Soft Computing

Subject Expert Panel

Dr. Ramchandra G. Pawar
Director, SIBACA, Lonavala
Pune

Ashwini Pandit
Subject Matter Expert

Content Review Panel

Gaurav Modi
Subject Matter Expert

Shubhada Pawar
Subject Matter Expert

This book contains the course content for Soft Computing.

First Edition 2013

Printed by
Universal Training Solutions Private Limited

Address
05th Floor, I-Space,
Bavdhan, Pune 411021.

Index

Contents

List of Figures

List of Tables

Abbreviations

AI	-	Artificial Intelligence
ANNs	-	Artificial Neural Networks
ART	-	Adaptive Resonance Theory
ATP	-	Adenosine Tri-phosphate
CAMD	-	Computer-Aided Molecular Design
GA	-	Genetic Algorithms
NMDA	-	N-Methyl-D-Aspartate
SC	-	Soft Computing
SOFM	-	Self Organising Feature Map
STM	-	Short Term Memory

Chapter I

Introduction to Soft Computing

Aim

The aim of this chapter is to:

- introduce development of soft computing
- elucidate genetic programming
- explain fuzzy logic

Objectives

The objectives of this chapter are to:

- explain genetic algorithm
- explicate artificial neurons
- elucidate uses of artificial neural networks

Learning outcome

At the end of this chapter, you will be able to:

- understand using neural networks with genetic algorithms
- identify genetic fuzzy systems
- describe multilayer networks

1.1 Introduction

Soft computing (SC) is a branch, in which, it is tried to build intelligent and wiser machines. Intelligence provides the power to derive the answer and not simply arrive to the answer. Purity of thinking, machine intelligence, freedom to work, dimensions, complexity and fuzziness handling capability increase, as we go higher and higher in the hierarchy as shown in figure below. The final aim is to develop a computer or a machine which will work in a similar way as human beings can do, i.e. the wisdom of human beings can be replicated in computers in some artificial manner.

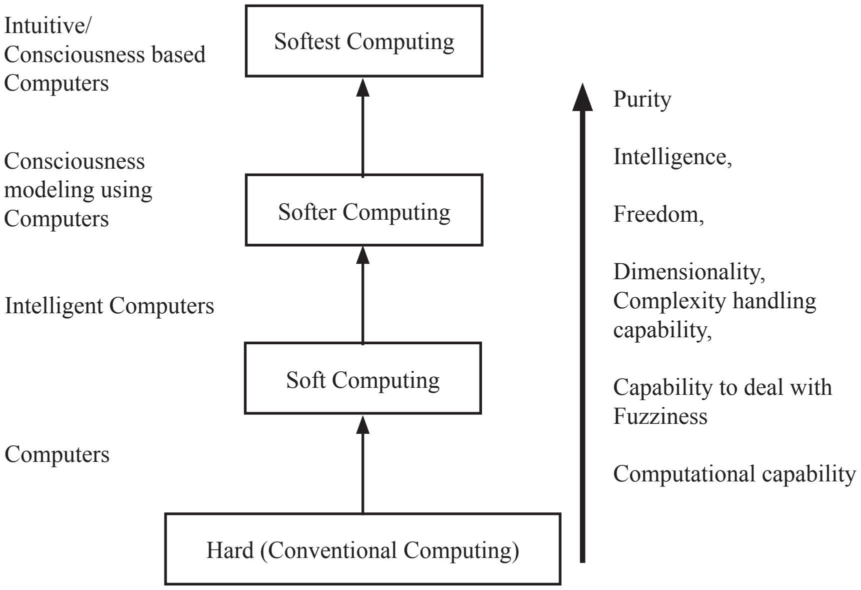


Fig. 1.1 Development of soft computing

Intuitive consciousness/wisdom is also one of the important area in the soft computing, which is always cultivated by meditation. This is indeed, an extraordinary challenge and virtually a new phenomenon, to include consciousness into the computers. Soft computing is an emerging collection of methodologies, which aim to exploit tolerance for imprecision, uncertainty, and partial truth to achieve robustness, tractability and total low cost.

