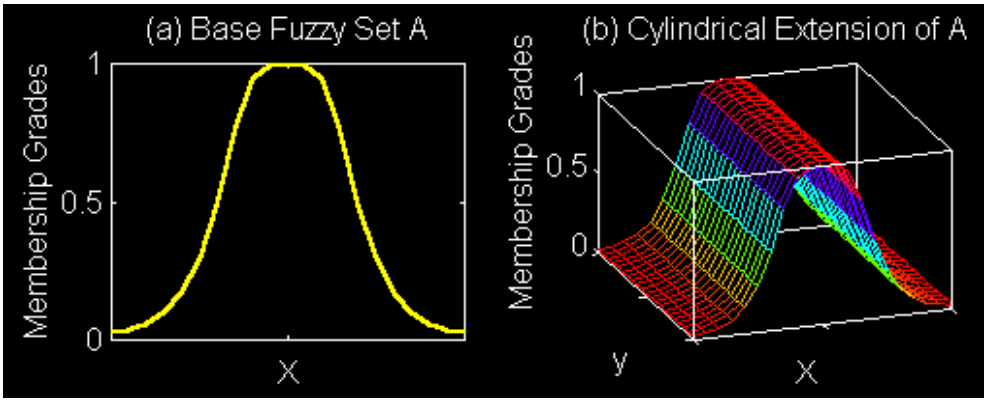
Trapezoidal MF: $trapmf(x;a,b,c,d) = \max\left(\min\left(\frac{x-a}{b-a}, 1, \frac{d-x}{d-c}\right), 0\right)$

Gaussian MF: $gaussmf(x;a,b,c) = e^{-\frac{1}{2}\left(\frac{x-c}{\sigma}\right)^2}$

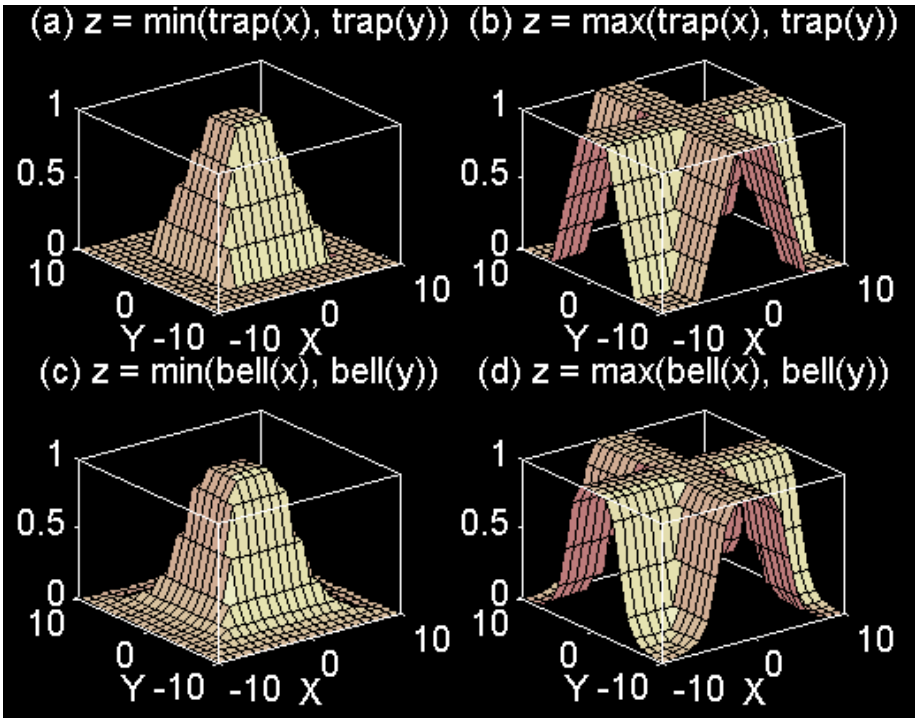
Generalized bell MF: $gbellmf(x;a,b,c) = \frac{1}{1+\left|\frac{x-c}{b}\right|^{2b}}$



Cylindrical Extension:



2D MFs:



Fuzzy Complement:

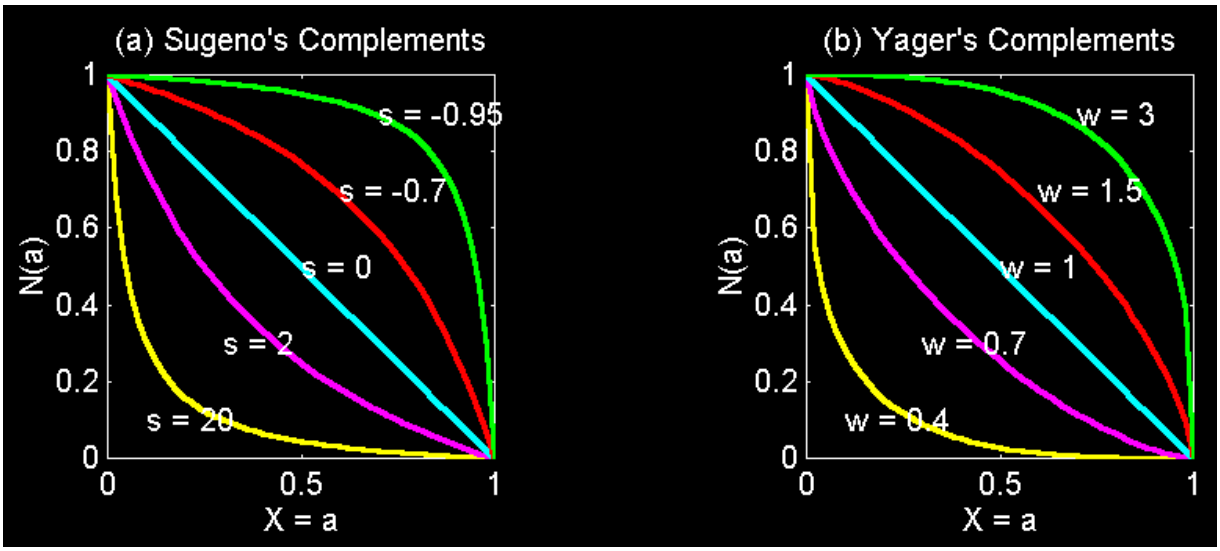
General requirements:

- Boundary: N(0)=1 and N(1) = 0
- Monotonicity: N(a) > N(b) if a < b
- Involution: N(N(a) = a

Two types of fuzzy complements:

Sugeno's complement: $N_s(a) = \frac{1-a}{1+sa}$

Yager's complement: $N_w(a) = (1-a^w)^{1/w}$



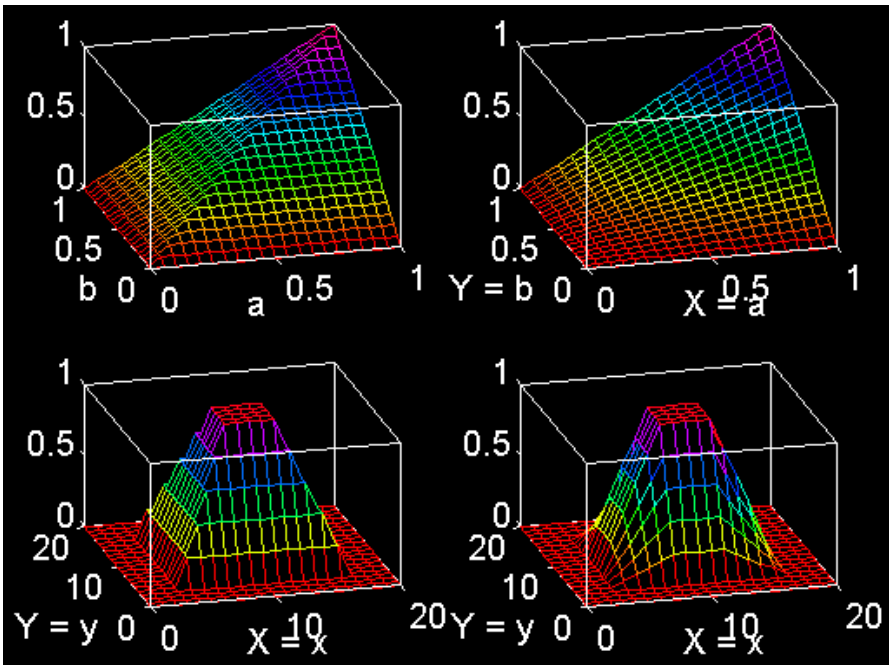
Fuzzy Intersection: T-norm

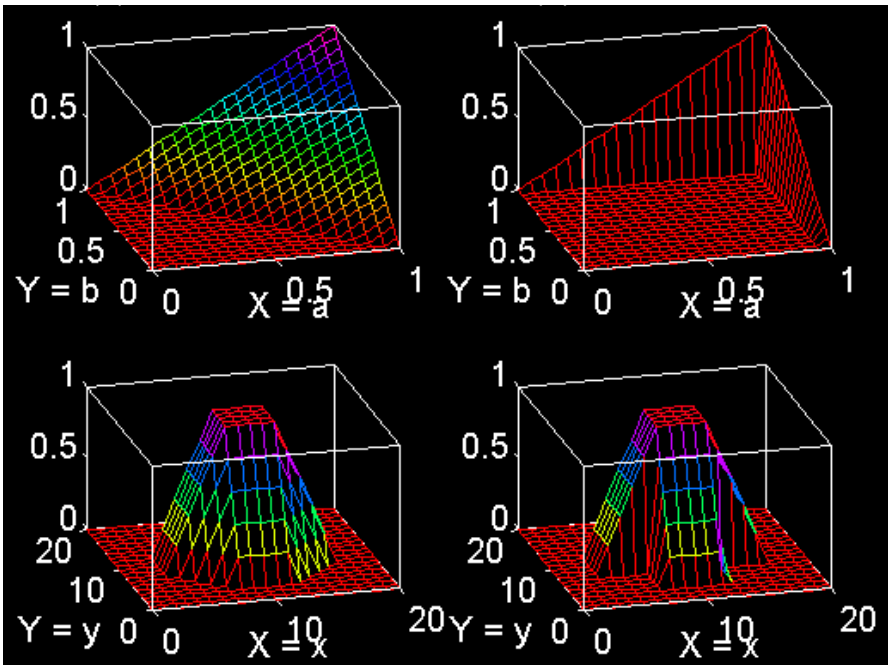
Basic requirements:

- Boundary: T(0, 0) = 0, T(a, 1) = T(1, a) = a
- Monotonicity: T(a, b) < T(c, d) if a < c and b < d
- Commutativity: T(a, b) = T(b, a)
- Associativity: T(a, T(b, c)) = T(T(a, b), c)

Four examples (page 37):

- Minimum: Tm(a, b)
- Algebraic product: Ta(a, b)
- Bounded product: Tb(a, b)
- Drastic product: Td(a, b)





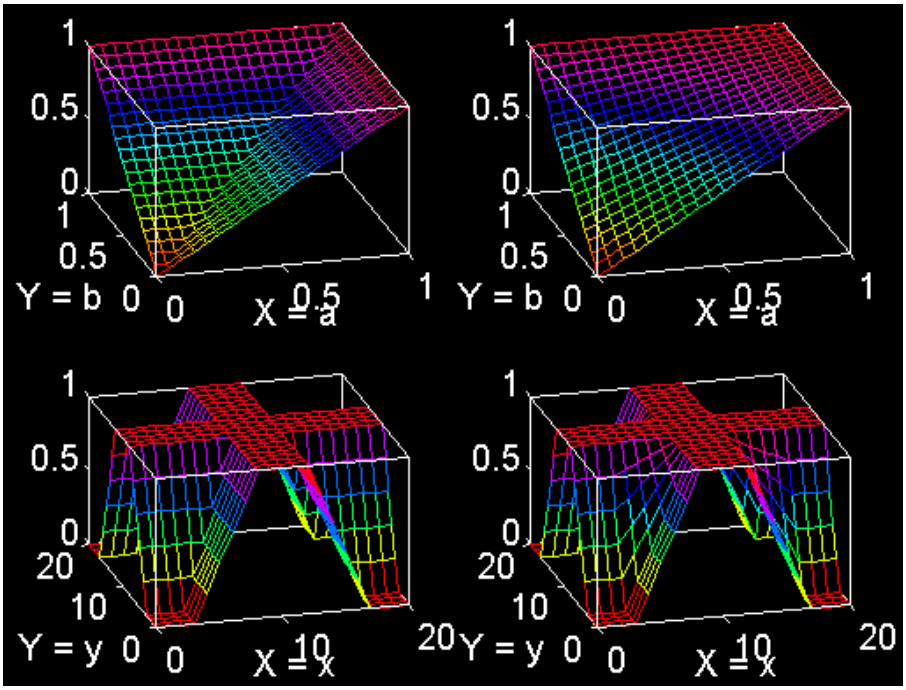
Fuzzy Union: T-conorm or S-norm

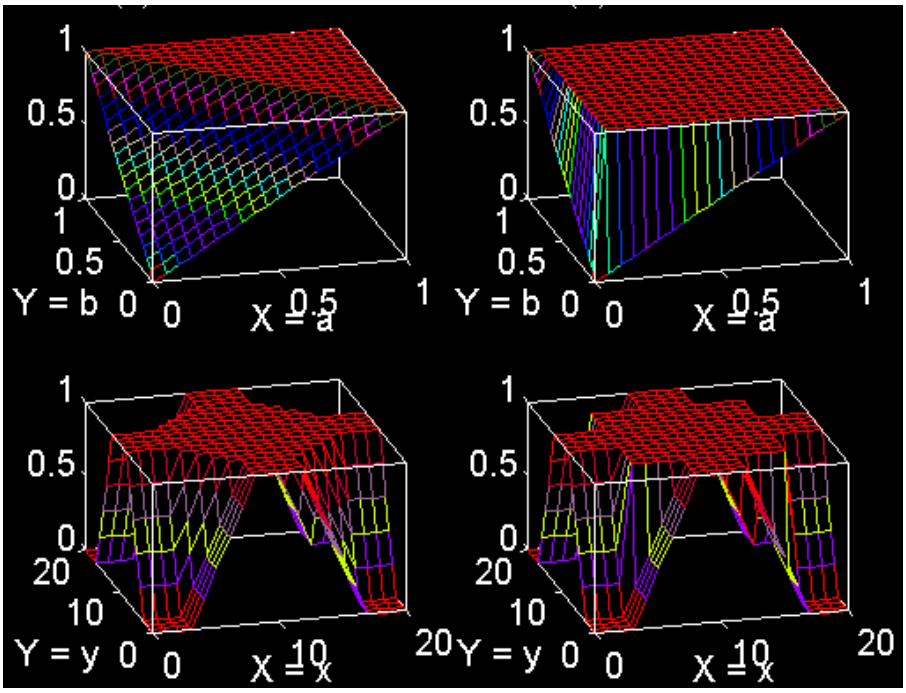
Basic requirements:

- Boundary: S(1, 1) = 1, S(a, 0) = S(0, a) = a
- Monotonicity: S(a, b) < S(c, d) if a < c and b < d
- Commutativity: S(a, b) = S(b, a)
- Associativity: S(a, S(b, c)) = S(S(a, b), c)

Four examples (page 38):

- Maximum: Sm(a, b)
- Algebraic sum: Sa(a, b)
- Bounded sum: Sb(a, b)
- Drastic sum: Sd(a, b)





Generalized DeMorgan's Law:

T-norms and T-conorms are duals which support the generalization of DeMorgan's law:

- T(a, b) = N(S(N(a), N(b)))
- S(a, b) = N(T(N(a), N(b)))

Parameterized T-norm and S-norm:

Parameterized T-norms and dual T-conorms have been proposed by several researchers:

- Yager
- Schweizer and Sklar
- Dubois and Prade
- Hamacher
- Frank
- Sugeno
- Dombi

Fuzzy Rules and Fuzzy Reasoning

It includes the followings.