Soft computing methodologies have been advantageous in many applications. In contrast to analytical methods, soft computing methodologies mimic consciousness and cognition in several important respects: they can learn from experience; they can universalize into domains where direct experience is absent; and, through parallel computer architectures that simulate biological processes, they can perform mapping from inputs to the outputs faster than inherently serial analytical representations.

The trade off, however, is a decrease in accuracy. If a tendency towards imprecision could be tolerated, then it should be possible to extend the scope of the applications even to those problems where the analytical and mathematical representations are readily available. The motivation for such an extension is the expected decrease in computational load and consequent increase of computation speeds that permit more robust system.

Soft computing is an umbrella term for a collection of computing techniques. The term was first coined by Professor Lotfi Zadeh, who developed the concept of fuzzy logic in the mid 60's. In his words: "Soft computing differs from conventional (hard) computing in that, unlike hard computing, it is tolerant of imprecision, uncertainty and partial truth. In effect, the role model for soft computing is the human mind. The guiding principle of soft computing is: Exploit the tolerance for imprecision, uncertainty and partial truth to achieve tractability, robustness and low solution cost"

The main techniques in soft computing are evolutionary computing, artificial neural networks, fuzzy logic and Bayesian statistics. Each technique can be used separately, but a powerful advantage of soft computing is the complementary nature of the techniques. Used together they can produce solutions to problems that are too complex or inherently noisy to tackle with conventional mathematical methods.

1.2 Evolutionary Computing

Evolutionary computing uses the Darwinian principle of ‘survival of the fittest’ to evolve optimum solutions to problems. There are a number of evolutionary techniques whose main similarity is the use of a population of random or pseudo-randomly generated solutions to a problem, each of which is judged on their adaptation to the problem being addressed using a ‘fitness function’.

The fitness function evaluates the solution and returns a numerical answer, which rates the solution’s suitability. This population is then used to generate a new population of solutions, by using one or more of the following genetic operators:

- Re-combination/cross-over: This operation involves combining the ‘genetic’ information from two parent solutions to produce an offspring solution.
- Mutation: This operation involves taking a solution and mutating its genetic information slightly.
- Cloning: This operation copies the solution to produce a clone.

The solutions that are used in these operations are selected from the main population pool and placed into the ‘breeding pool’ using a selection method. The protocol for creating this breeding pool differs between the various algorithms, as does the protocol for deciding which candidates enter the next generation. All the techniques require the population size to remain constant between generations. This evaluation and regeneration of the candidate populations is repeated until a near-optimal solution has been created.

1.3 Genetic Algorithms

Of all the evolutionary techniques genetic algorithms have proved to be the most commercially successful. The solutions in these algorithms are encoded into strings. The strings represent the properties that the resulting solution will have, for example a list of the values of variables in an equation or the properties of components in a design. New generations are created using all the genetic operators, but the recombination operator is given a much higher priority than the mutation and cloning operators, and mutation is usually only used as a ‘background’ operation to maintain diversity in the population. Re-combination is achieved by combining sections of the encoded strings from both parents to produce a child, as shown below:

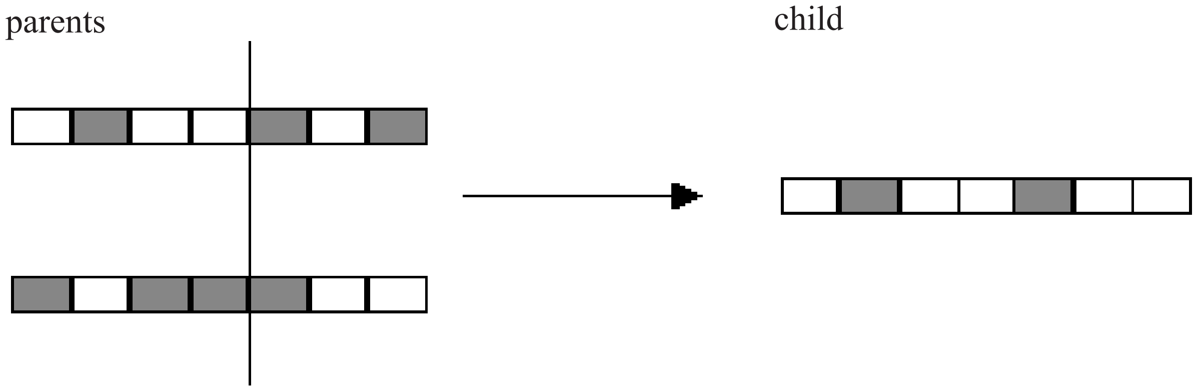


Fig. 1.2 Re-combination

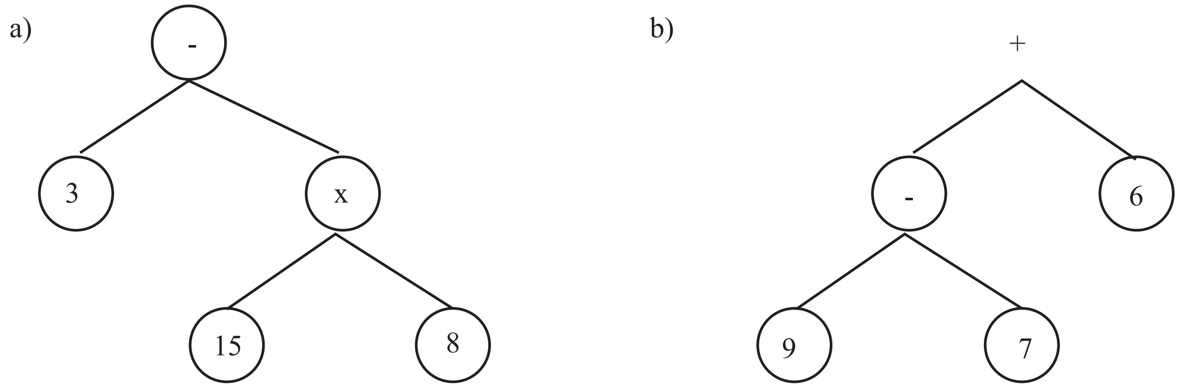
The mutation operator usually performs point mutations. Where values are encoded in binary, this can be as simple as using a bit-flip to alter the values, but where real numbers are used; a random number can be added or subtracted instead. The parent solutions that take part in creating the new generation are chosen based on their fitness. The most common way to do this is to evaluate the whole population using the fitness function and then produce a breeding pool using fitness proportional representation, so that the most successful candidates are more frequently represented.

Another way involves taking random groups of candidates from the main pool and competing them to find the fittest candidates in each group. Both of these techniques result in a selected pool that has a high proportion of 'good' solutions, but which does not exclude the 'bad' solutions. This is important to balance the need to create an optimum solution against the possibility of becoming too set on one evolutionary path, which may not lead to the overall optimum solution.

The replacement policy that is usually applied involves replacing the whole of the previous generation with the newly created individuals, but more subtle methods can be used where the generated children only replace the weakest individuals in the parent population.

1.4 Genetic Programming

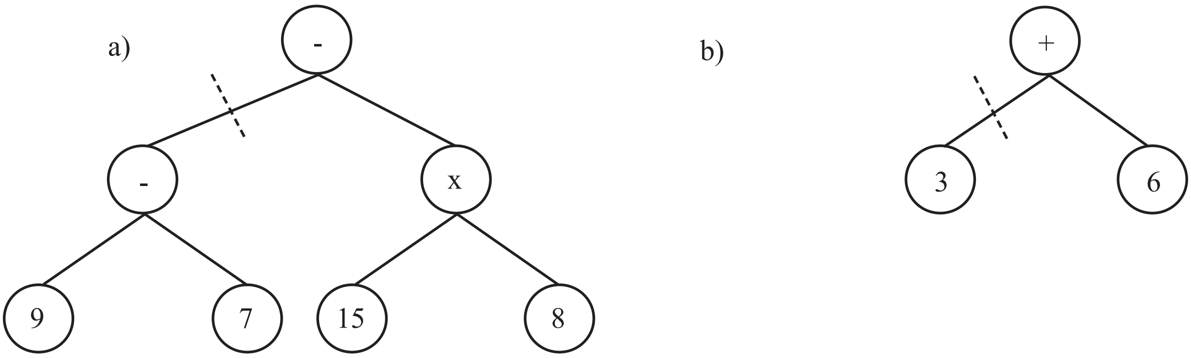
Genetic programming works in a very similar way to genetic algorithms. The main difference between the two techniques is the representation used for the candidate solutions. In genetic programming, the solutions are actual computer programs. As a result the representation of the candidate's genetic material needs to be more complex. Most commonly a tree structure is used to represent the structure of the program. The leaf nodes in this structure are values, and the internal nodes are functions selected from an allowed set. These functions typically would include arithmetic and Boolean operators as well as conditional functions.



Evaluates to
a) 3-(15 x 8)
b) (9 - 7) + 6

Fig. 1.3 Evaluation of the equation a) 3- (15x8) b) (9-7) + 6

The representation of genetic programs requires the genetic operators to act in a slightly different way. The recombination operator is still the most dominant, but its actions are altered to deal with tree structures. Usually recombination involves selecting a sub-tree at random from each parent, and then swapping them.



Evaluates to
a) (9 - 7) - (15 x 8)
b) 3 + 6

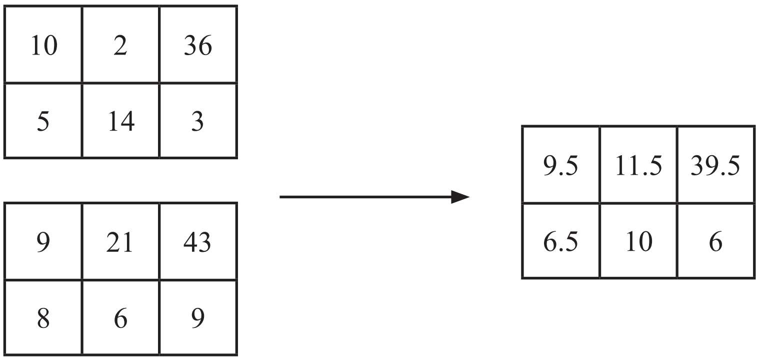
Fig. 1.4 Evaluation of the equation a) (9-7) – (15x8) b) 3+6

Similarly the mutation operator replaces a randomly selected sub tree with a randomly generated sub-tree.

1.5 Evolution Strategies

Evolution strategies were developed by researchers at the Technical Institute of Berlin, who wanted to automate the design of aerodynamic jointed plates. The representation that they used for the candidate solutions was a list of fixed-length, real-valued vectors, which describe the characteristics of the design. Initially they took a single parent design, mutated it and then applied the fitness function to both the child and the parent, choosing the fittest individual to carry on to the next generation. This approach has since been expanded to allow populations of parents and children. The parents are selected entirely randomly and are used to produce a child population with a size that is greater than the population size.

Originally the only genetic operator used was mutation, but an intermediate recombination operator can now be used to blend the characteristics of the parents. This operator works on the values in the parent vectors, and averages the values to produce the child. However, unlike genetic algorithms, mutation is the primary operator, with recombination only used secondarily.



10	2	36
5	14	3

9	21	43
8	6	9

9.5	11.5	39.5
6.5	10	6

Fig. 1.5 Recombination in evolution stages

There are two main types of replacement policies used, namely (μ,λ) and (μ+λ). In this notation μ is the population size and λ is the number of children produced. In the comma method the next generation is selected from the children, using the fitness function to determine the fittest solutions. The plus method selects best candidates from the parents and the children to make up the next generation.