- Extension principle
- Fuzzy relations
- Fuzzy if-then rules
- Compositional rule of inference
- Fuzzy reasoning

Extension Principle:

A is a fuzzy set on *X* : $A = \mu_A(x_1)/x_1 + \mu_A(x_2)/x_2 + \mu_A(x_n)/x_n$

The image of *A* under *f(.)* is a fuzzy set *B:* $B = \mu_A(x_1)/y_1 + \mu_A(x_2)/y_2 + \cdots + \mu_A(x_n)/y_n$

where *yi = f(xi)*, for *i = 1* to *n*.

If *f(.)* is a many-to-one mapping, then $\mu_B(y) = \max_{x=f^{-1}(y)} \mu_A(x)$

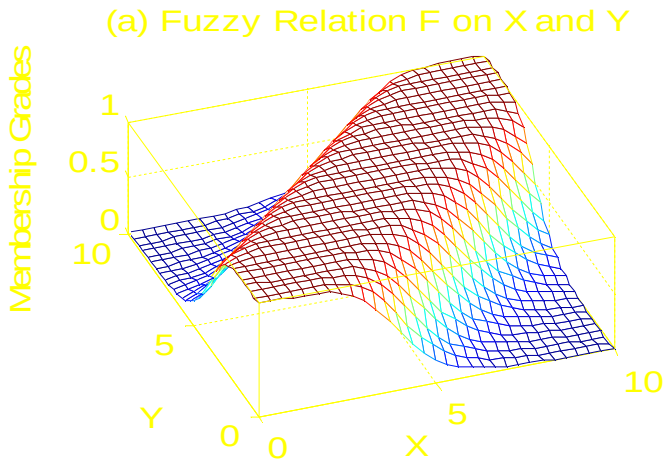
Fuzzy Relations:

A fuzzy relation *R* is a 2D MF: $R = \{((x,y), \mu_R(x,y)) \mid (x,y) \in X \times Y\}$

Examples:

- x is close to y (x and y are numbers)
- x depends on y (x and y are events)
- x and y look alike (x and y are persons or objects)
- If x is large, then y is small (x is an observed instrument reading and y is a corresponding control action which is inversely proportional to x.)

X is close to y is plotted as shown below.



Max-Min Composition:
The max-min composition of two fuzzy relations *R1* (defined on *X* and *Y*) and *R2* (defined on *Y* and *Z*):

$$\mu_{R_1 \circ R_2}(x,z) = \vee_y [\mu_{R_1}(x,y) \wedge \mu_{R_2}(y,z)]$$

- Associativity: $R \circ (S \circ T) = (R \circ S) \circ T$
- Distributivity over union: $R \circ (S \cup T) = (R \circ S) \cup (R \circ T)$
- Weak distributivity over intersection:
- Monotonicity: $S \subseteq T \Rightarrow (R \circ S) \subseteq (R \circ T)$

Max-Star Composition:

Max-product composition: $\mu_{R_1 \circ R_2}(x,z) = \vee_y [\mu_{R_1}(x,y)\mu_{R_2}(y,z)]$

In general, we have max * compositions: $\mu_{R_1 \circ R_2}(x,z) = \vee_y [\mu_{R_1}(x,y) * \mu_{R_2}(y,z)]$

where * is a T-norm operator.

Example of – Max * Compositions:

R_1: x is relevant to y

	y=α	y=β	y=γ	y=δ
x=1	0.1	0.3	0.5	0.7
x=2	0.4	0.2	0.8	0.9
x=3	0.6	0.8	0.3	0.2

R_2: y is relevant to z

	z=a	z=b
y=α	0.9	0.1
y=β	0.2	0.3
y=γ	0.5	0.6
y=δ	0.7	0.2

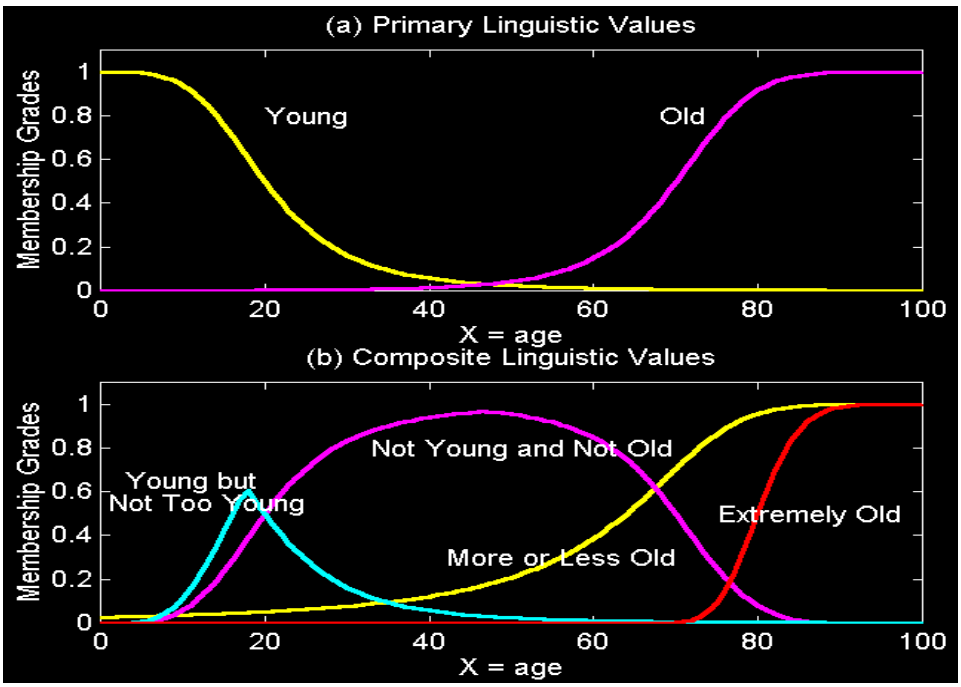
How relevant is x=2 to z=a?

$\mu_{R1 \circ R2}(2,a) = 0.7$ (by max-min composition)

$\mu_{R1 \circ R2}(2,a) = 0.63$ (max-product composition)

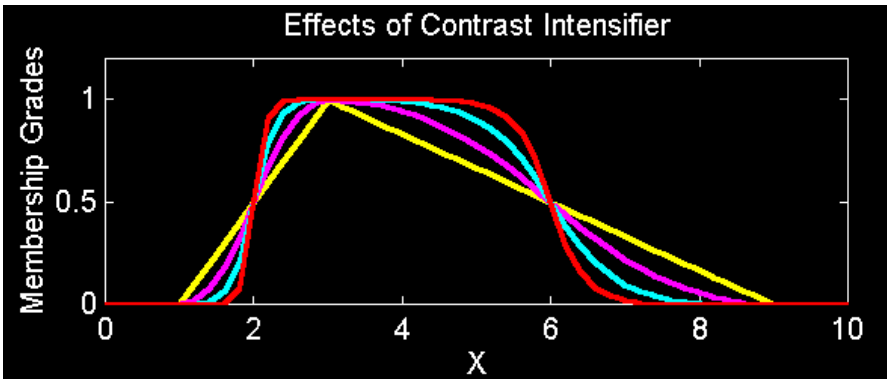
Linguistic Variables:

A numerical variable takes numerical values: *Age = 65*
A linguistic variables takes linguistic values: *Age is old*
A linguistic value is a fuzzy set.
All linguistic values form a term set (set of terms):
T(age) = {young, not young, very young, middle aged, not middle aged, old, not old, very old, more or less old, not very young and not very old, etc.}



Operations on Linguistic Values:
Concentration operation like very is defined as $CON(A) = A^2$.
Dilation operation like more or less is defined as $DIL(A) = A^{1/2}$.

Contrast intensification is defined as $$INT(A) = \begin{cases} 2A^2, & 0 \le \mu_A(x) \le 0.5 \\ \neg 2(\neg A)^2, & 0.5 \le \mu_A(x) \le 1 \end{cases}$$



Fuzzy If-Then Rules:
General format: If x is A then y is B
This is interpreted as a fuzzy set
Examples:

- If pressure is high, then volume is small.
- If the road is slippery, then driving is dangerous.
- If a tomato is red, then it is ripe.
- If the speed is high, then apply the brake a little.

Fuzzy If-Then Rules:
Two ways to interpret "If x is A then y is B"
A is coupled with B: (x is A) $\cap$ (y is B)
A entails B: (x is not A) $\cup$ (y is B)

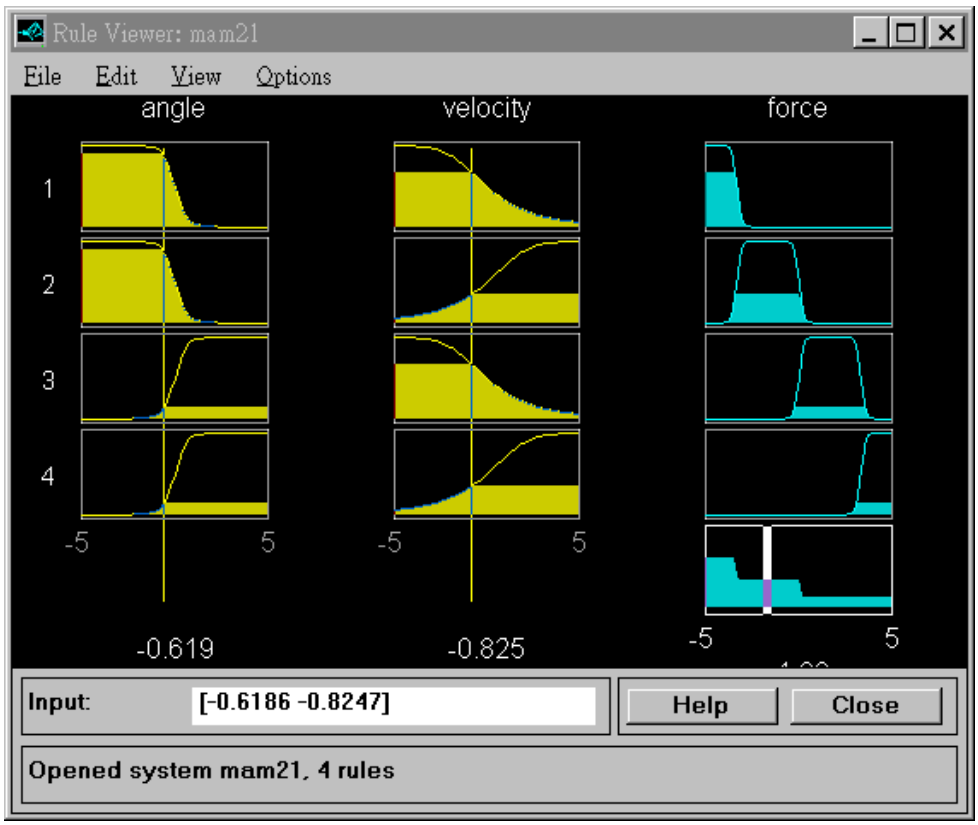
Example:
if (profession is athlete) then (fitness is high)
Coupling: Athletes, and only athletes, have high fitness.
The "if" statement (antecedent) is a necessary and sufficient condition.
Entailing: Athletes have high fitness, and non-athletes may or may not have high fitness.
The "if" statement (antecedent) is a sufficient but not necessary condition.

Fuzzy Reasoning:
Single rule with single antecedent Rule: if x is A then y is B
Premise: x is A', where A' is close to A
Conclusion: y is B'
Use max of intersection between A and A' to get B'

Single rule with multiple antecedents
Rule: if x is A and y is B then z is C
Premise: x is A' and y is B'
Conclusion: z is C'
Use min of (A ∩ A') and (B ∩ B') to get C'

Multiple rules with multiple antecedents
Rule 1: if x is A1 and y is B1 then z is C1
Rule 2: if x is A2 and y is B2 then z is C2
Premise: x is A' and y is B'
Conclusion: z is C'
Use previous slide to get C1' and C2'
Use max of C1' and C2' to get C' (next slide)

>> **ruleview mam21 (Matlab Fuzzy Logic Toolbox)**



Neural Network Representation

- An ANN is composed of processing elements called or ***neurons*** organized in different ways to form the network's structure.

Processing Elements

An ANN consists of neurons. Each of the neurons receives inputs, processes inputs and delivers a single output. A simplest ANN is a single neuron perceptron which is shown in Fig.2.

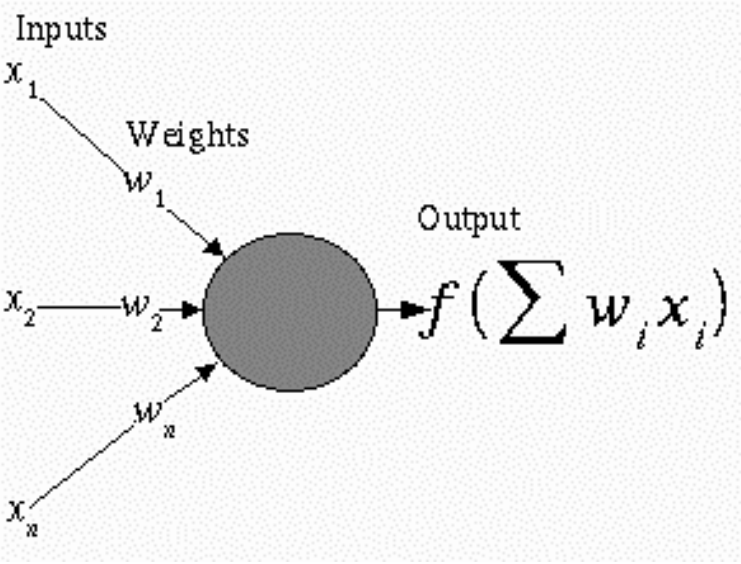


Fig. 2: A single neuron perceptron.

The input can be raw input data or the output of other perceptrons. The output can be the final result (e.g. 1 means yes, 0 means no) or it can be inputs to other perceptrons.

The network:

- Each ANN is composed of a collection of perceptrons grouped in layers. A typical structure is shown in Fig.3.
- Note the three layers of the shown ANN. They are input, intermediate (called the ***hidden layer***) and output layers.
- Several hidden layers can be placed between the input and output layers.

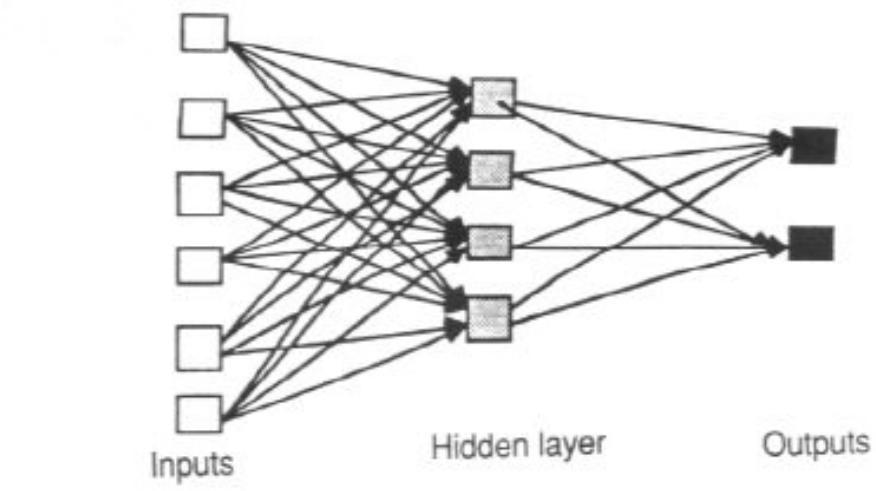


Fig. 3: A three layer ANN.

Perceptrons:

- A perceptron takes a vector of real-valued inputs, calculates a linear combination of these inputs, then outputs
  - a 1 if the result is greater than some threshold
  - –1 otherwise.
- Given real-valued inputs x_1 through x_n, the output $o(x_1, \ldots, x_n)$ computed by the perceptron is given as

$$o(x_1, \ldots, x_n) = 1 \quad \text{if } w_0 + w_1x_1 + \ldots + w_nx_n > 0$$
$$= -1 \quad \text{otherwise}$$

where w_i is a real-valued constant, or *weight.*

Notice the quantity $(-w_0)$ is a threshold that the weighted combination of inputs $w_1x_1 + \ldots + w_nx_n$ must surpass in order for perceptron to output a 1.

- To simplify notation, we imagine an additional constant input $x_0 = 1$, allowing us to write the above inequality as

$$\sum_{i=0}^{n} w_ix_i > 0$$

- Learning a perceptron involves choosing values for the weights $w_0, w_1, \ldots, w_n$.

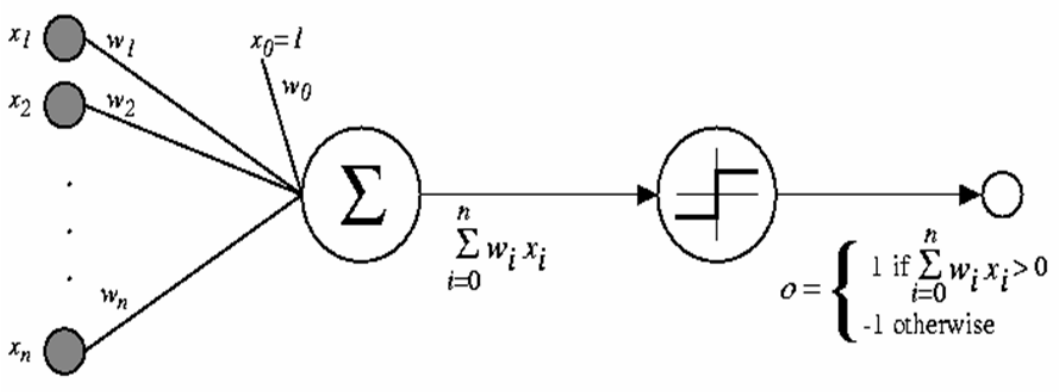


Fig. 4: An n input percetron connected to a thresh-holding function for binary output.

Representation Power of Perceptrons:

- We can view the perceptron as representing a hyperplane **decision surface** in the *n*-dimensional space of instances (i.e. points). The perceptron outputs a 1 for instances lying on one side of the hyperplane and outputs a –1 for instances lying on the other side, as in Figure 4. The equation for this decision hyperplane is

$$\vec{w} \cdot \vec{x} = 0$$

- Some sets of positive and negative examples cannot be separated by any hyperplane. Those that can be separated are called *linearly separated set of examples.*

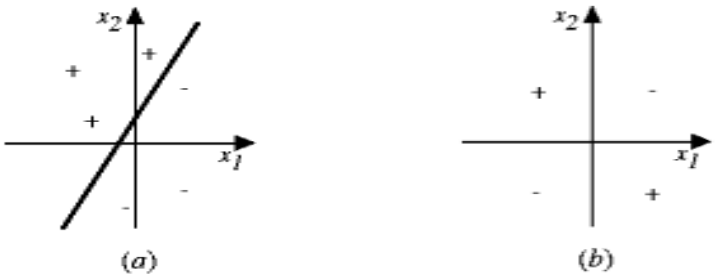
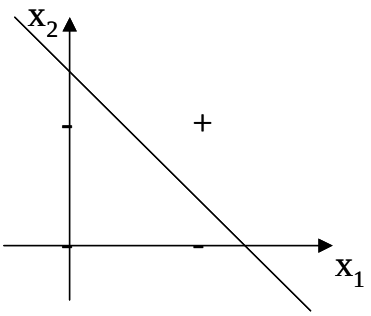


Fig.5: (a) Linearly separable classes (b) Not linearly separable classes

A single perceptron can be used to represent many boolean functions.
• AND function :
A decision hyper-plane described by $w_0 + w_1 x_1 + w_2 x_2 = 0$ with $w_0 = -0.8$, $w_1 = 0.5$ and $w_2 = 0.5$, that is, the hyper-plane $-0.8 + 0.5 x_1 + 0.5 x_2 = 0$ can represent Boolean AND function.



$$-0.8 + 0.5\,x_1 + 0.5\,x_2 = 0$$

<Training examples>

x_1	x_2	output
0	0	-1
0	1	-1
1	0	-1
1	1	1

<Test Results>

x_1	x_2	$\sum w_i x_i$	output
0	0	-0.8	-1
0	1	-0.3	-1
1	0	-0.3	-1
1	1	0.2	1

OR function:

Similarly a Decision hyper-plane $w_0 + w_1 x_1 + w_2 x_2 = 0$ with $w_0 = -0.3$, $w_1 = 0.5$ and $w_2 = 0.5$, that is, the hyper-plane $-0.3 + 0.5\,x_1 + 0.5\,x_2 = 0$ can represent an OR function.

<Training examples>

x_1	x_2	output
0	0	-1
0	1	1
1	0	1
1	1	1

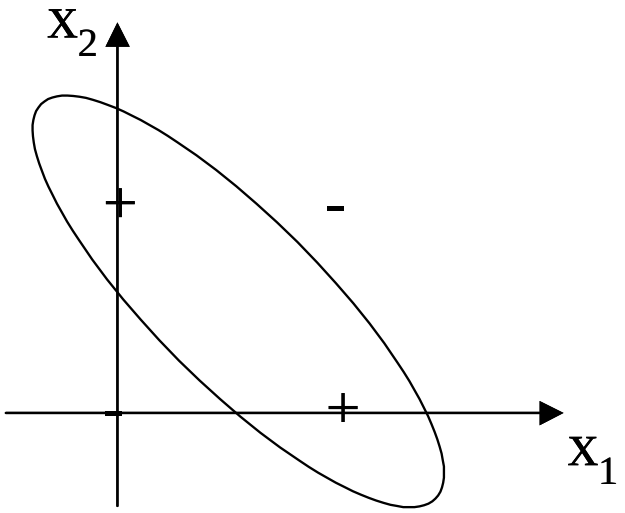
<Test Results>

x_1	x_2	$\sum w_i x_i$	output
0	0	-0.3	-1
0	1	0.2	-1
1	0	0.2	-1
1	1	0.7	1

XOR function:

It's impossible to implement the XOR function by a single perception. So a two-layer network of perceptrons can represent XOR function as per the equation

$$x_1 \oplus x_2 = x_1\bar{x}_2 + \bar{x}_1 x_2$$



Perceptron training rule:

Although we are interested in learning networks of many interconnected units, let us begin by understanding how to learn the weights for a single perceptron. Here learning is to determine a weight vector that causes the perceptron to produce the correct +1 or –1 for each of the given training examples. Several algorithms are known to solve this learning problem. Here we consider two: the perceptron rule and the delta rule. One way to learn an acceptable weight vector is to begin with random weights, then iteratively apply the perceptron to each training example, modifying the perceptron weights whenever it misclassifies an example. This process is repeated, iterating through the training examples as many as times needed until the perceptron classifies all training examples correctly. Weights are modified at each step according to the perceptron training rule, which revises the weight w_i associated with input x_i according to the rule.

$$w_i \leftarrow w_i + \Delta w_i$$

where $\Delta w_i = \eta(t - o)\, x_i$

Here: t is target output value for the current training example 'o' is perceptron output

η is small constant (e.g., 0.1) called learning rate

The role of the learning rate is to moderate the degree to which weights are changed at each step. It is usually set to some small value (e.g. 0.1) and is sometimes made to decrease as the number of weight-tuning iterations increases. We can prove that the algorithm will converge If training data is linearly separable and η sufficiently small.

If the data is not linearly separable, convergence is not assured.

Gradient Descent and the Delta Rule:

Although the perceptron rule finds a successful weight vector when the training examples are linearly separable, it can fail to converge if the examples are not linearly separately. A second training rule, called the delta rule, is designed to overcome this difficulty. The key idea of delta

rule: to use gradient descent to search the space of possible weight vector to find the weights that best fit the training examples. This rule is important because it provides the basis for the backpropagration algorithm, which can learn networks with many interconnected units. The delta training rule: considering the task of training an unthresholded perceptron, that is a linear unit, for which the output o is given by:

$$o = w_0 + w_1x_1 + \cdots + w_nx_n \quad (1)$$

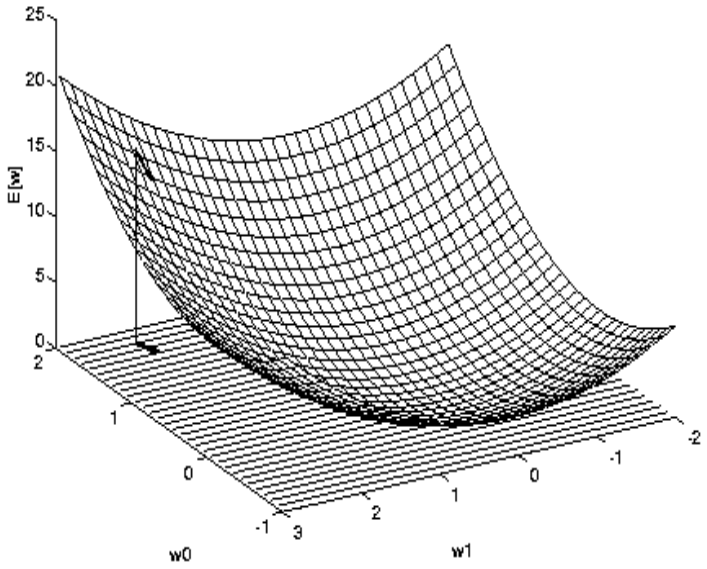
Thus, a linear unit corresponds to the first stage of a perceptron, without the threshold.

In order to derive a weight learning rule for linear units, let specify a measure for the training error of a weight vector, relative to the training examples. The Training Error can be computed as the following squared error

$$E[\vec{w}] = \frac{1}{2}\sum_{d \in D}(t_d - O_d)^2 \quad (2)$$

where D is set of training examples, t_d is the target output for the training example d and O_d is the output of the linear unit for the training example d. Here we characterize E as a function of weight vector because the linear unit output O depends on this weight vector.

Hypothesis Space: To understand the gradient descent algorithm, it is helpful to visualize the entire space of possible weight vectors and their associated E values, as illustrated in Figure 5.



Here the axes w_o, w_1 represents possible values for the two weights of a simple linear unit. The w_o,w_1 plane represents the entire hypothesis space. The vertical axis indicates the error E relative to some fixed set of training examples. The error surface shown in the figure summarizes the desirability of every weight vector in the hypothesis space. For linear units, this error surface must be parabolic with a single global minimum. And we desire a weight vector with this minimum.

Derivation of the Gradient Descent Rule: This vector derivative is called the *gradient* of E with respect to the vector $\langle w_0,\ldots,w_n\rangle$, written ∇E .

$$\nabla E[\vec{w}] = [\frac{dE}{dw_0}, \frac{dE}{dw_1}, \ldots\ldots\ldots \frac{dE}{dw_n}] \qquad (3)$$

Notice ∇E is itself a vector, whose components are the partial derivatives of E with respect to each of the w_i. When interpreted as a vector in weight space, the gradient specifies the direction that produces the steepest increase in E. The negative of this vector therefore gives the direction of steepest decrease. Since the gradient specifies the direction of steepest increase of E, the training rule for gradient descent is

$$\mathbf{w} \leftarrow \mathbf{w} + \Delta\mathbf{w}$$

where $\Delta\vec{w} = -\eta\nabla E[\vec{w}]$ (4)

Here η is a positive constant called the *learning rate*, which determines the step size in the gradient descent search. The negative sign is present because we want to move the weight vector in the direction that decreases E. This training rule can also be written in its component form

$$w_i \leftarrow w_i + \Delta w_i$$

where

$$\Delta w_i = -\eta \frac{dE}{dw_i} \qquad (5)$$

which makes it clear that steepest descent is achieved by altering each component w_i of weight vector in proportion to $\partial E/\partial w_i$. The vector of $\partial E/\partial w_i$ derivatives that form the gradient can be obtained by differentiating *E* from Equation (2), as

$$\frac{dE}{dw_i} = \frac{d}{2dw_i}\sum_{d\in D}(t_d - O_d)^2$$

$$= \frac{1}{2}\sum_{d\in D}\frac{d}{dw_i}(t_d - O_d)^2$$

$$= \frac{1}{2}\sum_{d\in D} 2(t_d - O_d)\frac{d}{dw_i}(t_d - O_d)$$

$$= \sum_{d\in D}(t_d - O_d)\frac{d}{dw_i}(t_d - \vec{w}.\vec{x}_d)$$

$$\frac{dE}{dw_i} = \sum_{d \in D} (t_d - O_d)(-\vec{x}_d) \qquad (6)$$

where x_{id} denotes the single input component x_i for the training example d. We now have an equation that gives $\partial E/\partial w_i$ in terms of the linear unit inputs x_{id}, output o_d and the target value t_d associated with the training example. Substituting Equation (6) into Equation (5) yields the weight update rule for gradient descent.

$$\Delta dw_i = \eta \sum_{d \in D} (t_d - O_d)(x_{id}) \qquad (7)$$

The **gradient descent** algorithm for training linear units is as follows: Pick an initial random weight vector. Apply the linear unit to all training examples, them compute Δw_i for each weight according to Equation (7). Update each weight w_i by adding Δw_i , them repeat the process. The algorithm is given in Figure 6.

Because the **error surface** contains only a single global minimum, this algorithm will converge to a weight vector with minimum error, regardless of whether the training examples are linearly separable, given a sufficiently small η is used. If η is too large, the **gradient descent** search runs the risk of overstepping the minimum in the error surface rather than settling into it. For this reason, one common modification to the algorithm is to **gradually reduce** the value of η as the number of gradient descent steps grows.

Stochastic Approximation to Gradient Descent: The key practical difficulties in applying gradient descent are:Converging to a local minimum can sometimes be quite slow (i.e., it can require many thousands of steps).If there are multiple local minima in the error surface, then there is no guarantee that the procedure will find the global minimum.One common variation on gradient descent intended to alleviate these difficulties is called *incremental gradient descent* (or *stochastic gradient descent*). The key differences between standard gradient descent and stochastic gradient descent are:

- In standard gradient descent, the error is summed over all examples before upgrading weights, whereas in stochastic gradient descent weights are updated upon examining each training example.
- The modified training rule is like the training example we update the weight according to

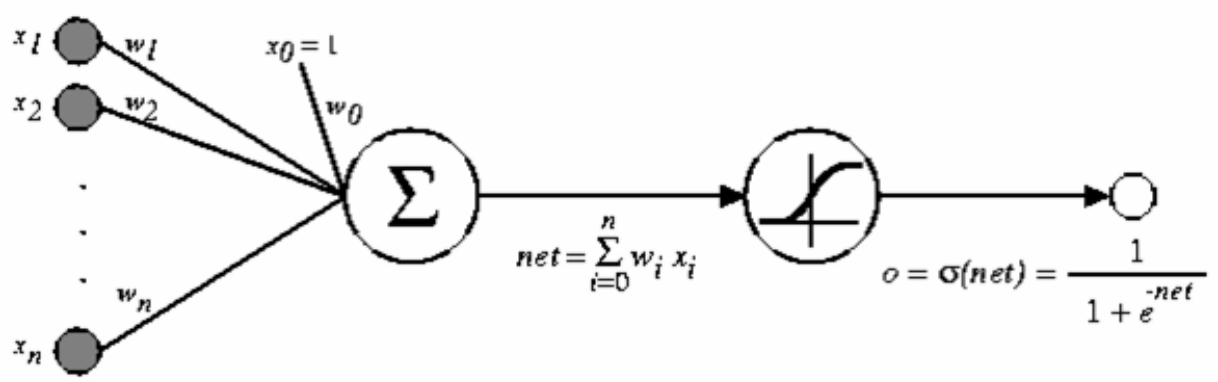
$$\Delta w_i = \eta(t - o)\, x_i \qquad (10)$$

Summing over multiple examples in standard gradient descent requires more computation per weight update step. On the other hand, because it uses the true gradient, standard gradient descent is often used with a larger step size per weight update than stochastic gradient descent

Stochastic gradient descent (i.e. incremental mode) can sometimes avoid falling into local minima because it uses the various gradient of *E* rather than overall gradient of *E* to guide its search. Both stochastic and standard gradient descent methods are commonly used in practice.