1.6 Evolutionary Programming

The final technique is often grouped together with evolution strategies, as they are very similar in their approach. Both tend to use fixed-length real-valued vectors to represent the solutions and both use the fitness function to select which candidates enter the next generation rather than which candidates become parents. In evolutionary programming the only genetic operator used is mutation, and it is used on the whole of the current generation to produce an equal number of children. The next generation is then selected from this combined pool using the fitness function to choose the best candidates.

1.7 Uses of Evolutionary Computing

Evolutionary computing has already proved itself as a useful tool in a range of industries. One example of this is the pharmaceutical industry, where genetic algorithms have been applied to computer-aided molecular design (CAMD). Designing a molecule to order is a difficult task. To find a new drug to bind to a known receptor you first need to analyse the structure of either other known legends or the receptor itself to produce information on the functional areas of the molecules involved.

This information can be used to assess the binding properties of new molecules, but the new molecules still need to be designed. Due to the complexity of the search space, conventional rule-based or numerical methods tend to be unsuccessful. This is where genetic algorithms can help. By encoding the structure of the chemicals into chromosomes or graphs, genetic algorithms can be used to evolve new chemicals. The fitness function must then build the molecules, checking that they make chemical sense, before assessing their binding properties. By using evolution unexpected chemicals can be produced, which can give chemists insights and suggestions into potential chemical families to investigate.

1.8 Fuzzy Logic

In Boolean logic, the only values allowed are completely true and completely false. This black and white representation of the world can be difficult to implement. If, for example, if you had a group of air temperatures and you asked the question ‘is x hot?’ then it would be difficult in many cases to give a definitive yes or no answer. One way to get around this in Boolean logic would be to define a strict boundary and say that any temperature over 30°C is hot, but this is very artificial. Why should 30.1°C be hot when 30°C is not?

In fuzzy logic you can assign degrees of truth and falsehood. If we use the example above, a temperature of 27°C would be considered ‘fairly’ hot, whereas a temperature of 25 would be ‘not very’ hot. These answers are given a numeric value in the interval 1 to 0, where 1 is the classical Boolean value true and 0 is false, for example ‘fairly’ could be translated to a value of 0.7 and ‘not very’ as 0.1.

1.9 Fuzzy Sets

This leads us into the idea of fuzzy sets. If we keep the example above, the question ‘is x hot?’ is a way of defining a subset of hot temperatures. In Boolean logic the members of the super set (called the ‘universe of discourse’) either belong to the hot subset or do not belong. With fuzzy sets, individuals can be given a degree of membership to a subset. This degree of membership is calculated using a membership function. An example of a membership function for the fuzzy subset ‘hot’ could have the distribution shown below:

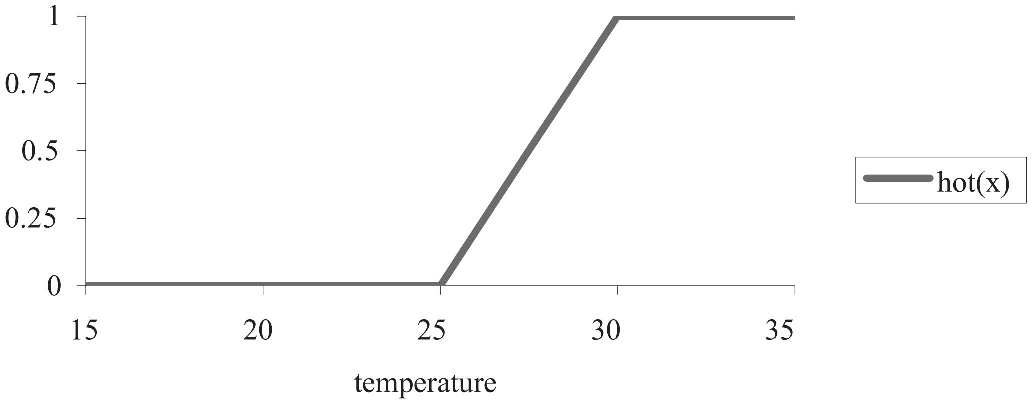


Fig. 1.6 An example of a membership function for the fuzzy subset

This is a fairly simple membership function, but more complex functions can be used.