MULTILAYER NETWORKS AND THE BACKPROPOGATION ALGORITHM:

Single perceptrons can only express linear decision surfaces. In contrast, the kind of multilayer networks learned by the backpropagation algorithm are capaple of expressing a rich variety of **nonlinear** decision surfaces.This section discusses how to learn such multilayer networks using a gradient descent algorithm similar to that discussed in the previous section.



$\sigma(x)$ is the sigmoid function

$$\frac{1}{1 + e^{-x}}$$

Like the perceptron, the sigmoid unit first computes a linear combination of its inputs, then applies a threshold to the result. In the case of sigmoid unit, however, the threshold output is a continuous function of its input.The sigmoid function σ(x) is also called the **logistic function**. Interesting property:

$$\frac{d\sigma(x)}{dx} = \sigma(x)(1 - \sigma(x))$$

Output ranges between 0 and 1, increasing monotonically with its input.We can derive gradient decent rules to train

- One sigmoid unit

- *Multilayer networks* of sigmoid units ⇒ Backpropagation

The Backpropagation (BP)Algorithm:

The BP algorithm learns the weights for a multilayer network, given a network with a fixed set of units and interconnections. It employs a **gradient descent** to attempt to minimize the squared error between the network output values and the target values for these outputs.Because we are considering networks with multiple output units rather than single units as before, we begin by redefining *E* to sum the errors over all of the network output units

$$E(\mathbf{w}) = \tfrac{1}{2} \sum_{d \in D} \sum_{k \in outputs} (t_{kd} - o_{kd})^2 \qquad (13)$$

where *outputs* is the set of output units in the network, and t_{kd} and o_{kd} are the target and output values associated with the *k*th output unit and training example *d*.

The BP algorithm is presented in Figure 8. The algorithm applies to layered feedforward networks containing 2 layers of sigmoid units, with units at each layer connected to all units from the preceding layer. This is an *incremental gradient descent version* of Backpropagation. The notation is as follows:

x_{ij} denotes the input from node *i* to unit *j*, and w_{ij} denotes the corresponding weight.

δ_n denotes the error term associated with unit *n*. It plays a role analogous to the quantity (*t* – *o*) in our earlier discussion of the delta training rule

Initialize all weights to small random numbers. Until satisfied do

1. Input the training example to the network and compute the network output.
2. For each output unit k $\quad \delta_k \leftarrow O_k(1 - O_k)\,(t_k - O_k)$
3. For each hidden unit h $\quad \delta_h \leftarrow O_h(1 - O_h) \sum_{k \in outports} w_{h,k}\delta_h$
4. Update each network weight $w_{i,j}$

$$w_{i,j} \leftarrow w_{i,j} + \Delta w_{i,j}$$
$$where\ \Delta w_{i,j} = \eta\, \delta_j x_{i,j}$$

In the BP algorithm, step1 propagates the input forward through the network. And the steps 2, 3 and 4 propagates the errors backward through the network.The main loop of BP repeatedly iterates over the training examples. For each training example, it applies the ANN to the example, calculates the error of the network output for this example, computes the gradient with respect to the error on the example, then updates all weights in the network. This gradient

descent step is iterated until ANN performs acceptably well. A variety of termination conditions can be used to halt the procedure.

One may choose to halt after a fixed number of iterations through the loop, or once the error on the training examples falls below some threshold, or once the error on a separate validation set of examples meets some criteria.

Adding Momentum

Because BP is a widely used algorithm, many variations have been developed. The most common is to alter the weight-update rule in Step 4 in the algorithm by making the weight update on the *n*th iteration *depend partially on* the update that occurred during the (*n* -1)th iteration, as follows

$$\Delta w_{i-j}(n) = \eta \delta_j x_{i-j} + \alpha \Delta w_{i-j}(n-1)$$

Here $\Delta w_{i,j}(n)$ is the weight update performed during the *n*-th iteration through the main loop of the algorithm.

- *n*-th iteration update depend on (*n-1*)th iteration

- α: constant between 0 and 1 is called the *momentum.*

Role of momentum term:

- keep the ball rolling through small local minima in the error surface.

- Gradually increase the step size of the search in regions where the gradient is unchanging, thereby speeding convergence

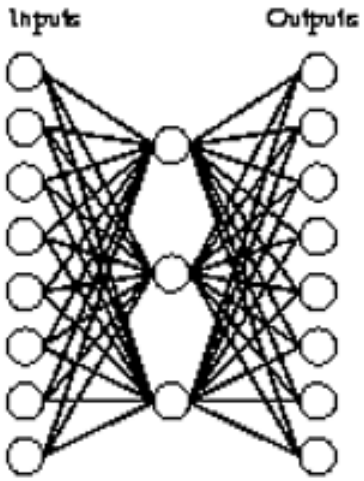
Expressive Capabilities of ANNs

- **Boolean functions:** Every boolean function can be represented by network with two layers of units where the number of hidden units required grows exponentially.
- **Continuous functions:** Every bounded continuous function can be approximated with arbitrarily small error, by network with two layers of units.
- **Arbitrary functions:** Any function can be approximated to arbitrary accuracy by a network with three layers of units .

Hidden layer representations

- Hidden layer representations
  - This 8x3x8 network was trained to learn the identity function.

- 8 training examples are used.
- After 5000 training iterations, the three hidden unit values encode the eight distinct inputs using the encoding shown on the right.



Input		Hidden Values				Output
10000000	→	.89	.04	.08	→	10000000
01000000	→	.01	.11	.88	→	01000000
00100000	→	.01	.97	.27	→	00100000
00010000	→	.99	.97	.71	→	00010000
00001000	→	.03	.05	.02	→	00001000
00000100	→	.22	.99	.99	→	00000100
00000010	→	.80	.01	.98	→	00000010
00000001	→	.60	.94	.01	→	00000001

Learning the 8x3x8 network

Most of the interesting weight changes occurred during the first 2500 iterations.

Figure 10.a The plot shows the sum of squared errors for each of the eight output units as the number of iterations increases. The sum of square errors for each output **decreases** as the procedure proceeds, more quickly for some output units and less quickly for others.

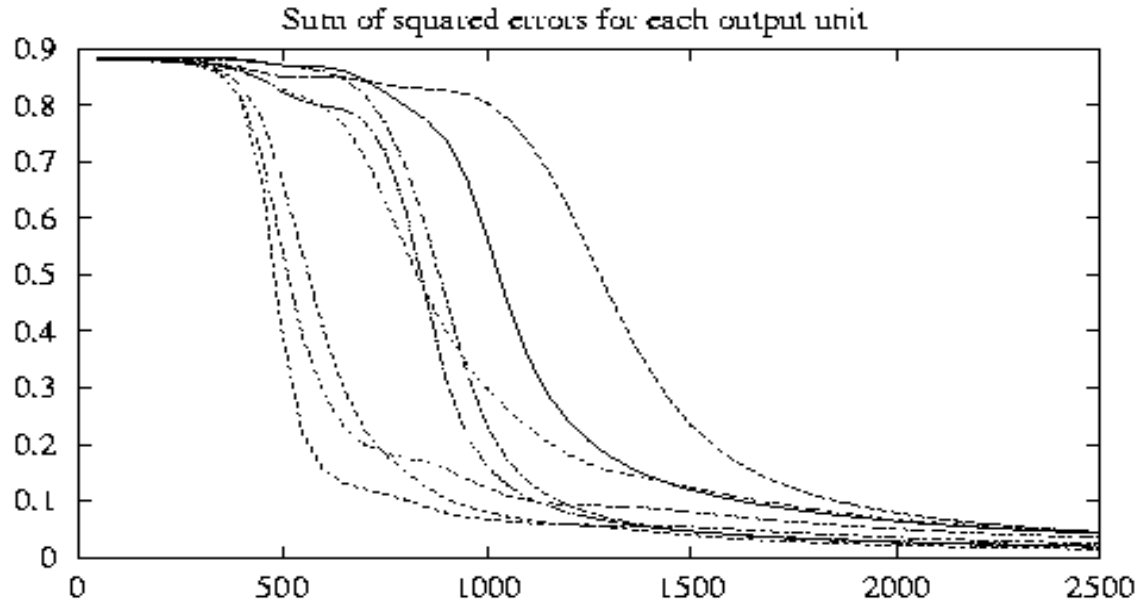


Fig-10.a

Figure 10.b Learning the 8 × 3 × 8 network. The plot shows the evolving hidden layer representation for the input string “010000000”. The network passes through a number of different encodings before **converging** to the final encoding

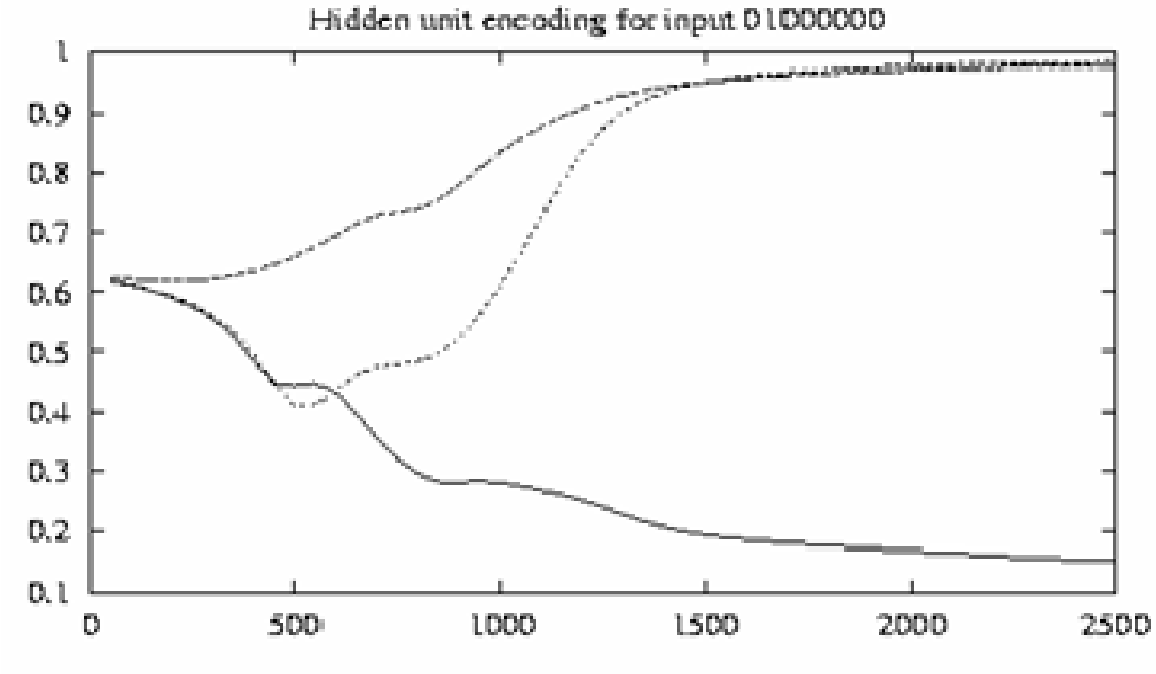
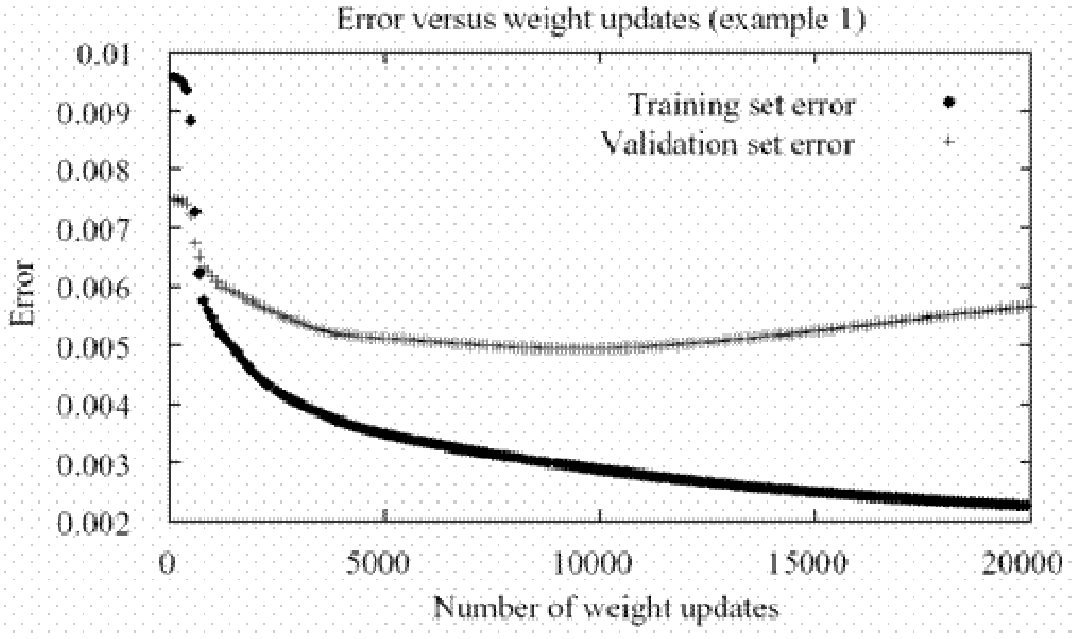
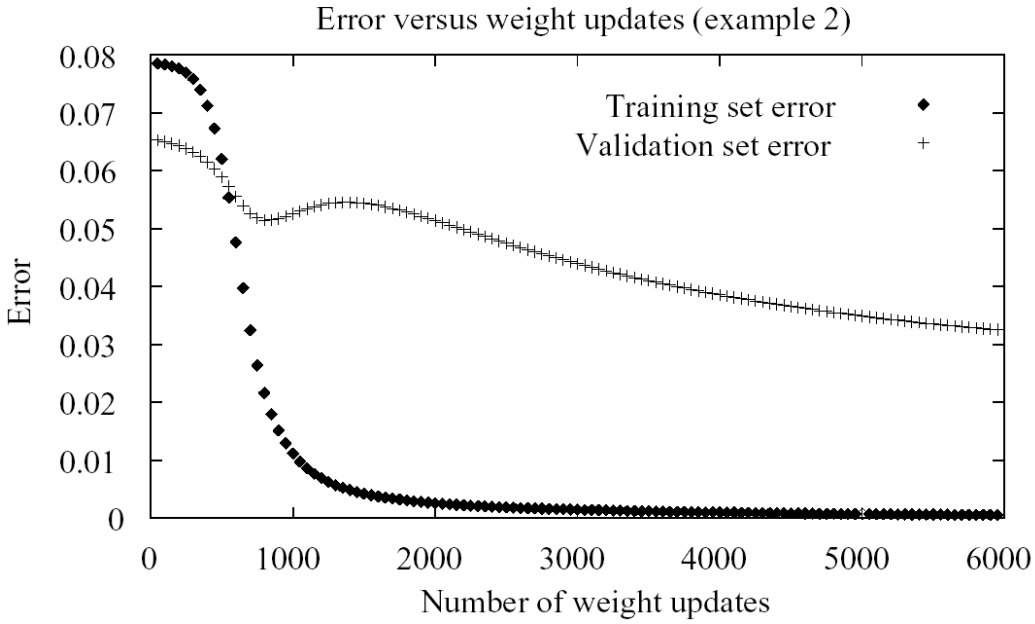


Fig 10.b

Generalization, Overfitting and Stopping Criterion

- Termination condition
  - Until the error E falls below some predetermined threshold
  - This is a poor strategy
- Overfitting problem
  - Backpropagation is susceptible to overfitting the training examples at the cost of decreasing **generalization accuracy** over other unseen examples.
  - To see the danger of minimizing the error over the training data, consider how the error *E* varies with the number of weight iteration.





The ***generalization accuracy*** measured over the training examples first decreases, then increases, even as the error over training examples continues to decrease. This occurs because the weights are being tuned to fit idiosyncrasies of the training examples that are not representative of the general distribution of examples.

Techniques to overcome overfitting problem

Weight decay: Decrease each weight by some small factor during each iteration. The motivation for this approach is to keep weight values small.

Cross-validation: a set of validation data in addition to the training data. The algorithm monitors the error w.r.t. this validation data while using the training set to drive the gradient descent search.

How much weight-tuning iteration should the algorithm perform? It should use the number of iterations that produces the lowest error over the validation set. Two copies of the weights are kept: one copy for training and a separate copy of the best weights thus far measured by their error over the validation set. Once the trained weights reach a higher error over the validation set than the stored weights, training is terminated and the stored weights are returned.

NEURAL NETWORK APPLICATION DEVELOPMENT

The development process for an ANN application has eight steps.

Step 1: (Data collection) The data to be used for the training and testing of the network are collected. Important considerations are that the particular problem is amenable to neural network solution and that adequate data exist and can be obtained.

Step 2: (Training and testing data separation) Trainning data must be identified, and a plan must be made for testing the performance of the network. The available data are divided into **training** and **testing** data sets. For a moderately sized data set, 80% of the data are randomly selected for training, 10% for testing, and 10% secondary testing.

Step 3: (Network architecture) A network architecture and a learning method are selected. Important considerations are the exact number of perceptrons and the number of layers.

Step 4: (Parameter tuning and weight initialization) There are parameters for tuning the network to the desired learning performance level. Part of this step is initialization of the network weights and parameters, followed by modification of the parameters as training performance feedback is received. Often, the initial values are important in determining the effectiveness and length of training.

Step 5: (Data transformation) Transforms the application data into the type and format required by the ANN.

Step 6: (Training) Training is conducted iteratively by presenting input and desired or known output data to the ANN. The ANN computes the outputs and adjusts the weights until the computed outputs are within an acceptable tolerance of the known outputs for the input cases.

Step 7: (Testing) Once the training has been completed, it is necessary to test the network. The **testing** examines the performance of the network using the derived weights by measuring the ability of the network to classify the testing data correctly. **Black-box testing** (comparing test results to historical results) is the primary approach for verifying that inputs produce the appropriate outputs.

Step 8: (Implementation) Now a stable set of weights are obtained.

Then the network can reproduce the desired output given inputs like those in the training set. The network is ready to use as a stand-alone system or as part of another software system where new input data will be presented to it and its output will be a recommended decision.

BENEFITS AND LIMITATIONS OF NEURAL NETWORKS

Benefits of ANNs

-*Usefulness for pattern recognition, classification, generalization, abstraction and interpretation of incomplete and noisy inputs*. (e.g. handwriting recognition, image recognition, voice and speech recognition, weather forecasting).

-*Robustness*. ANNs tend to be more robust than their conventional counterparts. They have the ability to cope with incomplete or fuzzy data. ANNs can be very tolerant of faults if properly implemented.

-*Fast processing speed*. Because they consist of a large number of massively interconnected processing units, all operating in parallel on the same problem, ANNs can potentially operate at considerable speed (when implemented on parallel processors).

- *Flexibility and ease of maintenance*. ANNs are very flexible in adapting their behavior to new and changing environments. They are also easier to maintain, with some having the ability to learn from experience to improve their own performance.

Limitations of ANNs

ANNs do not produce an *explicit model* even though new cases can be fed into it and new results obtained.

- *ANNs lack explanation capabilities*. Justifications for results is difficult to obtain because the connection weights usually do not have obvious interpretations.
- *Providing some human characteristics to problem solving* that are difficult to simulate using the logical, analytical techniques of expert systems and standard software technologies. (e.g. financial applications).

SOME ANN APPLICATIONS

- Tax form processing to identify tax fraud
- Enhancing auditing by finding irregularities
- Bankruptcy prediction
- Customer credit scoring

- Loan approvals
- Credit card approval and fraud detection
- Financial prediction
- Energy forecasting
- Computer access security (intrusion detection and classification of attacks)
- Fraud detection in mobile telecommunication networks

Recent improvements

Computational devices have been created in CMOS, for both biophysical simulation and neuromorphic computing. More recent efforts show promise for creating nano-devices for very large scale principal components analyses and convolution. If successful, these efforts could usher in a new era of neural computing that is a step beyond digital computing, because it depends on learning rather than programming and because it is fundamentally analog rather than digital even though the first instantiations may in fact be with CMOS digital devices.

Between 2009 and 2012, the recurrent neural networks and deep feedforward neural networks developed in the research group of Jürgen Schmidhuber at the Swiss AI Lab IDSIA have won eight international competitions in pattern recognition and machine learning. For example, multi-dimensional long short term memory (LSTM) won three competitions in connected handwriting recognition at the 2009 International Conference on Document Analysis and Recognition (ICDAR), without any prior knowledge about the three different languages to be learned.

Variants of the back-propagation algorithm as well as unsupervised methods by Geoff Hinton and colleagues at the University of Toront can be used to train deep, highly nonlinear neural architectures similar to the 1980 Neocognitron by Kunihiko Fukushima,[17] and the "standard architecture of vision",[18] inspired by the simple and complex cells identified by David H. Hubel and Torsten Wiesel in the primary visual cortex.

Deep learning feedforward networks, such as convolutional neural networks, alternate convolutional layers and max-pooling layers, topped by several pure classification layers. Fast GPU-based implementations of this approach have won several pattern recognition contests, including the IJCNN 2011 Traffic Sign Recognition Competition and the ISBI 2012 Segmentation of Neuronal Structures in Electron Microscopy Stacks challenge. Such neural

networks also were the first artificial pattern recognizers to achieve human-competitive or even superhuman performance on benchmarks such as traffic sign recognition (IJCNN 2012), or the MNIST handwritten digits problem of Yann LeCun and colleagues at NYU.

Successes in pattern recognition contests since 2009

Between 2009 and 2012, the recurrent neural networks and deep feedforward neural networks developed in the research group of Jürgen Schmidhuber at the Swiss AI Lab IDSIA have won eight international competitions in pattern recognition and machine learning. For example, the bi-directional and multi-dimensional long short term memory (LSTM) of Alex Graves et al. won three competitions in connected handwriting recognition at the 2009 International Conference on Document Analysis and Recognition (ICDAR), without any prior knowledge about the three different languages to be learned. Fast GPU-based implementations of this approach by Dan Ciresan and colleagues at IDSIA have won several pattern recognition contests, including the IJCNN 2011 Traffic Sign Recognition Competition, the ISBI 2012 Segmentation of Neuronal Structures in Electron Microscopy Stacks challenge, and others. Their neural networks also were the first artificial pattern recognizers to achieve human-competitive or even superhuman performance on important benchmarks such as traffic sign recognition (IJCNN 2012), or the MNIST handwritten digits problem of Yann LeCun at NYU. Deep, highly nonlinear neural architectures similar to the 1980 neocognitron by Kunihiko Fukushima and the "standard architecture of vision" can also be pre-trained by unsupervised methods of Geoff Hinton's lab at University of Toronto. A team from this lab won a 2012 contest sponsored by Merck to design software to help find molecules that might lead to new drugs.

Models

Neural network models in artificial intelligence are usually referred to as artificial neural networks (ANNs); these are essentially simple mathematical models defining a function $f: X \to Y$ or a distribution over X or both X and Y, but sometimes models are also intimately associated with a particular learning algorithm or learning rule. A common use of the phrase ANN model really means the definition of a *class* of such functions (where members of the class are obtained by varying parameters, connection weights, or specifics of the architecture such as the number of neurons or their connectivity).

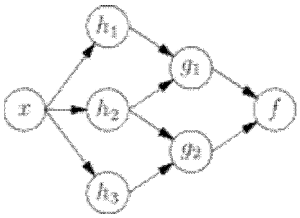
Network function

The word *network* in the term 'artificial neural network' refers to the inter–connections between the neurons in the different layers of each system. An example system has three layers. The first layer has input neurons which send data via synapses to the second layer of neurons, and then via more synapses to the third layer of output neurons. More complex systems will have more layers of neurons with some having increased layers of input neurons and output neurons. The synapses store parameters called "weights" that manipulate the data in the calculations.

An ANN is typically defined by three types of parameters:

1. The interconnection pattern between the different layers of neurons
2. The learning process for updating the weights of the interconnections
3. The activation function that converts a neuron's weighted input to its output activation.

Mathematically, a neuron's network function $f(x)$ is defined as a composition of other functions $g_i(x)$, which can further be defined as a composition of other functions. This can be conveniently represented as a network structure, with arrows depicting the dependencies between variables. A widely used type of composition is the *nonlinear weighted sum*, where $f(x)=K\left(\sum_i w_i g_i(x)\right)$, where K (commonly referred to as the activation function) is some predefined function, such as the hyperbolic tangent. It will be convenient for the following to refer to a collection of functions g_i as simply a vector $g=(g_1,g_2,\ldots,g_n)$.



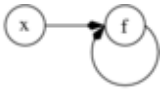
ANN dependency graph

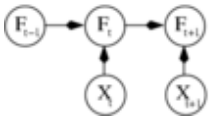
This figure depicts such a decomposition of f, with dependencies between variables indicated by arrows. These can be interpreted in two ways.

The first view is the functional view: the input x is transformed into a 3-dimensional vector h, which is then transformed into a 2-dimensional vector g, which is finally transformed into f. This view is most commonly encountered in the context of optimization.

The second view is the probabilistic view: the random variable $F=f(G)$ depends upon the random variable $G=g(H)$, which depends upon $H=h(X)$, which depends upon the random variable X. This view is most commonly encountered in the context of graphical models.

The two views are largely equivalent. In either case, for this particular network architecture, the components of individual layers are independent of each other (e.g., the components of g are independent of each other given their input h). This naturally enables a degree of parallelism in the implementation.





Two separate depictions of the recurrent ANN dependency graph

Networks such as the previous one are commonly called feedforward, because their graph is a directed acyclic graph. Networks with cycles are commonly called recurrent. Such networks are commonly depicted in the manner shown at the top of the figure, where f is shown as being dependent upon itself. However, an implied temporal dependence is not shown.

Learning

What has attracted the most interest in neural networks is the possibility of *learning*. Given a specific *task* to solve, and a *class* of functions F, learning means using a set of *observations* to find $f^* \in F$ which solves the task in some *optimal* sense.

This entails defining a cost function $C: F \to \mathbb{R}$ such that, for the optimal solution f^*, $C(f^*) \leq C(f) \forall f \in F$ – i.e., no solution has a cost less than the cost of the optimal solution (see Mathematical optimization).

The cost function C is an important concept in learning, as it is a measure of how far away a particular solution is from an optimal solution to the problem to be solved. Learning algorithms search through the solution space to find a function that has the smallest possible cost.

For applications where the solution is dependent on some data, the cost must necessarily be a *function of the observations*, otherwise we would not be modelling anything related to the data. It is frequently defined as a statistic to which only approximations can be made. As a simple example, consider the problem of finding the model f, which minimizes $C = E\left[(f(x) - y)^2\right]$, for data pairs (x, y) drawn from some distribution $\mathcal{D}$. In practical situations we would only have N samples from $\mathcal{D}$ and thus, for the above example, we would only minimize $\hat{C} = \frac{1}{N} \sum_{i=1}^{N} (f(x_i) - y_i)^2$. Thus, the cost is minimized over a sample of the data rather than the entire data set.

When $N \to \infty$ some form of online machine learning must be used, where the cost is partially minimized as each new example is seen. While online machine learning is often used when $\mathcal{D}$ is fixed, it is most useful in the case where the distribution changes slowly over time. In neural network methods, some form of online machine learning is frequently used for finite datasets.

See also: Mathematical optimization, Estimation theory and Machine learning

Choosing a cost function

While it is possible to define some arbitrary ad hoc cost function, frequently a particular cost will be used, either because it has desirable properties (such as convexity) or because it arises naturally from a particular formulation of the problem (e.g., in a probabilistic formulation the posterior probability of the model can be used as an inverse cost). Ultimately, the cost function will depend on the desired task. An overview of the three main categories of learning tasks is provided below:

Learning paradigms

There are three major learning paradigms, each corresponding to a particular abstract learning task. These are supervised learning, unsupervised learning and reinforcement learning.

Supervised learning

In supervised learning, we are given a set of example pairs $(x, y), x \in X, y \in Y$ and the aim is to find a function $f: X \to Y$ in the allowed class of functions that matches the examples. In other words, we wish to *infer* the mapping implied by the data; the cost function is related to the mismatch between our mapping and the data and it implicitly contains prior knowledge about the problem domain.

A commonly used cost is the mean-squared error, which tries to minimize the average squared error between the network's output, f(x), and the target value y over all the example pairs. When one tries to minimize this cost using gradient descent for the class of neural networks called multilayer perceptrons, one obtains the common and well-known backpropagation algorithm for training neural networks.

Tasks that fall within the paradigm of supervised learning are pattern recognition (also known as classification) and regression (also known as function approximation). The supervised learning paradigm is also applicable to sequential data (e.g., for speech and gesture recognition). This can be thought of as learning with a "teacher," in the form of a function that provides continuous feedback on the quality of solutions obtained thus far.

Unsupervised learning

In unsupervised learning, some data x is given and the cost function to be minimized, that can be any function of the data x and the network's output, f.

The cost function is dependent on the task (what we are trying to model) and our *a priori* assumptions (the implicit properties of our model, its parameters and the observed variables).

As a trivial example, consider the model $f(x)=a$ where a is a constant and the cost $C=E[(x-f(x))^2]$. Minimizing this cost will give us a value of a that is equal to the mean of the data. The cost function can be much more complicated. Its form depends on the application: for example, in compression it could be related to the mutual information between x and $f(x)$, whereas in statistical modeling, it could be related to the posterior probability of the model given the data. (Note that in both of those examples those quantities would be maximized rather than minimized).

Tasks that fall within the paradigm of unsupervised learning are in general estimation problems; the applications include clustering, the estimation of statistical distributions, compression and filtering.

Reinforcement learning

In reinforcement learning, data x are usually not given, but generated by an agent's interactions with the environment. At each point in time t, the agent performs an action y_t and the environment generates an observation x_t and an instantaneous cost c_t, according to some (usually unknown) dynamics. The aim is to discover a *policy* for selecting actions that minimizes some measure of a long-term cost; i.e., the expected cumulative cost. The environment's dynamics and the long-term cost for each policy are usually unknown, but can be estimated.

More formally the environment is modelled as a Markov decision process (MDP) with states $s_1,...,s_n \in S$ and actions $a_1,...,a_m \in A$ with the following probability distributions: the instantaneous cost distribution $P(c_t|s_t)$, the observation distribution $P(x_t|s_t)$ and the transition $P(s_{t+1}|s_t,a_t)$, while a policy is defined as conditional distribution over actions given the observations. Taken together, the two then define a Markov chain (MC). The aim is to discover the policy that minimizes the cost; i.e., the MC for which the cost is minimal.

ANNs are frequently used in reinforcement learning as part of the overall algorithm. Dynamic programming has been coupled with ANNs (Neuro dynamic programming) by Bertsekas and Tsitsiklis and applied to multi-dimensional nonlinear problems such as those involved in vehicle routing, natural resources management or medicine because of the ability of ANNs to mitigate losses of accuracy even when reducing the discretization grid density for numerically approximating the solution of the original control problems.