1.10 Fuzzy Logic Operators

Once you have converted all your inputs into fuzzy values, a process known as fuzzification, you can now apply fuzzy logic. In Boolean logic algebraic sentences are created using the operators AND, OR and NOT. These operators can also be used in fuzzy logic. The most common interpretation of these operators in fuzzy logic is:

NOT x = 1-x
x AND y = MIN(x, y)
x OR y = MAX(x, y)

These operators produce the same results for the true and false values (1 and 0) as Boolean operators, which shows that Boolean logic is a simply a subset of Fuzzy logic. For example, if we define a new fuzzy set for warm temperatures. The membership function could look as follows:

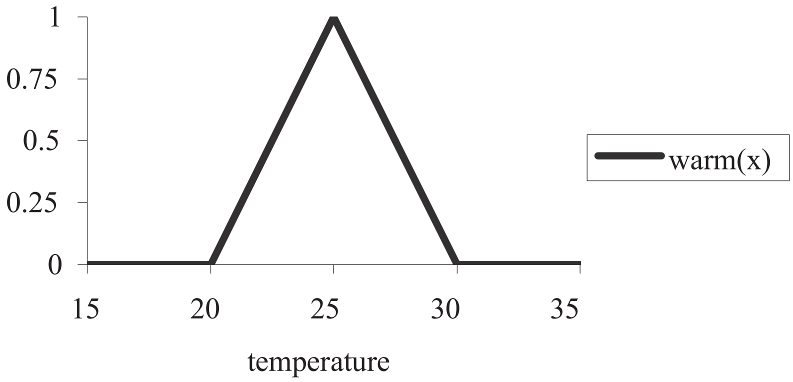


Fig. 1.7 Membership function

We can now construct some logical expressions to show how the fuzzy logic operators work. If a member of the universe of discourse were a temperature of 26.5°C, the membership functions for the two sets would give values of 0.3 for the hot set and 0.7 for the warm set. We can now ask questions such as:

Is x NOT hot?

Which would give the result: 1 – 0.3 = 0.7

Is x hot AND warm?

Which would give the result: MIN (0.3, 0.7) = 0.3

Is x hot OR warm?

Which would give the result: MAX (0.3, 0.7) = 0.7

1.11 Fuzzy Inference Systems

A fuzzy inference system uses fuzzy logical rules to produce output values from input values. These rules take the form: (antecedent) IF car is powerful AND driver is young (consequent) THEN insurance premium is very expensive. The antecedent will produce a numerical answer, for example a 20 year old driver in a Ferrari could be given membership values of 0.8 for young and 0.9 for v powerful giving the antecedent an answer of 0.8.

The consequent assigns a fuzzy set to an output variable, using the answer to the antecedent to alter the fuzzy set's distribution. For example in the rule above we are setting the value of the output variable 'insurance premium' to the fuzzy set 'very expensive', which would have the distribution shown below:

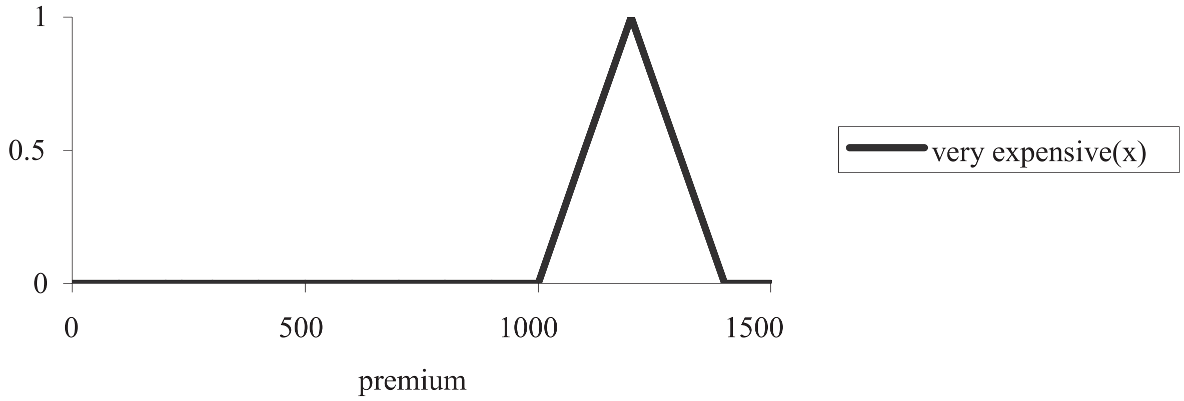


Fig. 1.8 Distribution

The degree to which the output variable belongs to that fuzzy set is defined by the antecedent, so to combine the two parts we truncate the output fuzzy set at the value returned. This is called implication. In this example the antecedent returned a value of 0.8, so we 'cut' the top of the output fuzzy set off at that value.

This fuzzy set output now needs to be converted into a crisp output value in order to be of any use, a process called de-fuzzification. The most conventional way of doing this is to find the centroid of the fuzzy set's distribution, which is the point at the centre of the area under the curve. When more than one rule is defined all the rules are evaluated in parallel. This leads to a situation where the output values can have more than one fuzzy set assigned to them. These fuzzy sets first need to be aggregated before they can be defuzzified; which is done usually by applying a MAX operator to them.

For example if the insurance premium inference system had three more rules:

- IF driver is mature and car is average THEN premium is medium
- IF no claims period is long THEN premium is low
- IF no claims period is short THEN premium is expensive

The resulting premium before de-fuzzification could look like this:

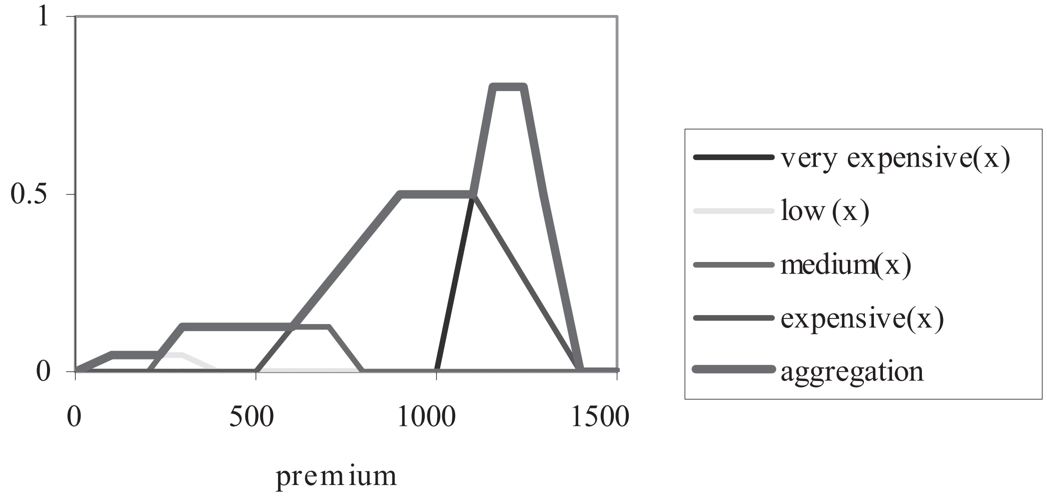


Fig. 1.9 Resulting premium before de-fuzzification

This would then be de-fuzzified, using the centroid function to give a crisp output value.

1.12 Uses of Fuzzy Systems

In Japan, Fuzzy Logic has been used successfully for decades. There are a number of high profile cases that use Fuzzy Logic, for example the subway trains in the Japanese city of Sendai are controlled using a fuzzy logic system. This system was designed using train drivers expert knowledge and has been shown to accelerate and brake more smoothly than human operators and stop at the stations with more accuracy.