Tasks that fall within the paradigm of reinforcement learning are control problems, games and other sequential decision making tasks.
See also: dynamic programming and stochastic control

Learning algorithms

Training a neural network model essentially means selecting one model from the set of allowed models (or, in a Bayesian framework, determining a distribution over the set of allowed models) that minimizes the cost criterion. There are numerous algorithms available for training neural network models; most of them can be viewed as a straightforward application of optimization theory and statistical estimation.
Most of the algorithms used in training artificial neural networks employ some form of gradient descent, using backpropagation to compute the actual gradients. This is done by simply taking the derivative of the cost function with respect to the network parameters and then changing those parameters in a gradient-related direction.
Evolutionary methods, gene expression programming, simulated annealing, expectation-maximization, non-parametric methods and particle swarm optimization are some commonly used methods for training neural networks.

Employing artificial neural networks

Perhaps the greatest advantage of ANNs is their ability to be used as an arbitrary function approximation mechanism that 'learns' from observed data. However, using them is not so straightforward, and a relatively good understanding of the underlying theory is essential.

- Choice of model: This will depend on the data representation and the application. Overly complex models tend to lead to problems with learning.
- Learning algorithm: There are numerous trade-offs between learning algorithms. Almost any algorithm will work well with the *correct hyperparameters* for training on a particular fixed data set. However, selecting and tuning an algorithm for training on unseen data requires a significant amount of experimentation.
- Robustness: If the model, cost function and learning algorithm are selected appropriately the resulting ANN can be extremely robust.

With the correct implementation, ANNs can be used naturally in online learning and large data set applications. Their simple implementation and the existence of mostly local dependencies exhibited in the structure allows for fast, parallel implementations in hardware.

Applications

The utility of artificial neural network models lies in the fact that they can be used to infer a function from observations. This is particularly useful in applications where the complexity of the data or task makes the design of such a function by hand impractical.

Real-life applications

The tasks artificial neural networks are applied to tend to fall within the following broad categories:

- Function approximation, or regression analysis, including time series prediction, fitness approximation and modeling.

- Classification, including pattern and sequence recognition, novelty detection and sequential decision making.
- Data processing, including filtering, clustering, blind source separation and compression.
- Robotics, including directing manipulators, prosthesis.
- Control, including Computer numerical control.

Application areas include the system identification and control (vehicle control, process control, natural resources management), quantum chemistry, game-playing and decision making (backgammon, chess, poker), pattern recognition (radar systems, face identification, object recognition and more), sequence recognition (gesture, speech, handwritten text recognition), medical diagnosis, financial applications (e.g. automated trading systems), data mining (or knowledge discovery in databases, "KDD"), visualization and e-mail spam filtering.

Artificial neural networks have also been used to diagnose several cancers. An ANN based hybrid lung cancer detection system named HLND improves the accuracy of diagnosis and the speed of lung cancer radiology. These networks have also been used to diagnose prostate cancer. The diagnoses can be used to make specific models taken from a large group of patients compared to information of one given patient. The models do not depend on assumptions about correlations of different variables. Colorectal cancer has also been predicted using the neural networks. Neural networks could predict the outcome for a patient with colorectal cancer with more accuracy than the current clinical methods. After training, the networks could predict multiple patient outcomes from unrelated institutions.

Neural networks and neuroscience

Theoretical and computational neuroscience is the field concerned with the theoretical analysis and the computational modeling of biological neural systems. Since neural systems are intimately related to cognitive processes and behavior, the field is closely related to cognitive and behavioral modeling.

The aim of the field is to create models of biological neural systems in order to understand how biological systems work. To gain this understanding, neuroscientists strive to make a link between observed biological processes (data), biologically plausible mechanisms for neural processing and learning (biological neural network models) and theory (statistical learning theory and information theory).

Types of models

Many models are used in the field, defined at different levels of abstraction and modeling different aspects of neural systems. They range from models of the short-term behavior of individual neurons, models of how the dynamics of neural circuitry arise from interactions between individual neurons and finally to models of how behavior can arise from abstract neural modules that represent complete subsystems. These include models of the long-term, and short-term plasticity, of neural systems and their relations to learning and memory from the individual neuron to the system level.

Neural network software

Main article: Neural network software

Neural network software is used to simulate, research, develop and apply artificial neural networks, biological neural networks and, in some cases, a wider array of adaptive systems.

Types of artificial neural networks

Main article: Types of artificial neural networks

Artificial neural network types vary from those with only one or two layers of single direction logic, to complicated multi–input many directional feedback loops and layers. On the whole, these systems use algorithms in their programming to determine control and organization of their functions. Most systems use "weights" to change the parameters of the throughput and the varying connections to the neurons. Artificial neural networks can be autonomous and learn by input from outside "teachers" or even self-teaching from written-in rules.

Theoretical properties

Computational power

The multi-layer perceptron (MLP) is a universal function approximator, as proven by the universal approximation theorem. However, the proof is not constructive regarding the number of neurons required or the settings of the weights.

Work by Hava Siegelmann and Eduardo D. Sontag has provided a proof that a specific recurrent architecture with rational valued weights (as opposed to full precision real number-valued weights) has the full power of a Universal Turing Machine using a finite number of neurons and standard linear connections. They have further shown that the use of irrational values for weights results in a machine with super-Turing power.

Capacity

Artificial neural network models have a property called 'capacity', which roughly corresponds to their ability to model any given function. It is related to the amount of information that can be stored in the network and to the notion of complexity.

Convergence

Nothing can be said in general about convergence since it depends on a number of factors. Firstly, there may exist many local minima. This depends on the cost function and the model. Secondly, the optimization method used might not be guaranteed to converge when far away from a local minimum. Thirdly, for a very large amount of data or parameters, some methods become impractical. In general, it has been found that theoretical guarantees regarding convergence are an unreliable guide to practical application.

Generalization and statistics

In applications where the goal is to create a system that generalizes well in unseen examples, the problem of over-training has emerged. This arises in convoluted or over-specified systems when the capacity of the network significantly exceeds the needed free parameters. There are two schools of thought for avoiding this problem: The first is to use cross-validation and similar techniques to check for the presence of overtraining and optimally select hyperparameters such as to minimize the generalization error. The second is to use some form of *regularization*. This is a concept that emerges naturally in a probabilistic (Bayesian) framework, where the regularization can be performed by selecting a larger prior probability over simpler models; but

also in statistical learning theory, where the goal is to minimize over two quantities: the 'empirical risk' and the 'structural risk', which roughly corresponds to the error over the training set and the predicted error in unseen data due to over-fitting.

Confidence analysis of a neural network

Supervised neural networks that use an MSE cost function can use formal statistical methods to determine the confidence of the trained model. The MSE on a validation set can be used as an estimate for variance. This value can then be used to calculate the confidence interval of the output of the network, assuming a normal distribution. A confidence analysis made this way is statistically valid as long as the output probability distribution stays the same and the network is not modified.

By assigning a softmax activation function, a generalization of the logistic function, on the output layer of the neural network (or a softmax component in a component-based neural network) for categorical target variables, the outputs can be interpreted as posterior probabilities. This is very useful in classification as it gives a certainty measure on classifications.

The softmax activation function is:

$$y_i = \frac{e^{x_i}}{\sum_{j=1}^{c} e^{x_j}}$$

Controversies

Training issues

A common criticism of neural networks, particularly in robotics, is that they require a large diversity of training for real-world operation. This is not surprising, since any learning machine needs sufficient representative examples in order to capture the underlying structure that allows it to generalize to new cases. Dean Pomerleau, in his research presented in the paper "Knowledge-based Training of Artificial Neural Networks for Autonomous Robot Driving," uses a neural network to train a robotic vehicle to drive on multiple types of roads (single lane, multi-lane, dirt, etc.). A large amount of his research is devoted to (1) extrapolating multiple training scenarios from a single training experience, and (2) preserving past training diversity so that the system does not become over-trained (if, for example, it is presented with a series of right turns – it should not learn to always turn right). These issues are common in neural networks that must decide from amongst a wide variety of responses, but can be dealt with in several ways, for example by randomly shuffling the training examples, by using a numerical optimization

algorithm that does not take too large steps when changing the network connections following an example, or by grouping examples in so-called mini-batches.

Hardware issues

To implement large and effective software neural networks, considerable processing and storage resources need to be committed. While the brain has hardware tailored to the task of processing signals through a graph of neurons, simulating even a most simplified form on Von Neumann technology may compel a neural network designer to fill many millions of database rows for its connections – which can consume vast amounts of computer memory and hard disk space. Furthermore, the designer of neural network systems will often need to simulate the transmission of signals through many of these connections and their associated neurons – which must often be matched with incredible amounts of CPU processing power and time. While neural networks often yield *effective* programs, they too often do so at the cost of *efficiency* (they tend to consume considerable amounts of time and money).

Computing power continues to grow roughly according to Moore's Law, which may provide sufficient resources to accomplish new tasks. Neuromorphic engineering addresses the hardware difficulty directly, by constructing non-Von-Neumann chips with circuits designed to implement neural nets from the ground up.

Practical counterexamples to criticisms

Arguments against Dewdney's position are that neural nets have been successfully used to solve many complex and diverse tasks, ranging from autonomously flying aircraft to detecting credit card fraud .

Technology writer Roger Bridgman commented on Dewdney's statements about neural nets:

Neural networks, for instance, are in the dock not only because they have been hyped to high heaven, (what hasn't?) but also because you could create a successful net without understanding how it worked: the bunch of numbers that captures its behaviour would in all probability be "an opaque, unreadable table...valueless as a scientific resource".

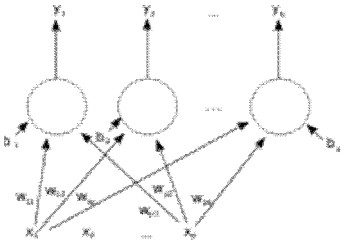
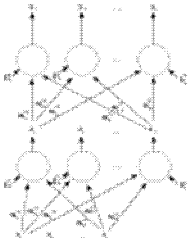
In spite of his emphatic declaration that science is not technology, Dewdney seems here to pillory neural nets as bad science when most of those devising them are just trying to be good engineers. An unreadable table that a useful machine could read would still be well worth having.

Although it is true that analyzing what has been learned by an artificial neural network is difficult, it is much easier to do so than to analyze what has been learned by a biological neural network. Furthermore, researchers involved in exploring learning algorithms for neural networks are gradually uncovering generic principles which allow a learning machine to be successful. For example, Bengio and LeCun (2007) wrote an article regarding local vs non-local learning, as well as shallow vs deep architecture.

Hybrid approaches

Some other criticisms came from believers of hybrid models (combining neural networks and symbolic approaches). They advocate intermix of these two approaches and believe that hybrid models can better capture the mechanisms of the human mind.

Gallery

- 
  A single-layer feedforward artificial neural network. Arrows originating from x_2 are omitted for clarity. There are p inputs to this network and q outputs. In this system, the value of the qth output, y_q would be calculated as $y_q = \sum (x_i * w_{iq})$
- 
  A two-layer feedforward artificial neural network.

Bee Algorithm (BA)

The Bees Algorithm [9] or BA is a bio-inspired metaheuristic behavior of honey bees and how they searching for plant to obtain the necessary pollen for honey production.

A colony of bees search in a large territory looking for new sources of food and begins to thrive on discovering new food sources. When these sources have much pollen are visited by large numbers of bees and when the pollen decreases the number of bees collected from these sources decreases too.

When season for recollecting pollen start, the colony sent so many bees, which are called scouts bees to reconnoiter randomly the territory to inform at the colony where are the best food sources. Once the harvesting season starts the colony maintains a certain percentage of their scout bees in order to detect new and better sources of food. When scout bees have returned to the colony and found a better food source than the currently is using the colony, makes the Dance by means of which transmits the exact position of the source food and then the colony began to send more bees to the food source.

An application of the Bee Swarm Optimization BSO to the Knapsack Problem is given below.

Require: population size (ps), neighborhood (n).

Initialize population with random solutions.

Evaluate fitness of the population.

While stopping criterion not met do

select the sites for neighborhood search
recruit bees for n selected sites and evaluate their fitness
select the fittest bee from each site
assign remaining bees to search randomly and evaluate their fitness
end while

Particle Swarm Optimization (PSO)

The Particle Swarm Optimization [4][7] or PSO is a Bio-inspired metaheuristic in flocks of birds or schools of fish. It was developed by J. Kennedy and R. Eberthart based on a concept called social metaphor, this metaheuristics simulates a society where all individuals contribute their knowledge to obtain a better solution, there are three factors that influence for change in status or behavior of an individual:

• The Knowledge of the environment or adaptation which speaks of the importance given to the experience of the individual.

• His Experience or local memory is the importance given to the best result found by the individual.

• The Experience of their neighbors or Global memory referred to how important it is the best result I have obtained their neighbors or other individuals.

In this metaheuristic each individual is called particle and moves through a multidimensional space that represents the social space or search space depends on the dimension of space which depends on the variables used to represent the problem.

For the update of each particle using something called velocity vector which tells them how fast it will move the particle in each of the dimensions, the method for updating the speed of PSO is given by equation (10), and it is updating by the equation (11).

where:

• is the velocity of the i-th particle
• is adjustment factor to the environment.
• is the memory coefficient in the neighborhood.
• is the coefficient memory.
• is the position of the i-th particle.
• is the best position found so far by all particles.
• is the best position found by the i-th particle

The Algorithm of the PSO applied to the Knapsack Problem is given below.

Require: adaptation to environment coefficient, local memory coefficient, neighborhood memory coefficient, swarm size
Start the swarm particles.
Start the velocity vector for each particle in the swarm.
While stopping criterion not met do
for i=1 to n do
if the i-particle's fitness is better than the local best then replace the local best with the iparticle

if the i-particle's fitness is better than the global best then replace the global best with the iparticle.
Update the velocity vector
Update the particle's position with the velocity vector
end for
end while

Bee Swarm Optimization (BSO)

The Bee Swarm Optimization or BSO is a hybrid metaheuristic population between the PSO and the BA. The main idea of BSO is based on taking the best of each metaheuristics to obtain better results than they would obtain.

The BSO use the velocity vector and the way to updating it, equation (10), and applies the social metaphor to get better results from the PSO and use the exploration and a search radius from the BA to indicate which is where they look for a better result.

The first thing that the BSO do is update the position of the particles through the velocity vector and then select a specified number of particles, in this case it is proposed to select the best as being the new food supply that the scout bees discovered,

and conducted a search of the area enclosed by the radius search and if there are a better solution in this area than the same particle around which are looking for then the particle is replaced with the position of the best solution found in the area.

For this metaheuristic to work properly you need a way to determine what the next and earlier particle position, if we cannot determine this then it is impossible to apply this meta-heuristic because you would not know what the next and previous particle to search in that area.

We defined the next and before elements like binary operations, adding and subtracting one number to the solution vector. For example if we have a Knapsack Problem with 5 elements the solution vector will have 5 spaces and in each space can be 0 or 1, we can see the example of the next and before solution vector in the Figure 1.

Fig. 1. Example of the previous and next solution vector

The algorithm of the BSO applied to the Knapsack Problem implemented in the present paper is the following.

Require: adaptation to environment coefficient, local memory coefficient, neighborhood memory coefficient, swarm size, scout bees, search radio
Start the swarm particles.
Start the velocity vector for each.
While stopping criterion not met do
for i=1 to n do
if the i-particle's fitness is better than the local best then replace the local best with the iparticle
if the i-particle's fitness is better than the global best then replace the global best with the iparticle.

Update the velocity vector
Update the particle's position with the velocity vector
Choose the best particles for all best particle
Search if there are some better particle in the search radio and if exist it replace the particle with the best particle in the search radio
end for all
end for
end while

GENETIC ALGORITHM

BASIC CONCEPT:

GA encodes the decision variables of a search problem into finite-length strings of alphabets of certain cardinality. The strings which are candidate solutions to the search problem are referred to as chromosomes, the alphabets are referred to as genes and the values of genes are called alleles. For example, in a problem such as the traveling salesman problem, a chromosome represents a route, and a gene may represent a city. In contrast to traditional optimization techniques, GAs work with coding of parameters, rather than the parameters themselves. To evolve good solutions and to implement natural selection, we need a measure for distinguishing good solutions from bad solutions. The measure could be an objective function that is a mathematical model or a computer simulation, or it can be a subjective function where humans choose better solutions over worse ones. In essence, the fitness measure must determine a candidate solution’s relative fitness, which will subsequently be used by the GA to guide the evolution of good solutions. Another important concept of GAs is the notion of population. Unlike traditional search methods, genetic algorithms rely on a population of candidate solutions.

The population size, which is usually a user-specified parameter, is one of the important factors affecting the scalability and performance of genetic algorithms. For example, small population sizes might lead to premature convergence and yield substandard solutions. On the other hand, large population sizes lead to unnecessary expenditure of valuable computational time. Once the problem is encoded in a chromosomal manner and a fitness measure for discriminating good solutions from bad ones has been chosen, we can start to evolve solutions to the search problem using the following steps:

1. Initialization: The initial population of candidate solutions is usually generated randomly across the search space. However, domain-specific knowledge or other information can be easily incorporated.
2. Evaluation: Once the population is initialized or an offspring population is created, the fitness values of the candidate solutions are evaluated.
3. Selection: Selection allocates more copies of those solutions with higher fitness values and thus imposes the survival-of-the-fittest mechanism on the candidate solutions. The main idea of selection is to prefer better solutions to worse ones, and many selection procedures have been proposed to accomplish this idea, including roulette-wheel selection, stochastic universal selection, ranking selection and tournament selection, some of which are described in the next section.
4. Recombination: Recombination combines parts of two or more parental solutions to create new, possibly better solutions (i.e. offspring). There are many ways of accomplishing this (some of which are discussed in the next section), and competent performance depends on a properly designed recombination mechanism. The offspring under recombination will not be identical to any particular parent and will instead combine parental traits in a novel manner
5. Mutation: While recombination operates on two or more parental chromosomes, mutation locally but randomly modifies a solution. Again, there are many variations of mutation, but it usually involves one or more changes being made to an individual's trait or traits. In other words, mutation performs a random walk in the vicinity of a candidate solution.
6. Replacement. The offspring population created by selection, recombination, and mutation replaces the original parental population. Many replacement techniques such as elitist replacement, generation-wise replacement and steady-state replacement methods are used in GAs.
7. Repeat steps 2–6 until a terminating condition is met. Goldberg has likened GAs to mechanistic versions of certain modes of human innovation and has shown that these operators when analyzed individually are ineffective, but when combined together they can work well.

ENCODING:

As for any search and learning method, the way in which candidate solutions are encoded is a central, if not the central, factor in the success of a genetic algorithm. Most GA applications use fixed−length, fixed−order bit strings to encode candidate solutions. However, in recent years, there have been many experiments with other kinds of encodings,Common approaches used are:

Binary Encoding: Every chromosome is a string of 0 or 1.Suppose we have a knapsack of capacity C and N items, then we can encode this problem as follows Chromosome, in this case is a string of 0s and 1s with N bits Represent item i of problem with i^{th} bit in the chromosome i^{th} bit is 1 iff i^{th} item has been selected, 0 otherwise. The set of all such chromosomes (2^N) is the solution space of the problem.

Chromosome 1: 1 0 1 1 0 0 1 0 1 1 0 0 1 0 1 0 1 1 1 0 0 1 0 1
Chromosome 2: 1 1 1 1 1 1 1 1 0 0 0 0 0 1 1 0 0 0 0 0 1 1 1 1 1

The example shown above has 24 items (and therefore 24 bits) with item1 selected in both chromosome 1 and 2 whereas item2 is selected in chromosome 2 but not in chromosome 1.

Permutation Encoding (Travelling Salesman Problem): Every chromosome is a string of numbers that represent position in a sequence.Problem description: There are cities and given distances between them. Travelling salesman has to visit all of them, but he doesn't want to travel more than necessary. Find a sequence of cities with a minimal travelled distance.

Chromosome A: 1 5 3 2 6 4 7 9 8
Chromosome B: 8 5 6 7 2 3 1 4 9

Encoding: Here, encoded chromosomes describe the order of cities the salesman visits. For example, in chromosome A, the salesman visits city-1 followed by city-5 followed by city-3 and so on.

Tree Encoding : (Genetic Programming) In tree encoding, every chromosome is a tree of some objects, such as functions or commands in programming language. Tree encoding is useful for evolving programs or any other structures that can be encoded in trees.

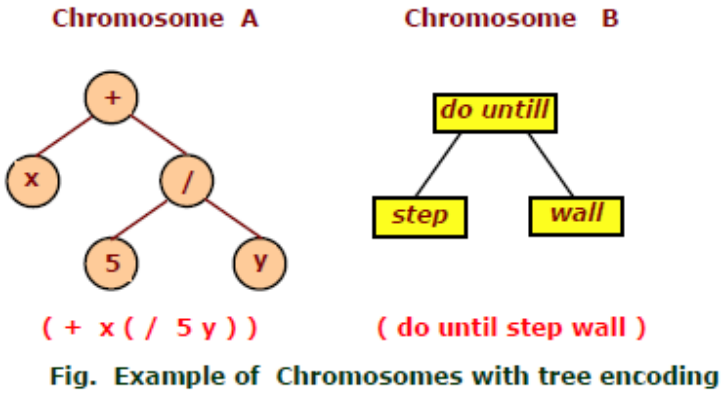


Fig. Example of Chromosomes with tree encoding

Value Encoding: Every chromosome is a sequence of some values (real numbers, characters or objects).Direct value encoding can be used in problems, where some complicated values, such as real numbers, are used. Use of binary encoding for this type of problems would be very difficult. In value encoding, every chromosome is a string of some values. Values can be anything connected to problem, form numbers, real numbers or chars to some complicated objects.

Chromosome A 1.2324 5.3243 0.4556 2.3293 2.4545
Chromosome B ABDJEIFJDHDIERJFDLDFLFEGT
Chromosome C (back), (back), (right), (forward), (left)

Example of Problem: Finding weights for neural network
The problem: There is some neural network with given architecture. Find weights for inputs of neurons to train the network for wanted output.
Encoding**:** Real values in chromosomes represent corresponding weights for inputs.

FITNESS FUNCTION:

A fitness function is a particular type of objective function that is used to summaries, as a single figure of merit, how close a given design solution is to achieving the set aims. In particular, in the fields of genetic programming and genetic algorithms, each design solution is represented as a string of numbers (referred to as a chromosome). After each round of testing, or simulation, the idea is to delete the 'n' worst design solutions, and to breed 'n' new ones from the best design solutions. Each design solution, therefore, needs to be awarded a figure of merit, to indicate how close it came to meeting the overall specification, and this is generated by applying the fitness function to the test, or simulation, results obtained from that solution. The reason that genetic algorithms cannot be considered to be a lazy way of performing design work is precisely because of the effort involved in designing a workable fitness function. Even though it is no longer the human designer, but the computer, that comes up with the final design, it is the human designer who has to design the fitness function. If this is designed badly, the algorithm will either converge on an inappropriate solution, or will have difficulty converging at all. Moreover, the fitness function must not only correlate closely with the designer's goal, it must also be computed quickly. Speed of execution is very important, as a typical genetic algorithm must be iterated many times in order to produce a usable result for a non-trivial problem. Fitness approximation may be appropriate, especially in the following cases:

- Fitness computation time of a single solution is extremely high
- Precise model for fitness computation is missing
- The fitness function is uncertain or noisy.

Two main classes of fitness functions exist: one where the fitness function does not change, as in optimizing a fixed function or testing with a fixed set of test cases; and one where the fitness function is mutable, as in niche differentiation or co-evolving the set of test cases. Another way of looking at fitness functions is in terms of a fitness landscape, which shows the fitness for each possible chromosome. Definition of the fitness function is not straightforward in many cases and often is performed iteratively if the fittest solutions produced by GA are not what is desired. In some cases, it is very hard or impossible to come up even with a guess of what fitness function definition might be. Interactive genetic algorithms address this difficulty by outsourcing evaluation to external agents (normally humans).GAs are naturally suitable for solving maximization problems. Maximization problems are usually transformed into maximization problem by suitable transformation. In general, a fitness function *F(i)* is first derived from the objective function and used in successive genetic operations. Fitness in biological sense is a quality value which is a measure of the reproductive efficiency of chromosomes. In genetic algorithm, fitness is used to allocate reproductive traits to the individuals in the population and thus act as some measure of goodness to be maximized. This means that individuals with higher fitness value will have higher probability of being selected as candidates for further examination. Certain genetic operators require that the fitness function be non-negative, although certain operators need not have this requirement. For maximization problems, the fitness function can be considered to be the same as the objective function or $F(i)=O(i)$. For minimization problems, to generate non-negative values in all the cases and to reflect the relative fitness of individual string, it is necessary to map the underlying natural objective function to fitness function form. A number of such transformations is possible. Two commonly adopted fitness mappings are presented below.

$$F(x)=\frac{1}{1+f(x)}$$

This transformation does not alter the location of the minimum, but converts a minimization problem to an equivalent maximization problem. An alternate function to transform the objective function to get the fitness value $F(i)$ as below.

$$F(i) = V - \frac{O(i)\ P}{\sum_{i=1}^{P} O(i)}$$

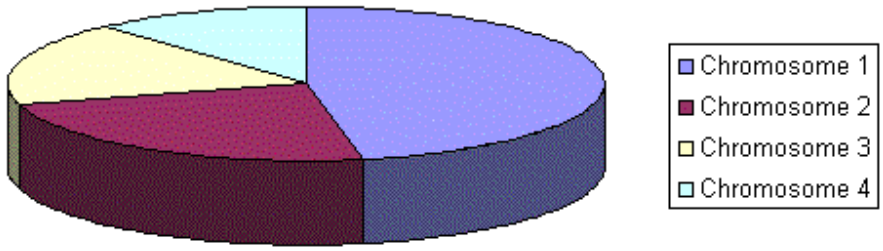
where, *O(i)* is the objective function value of *i* th individual, P is the population size and V is a large value to ensure non-negative fitness values. The value of V adopted in this work is the maximum value of the second term of equation so that the fitness value corresponding to maximum value of the objective function is zero. This transformation also does not alter the location of the solution, but converts a minimization problem to an equivalent maximization problem. The fitness function value of a string is known as the string fitness.

SELECTION:

Chromosomes are selected from the population to be parents to crossover. The problem is how to select these chromosomes. According to Darwin's evolution theory the best ones should survive and create new offspring. In principle, a population of individuals selected from the search space , often in a random manner, serves as candidate solutions to optimize the problem.The individuals in this population are evaluated through ("fitness") adaptation function. A selection mechanism is then used to select individuals to be used as parents to those of the next generation. These individuals will then be crossed and mutated to form the new offspring. The next generation is finally formed by an alternative mechanism between parents and their offspring [4]. This process is repeated until a certain satisfaction condition. There are many methods how to select the best chromosomes, for example roulette wheel selection, Boltzman selection, tournament selection, rank selection, steady state selection and some others.

Roulette Wheel Selection :

Parents are selected according to their fitness. The better the chromosomes are, the more chances to be selected they have. Imagine a roulette wheel where are placed all chromosomes in the population, every has its place big accordingly to its fitness function, like on the following picture.



Then a marble is thrown there and selects the chromosome. Chromosome with bigger fitness will be selected more times.The conspicuous characteristic of this selection method is the fact that it gives to each individual *i* of the current population a probability of p(i) being selected, proportional to its fitness f(i).

$$p(i) = \frac{f(i)}{\sum_{j=1}^{n} f(j)}$$

Where *n* denotes the population size in terms of the number of individuals. A well-known drawback of this technique is the risk of premature convergence of the GA to a local optimum, due to the possible presence of a dominant individual that always wins the competition and is selected as a parent.

Linear Rank Selection (LRS):

LRS is also a variant of RWS that tries to overcome the drawback of premature convergence of the GA to a local optimum. It is based on the rank of individuals rather than on their fitness. The rank n is accorded to the best individual whilst the worst individual gets the rank 1. Thus, based on its rank, each individual *i* has the probability of being selected given by the expression

$$p(\mathrm{i}) = \frac{\mathrm{rank(i)}}{\mathrm{n} * (\mathrm{n}-1)}$$

Exponential Rank Selection (ERS) :

The ERS is based on the same principle as LRS, but it differs from LRS by the probability of selecting each individual. For ERS, this probability is given by the expression:

$$\mathrm{p(i)} = 1.0 * \exp\left(\frac{-rang(i)}{\mathrm{c}}\right)$$

$$\mathrm{c} = \frac{(n*2*(n-1))}{(6*(n-1)+n)}$$

Tournament Selection (TOS)

Tournament selection is a variant of rank-based selection methods. Its principle consists in randomly selecting a set of *k* individuals. These individuals are then ranked according to their relative fitness and the fittest individual is selected for reproduction. The whole process is

repeated n times for the entire population. Hence, the probability of each individual to be selected is given by the expression:

$$p(i) = \begin{cases} \frac{C_{n-1}^{k-1}}{C_n^k} & if\ i \in [1, n-k-1] \\ 0 & if\ i \in [n-k, n] \end{cases}$$

Steady-State Selection:

This is not particular method of selecting parents. Main idea of this selection is that big part of chromosomes should survive to next generation. GA then works in a following way. In every generation are selected a few (good - with high fitness) chromosomes for creating a new offspring. Then some (bad - with low fitness) chromosomes are removed and the new offspring is placed in their place. The rest of population survives to new generation.

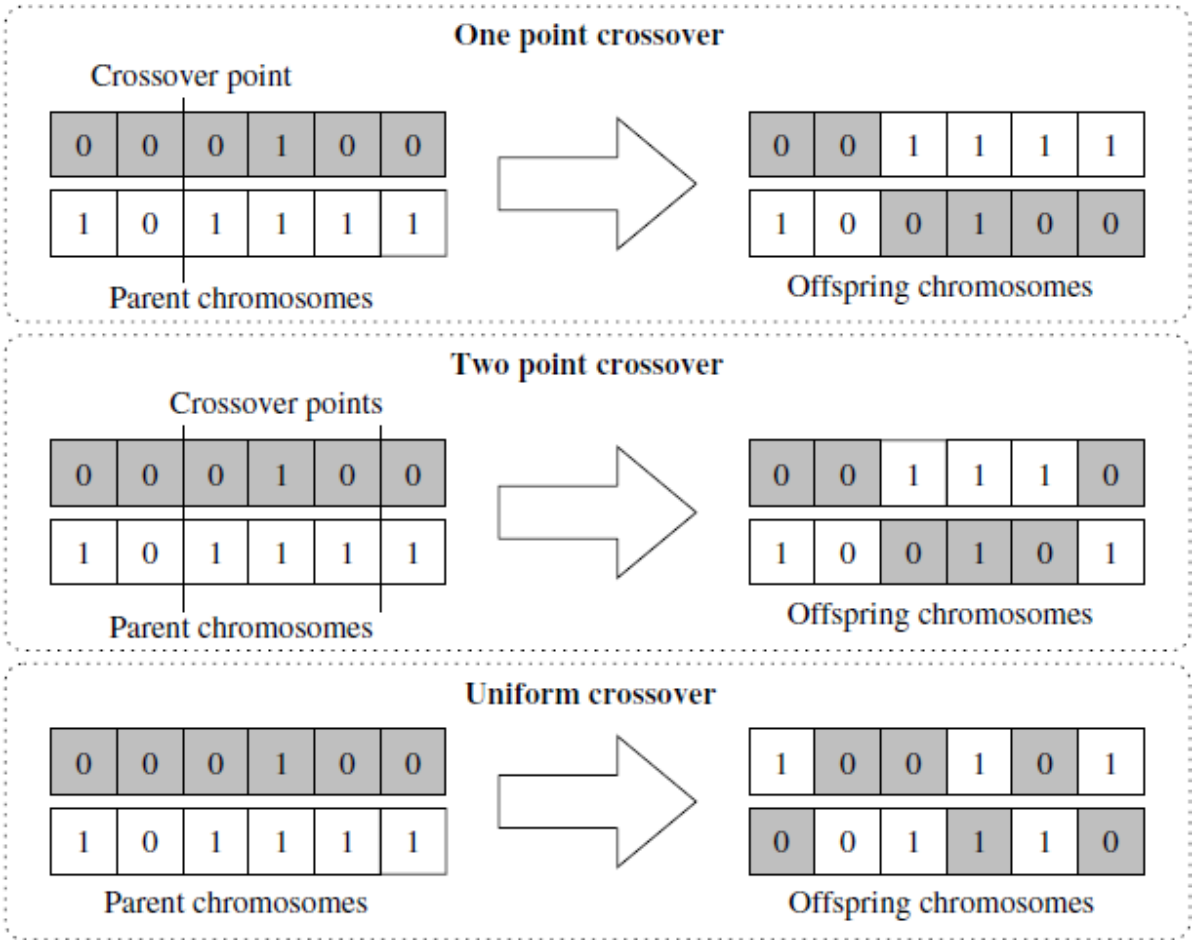
Elitism

Idea of elitism has been already introduced. When creating new population by crossover and mutation, we have a big chance, that we will loose the best chromosome. Elitism is name of method, which first copies the best chromosome (or a few best chromosomes) to new population. The rest is done in classical way. Elitism can very rapidly increase performance of GA, because it prevents losing the best found solution.

REPRODUCTION:

After selection, individuals from the mating pool are recombined (or crossed over) to create new, hopefully better, offspring. In the GA literature, many crossover methods have been designed and some of them are described in this section. In most recombination operators, two individuals are randomly selected and are recombined with a probability *pc*, called the crossover probability. That is, a uniform random number, *r*, is generated and if $r \leq pc$, the two randomly selected individuals undergo recombination. Otherwise, that is, if $r > pc$, the two offspring are simply copies of their parents. The value of *pc* can either be set experimentally, or can be set based on schema-theorem principles.

***k*-point Crossover** One-point, and two-point crossovers are the simplest and most widely applied crossover methods. In one-point crossover, illustrated in Figure a crossover site is selected at random over the string length, and the alleles on one side of the site are exchanged between the individuals. In two-point crossover, two crossover sites are randomly selected. The alleles between the two sites are exchanged between the two randomly paired individuals. Two-point crossover is also illustrated in Figure The concept of one-point crossover can be extended to *k*-point crossover, where *k* crossover points are used, rather than just one or two.



Uniform Crossover Another common recombination operator is uniform crossover. In uniform crossover, illustrated in Figure every allele is exchanged between the a pair of randomly selected chromosomes with a certain probability, *pe*, known as the swapping probability. Usually the swapping probability value is taken to be 0.5.

Uniform Order-Based Crossover : The *k*-point and uniform crossover methods described above are not well suited for search problems with permutation codes such as the ones used in the traveling salesman problem. They often create offspring that represent invalid solutions for the search problem. Therefore, when solving search problems with permutation codes, a problem-specific repair mechanism is often required (and used) in conjunction with the above recombination methods to always create valid candidate solutions. Another alternative is to use recombination methods developed specifically for permutation codes, which always generate valid candidate solutions. Several such crossover techniques are described in the following paragraphs starting with the uniform order-based crossover. In uniform order-based crossover, two parents (say P1 and P2) are randomly selected and a random binary template is generated .Some of the genes for offspring C1 are filled by taking the genes from parent P1 where there is a one in the template. At this point we have C1 partially filled, but it has some “gaps”. The genes of parent P1 in the positions corresponding to zeros in the template are taken and sorted in the

same order as they appear in parent P2. The sorted list is used to fill the gaps in C1. Offspring C2 is created by using a similar process

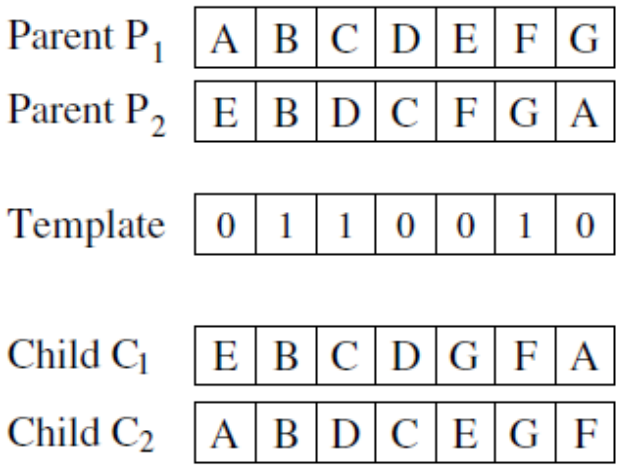


Illustration of uniform order crossover.

Order-Based Crossover The order-based crossover operator (Davis, 1985) is a variation of the uniform order-based crossover in which two parents are randomly selected and two random crossover sites are generated. The genes between the cut points are copied to the children. Starting from the second crossover site copy the genes that are not already present in the offspring from the alternative parent (the parent other than the one whose genes are copied by the offspring in the initial phase) in the order they appear. For example, as shown in Figure, for offspring C1, since alleles C, D, and E are copied from the parent P1, we get alleles B, G, F, and A from the parent P2. Starting from the second crossover site, which is the sixth gene, we copy alleles B and G as the sixth and seventh genes respectively. We then wrap around and copy alleles F and A as the first and second genes.

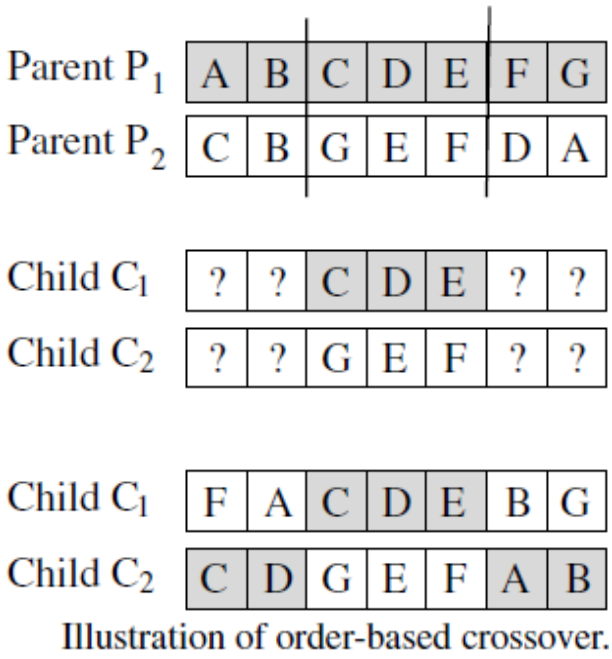


Illustration of order-based crossover.

Partially Matched Crossover (PMX): Apart from always generating valid offspring, the PMX operator also preserves orderings within the chromosome. In PMX, two parents are randomly selected and two random crossover sites are generated. Alleles within the two crossover sites of a parent are exchanged with the alleles corresponding to those mapped by the other parent. For example, as illustrated in Figure, looking at parent P1, the first gene within the two crossover sites, 5, maps to 2 in P2. Therefore, genes 5 and 2 are swapped in P1. Similarly we swap 6 and 3, and 10 and 7 to create the offspring C1. After all exchanges it can be seen that we have achieved a duplication of the ordering of one of the genes in between the crossover point within the opposite chromosome, and vice versa.

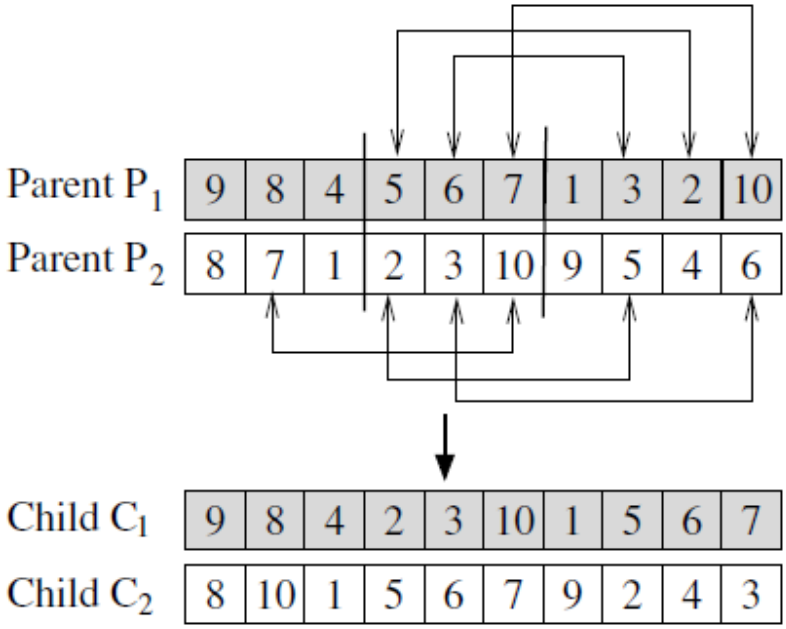


Illustration of partially matched crossover.

Cycle Crossover (CX): We describe cycle crossover with help of a simple illustration (reproduced from Goldberg (1989b) with permission). Consider two randomly selected parents P1 and P2 as shown in Figure that are solutions to a traveling salesman problem. The offspring C1 receives the first variable representing city 9) from P1. We then choose the variable that maps onto the same position in P2. Since city 9 is chosen from P1 which maps to city 1 in P2, we choose city 1 and place it into C1 in the same position as it appears in P1 (fourth gene), as shown in Figure 4.5. City 1 in P1 now maps to city 4 in P2, so we place city 4 in C1 in the same position it occupies in P1 (sixth gene). We continue this process once more and copy city 6 to the ninth gene of C1 from P1. At this point, since city 6 in P1 maps to city 9 in P2, we should take city 9 and place it in C1, but this has already been done, so we have completed a cycle; which is where this operator gets its name. The missing cities in offspring C1 is filled from P2. Offspring C2 is created in the same way by starting with the first city of parent P2

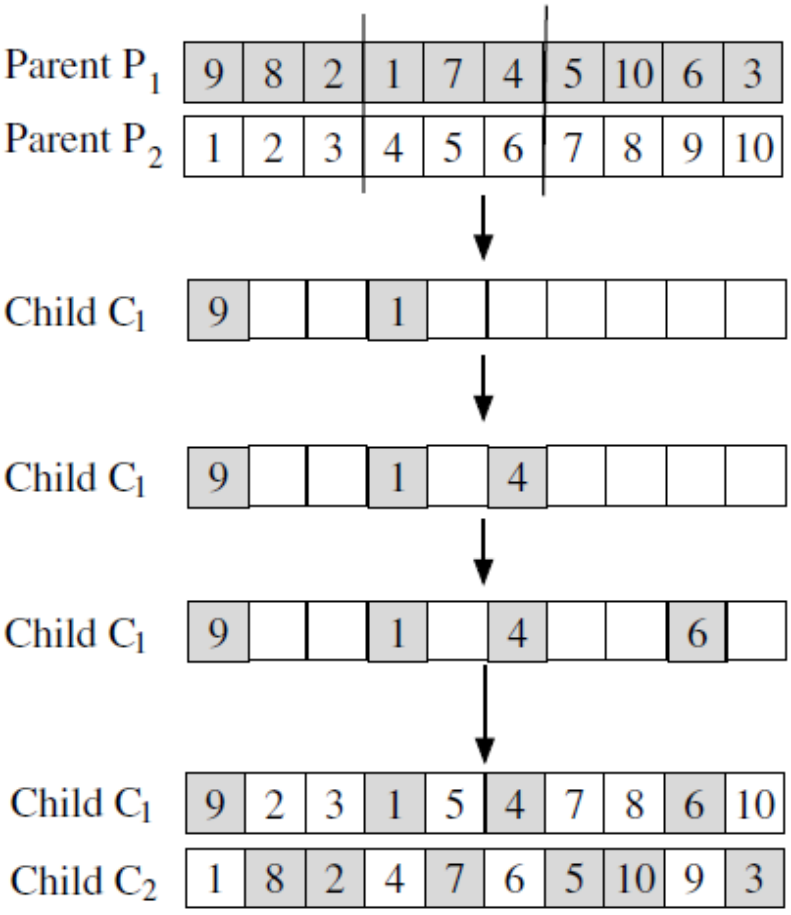


Illustration of cycle crossover.

DIFFERENCES BETWEEN GA'S AND TRADITIONAL METHODS:

The brief list, based on Goldberg (1989), of the essential differences between GAs and other forms of optimization is the following:.

- Genetic algorithms a coded form of the function values (parameter set), rather than with the actual values them. So, for example, if we want to find the minimum of the function f(x) =x3+x2+5, the GA would not deal directly with x or y values, but with strings that encode these values. For this case, strings representing the binary x values should be used.
- Genetic algorithms use a set, or population, of points to conduct a search, not just a single point on the problem space. This gives GAs the power to search noisy spaces littered with local optimum points. Instead of relying on a single point to search through the space, the GAs looks at many different areas of the problem space at once, and uses all of this information to guide it.
- Genetic algorithms use only payoff information to guide themselves through the problem space. Many search techniques need a variety of information to guide themselves. Hill climbing methods require derivatives, for example. The only information a GA needs is some measure of fitness about a point in the space (sometimes known as an objective function value). Once the GA knows the current measure of "goodness" about a point, it can use this to continue searching for the optimum.
- GAs are probabilistic in nature, not deterministic. This is a direct result of the randomization techniques used by GAs.
- GAs are inherently parallel. Here lies one of the most powerful features of genetic algorithms. GAs, by their nature, are very parallel, dealing with a large number of points (strings) simultaneously. Holland has estimated that a GA processing n strings at each generation, the GA in reality processes n^3 useful substrings.

  * GA's work with string coding of variables instead of variables so that coding discrediting the search space even though the function is continuous.

  * GA's work with population of points instead of single point.

  * In GA's previously found good information is emphasized using reproduction operator and propagated adaptively through crossover and mutation operators.

  * GA does not require any auxiliary information except the objective function values.
  * GA uses the probabilities in their operators.
  This nature of narrowing the search spaces the search progresses is adaptive and is the unique characteristic of Genetic Algorithms.

SOME APPLICATIONS OF GENETIC ALGORITHMS:

The algorithm described above is very simple, but variations on this basic theme have been used in a large number of scientific and engineering problems and models, including the following:

- Optimization: GAs have been used in a wide variety of optimization tasks, including numerical optimization as well as combinatorial optimization problems such as circuit layout and job-shop scheduling.
- Automatic Programming: GAs have been used to evolve computer programs for specific tasks, and to design other computational structures, such as cellular automata and sorting networks.
- Machine learning: GAs have been used for many machine-learning applications, including classification and prediction tasks such as the prediction of weather or protein structure. GAs have also been used to evolve aspects of particular machine-learning systems, such as weights for neural networks, rules for learning classifier systems or symbolic production systems, and sensors for robots.
- Economic models: GAs have been used to model processes of innovation, the development of bidding strategies, and the emergence of economic markets.
- Immune system models: GAs have been used to model various aspects of the natural immune system including somatic mutation during an individual's lifetime and the discovery of multi-gene families during evolutionary time.
- Ecological models: GAs have been used to model ecological phenomena such as biological arms races, host-parasite co-evolution, symbiosis, and resource flow in ecologies.

Genetic Algorithm Application Areas:

- Dynamic process control
- Induction of rule optimization
- Discovering new connectivity topologies
- Simulating biological models of behavior and evolution
- Complex design of engineering structures
- Pattern recognition
- Scheduling
- Transportation
- Layout and circuit design
- Telecommunication
- Graph-based problems