More recent examples include the incorporation of fuzzy logic controllers in ABS technology. Braking of a vehicle occurs when the wheels rotate slower than the speed of the vehicle, but when the wheels stop moving completely, i.e. when the car skids, it is less easy to steer and braking is not optimal. The braking effect is related to the difference in velocity between the wheels and the car, known as the ‘slack’. The ABS controller adjusts the by-pass valves on the brake fluid to keep the slack on the wheels optimal for the braking effect.

Unfortunately this is not a linear problem, as the optimal level of slack differs depending on the road conditions. Most ABS systems therefore compromise and set the target level to 0.1, but if a system could be developed to assess the condition of the road then much better performance could be achieved. A number of companies have researched using sensors on the car to detect these conditions, but it has been shown that the cost of this would be prohibitive.

Fuzzy logic provides an alternative solution, by using the sensors already in place to feedback the condition of the road once braking has started. Much as the human driver can estimate the road conditions by assessing the effects of applying the brakes, the fuzzy logic controller has a rule-base that uses the speed of the car, the speed of the wheels and the pressure of the brake fluid to calculate the change that should be made to the braking profile.

Research has shown that using only six fuzzy-logic rules gives a significant improvement in braking performance. As fuzzy logic systems have a high computational efficiency, this calculation can be performed easily within the time constraints of a successful control loop. This has been shown to enable the controller to adjust to changing road conditions during braking.

The technology also lends itself to developing decision support systems for areas where expert knowledge is required. One successful example in the financial sector is a system for differentiating fraudulent insurance claims from genuine ones, helping to make the efforts of fraud investigators more targeted. A system was developed using expert knowledge from a range of insurance companies and it has been demonstrated to automatically settle up to 85% of all insurance claims.

1.13 Artificial Neural Networks

In human brains large numbers of neurons with numerous connections to one another are able to calculate highly complex problems and react to new situations by learning from previous experiences. This is the basis of artificial neural networks (ANNs), although on a vastly more simplified level. The basic unit of an artificial neural network is an artificial neuron. It receives numerical inputs and produces an output based on these, and possibly on a small local memory store. These units are built up into networks that can learn from input data to perform complex calculations that are generalised to the problem and therefore tolerant of noisy data and able to work on previously un-encountered input data.

1.14 Artificial Neurons

Artificial neurons, or nodes, are built on very simple models of real neurons. In a real neuron the cell receives electrical inputs through its dendrites. These reach the cell body where they summate. If the summation of the inputs is over a threshold level for that neuron it fires off an electrical signal down its axon. This is replicated in the artificial neuron, whose 'dendrites' are numerical inputs, which are summated and then subjected to a threshold function to determine the output of the unit. The threshold function can be linear or more complex, for example sigmoid. Importantly the inputs are also given weights, to show how important they are to the function. These weights can be positive or negative and they are used when summating the inputs to determine the total level of activation of the unit.

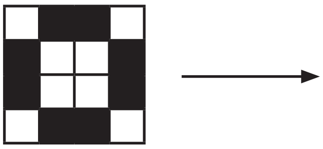
These weights and the threshold function itself can be adjusted to alter the output of the neuron. This ability allows the neuron to be trained to perform a specific function. Given a known set of data, perhaps experimentally obtained, the artificial neuron can be initialised with a number of inputs to match the number of inputs in the data, each with a random initial weight. When this neuron is presented with the first row of data in the set, the output can be compared to the actual measured output, to find the degree of error. This figure can be fed back to slightly adjust the weights on the inputs proportionally to their current weight.

This process continues until the neuron converges, which occurs when the error rate of the neuron, when classifying a complete pass through the data set, falls below a set tolerance. By themselves artificial neurons are only capable of calculating linearly separable problems, but when they are grouped into networks they can be trained to map any function. There are different configurations that can be used, two of which are explained below.

1.15 Self Organising Feature Maps

The first configuration is the Self Organising Feature Map (SOFM). These networks are used to find groupings within an input dataset. A 2D network of nodes is topologically mapped to the input data. Each node in the grid receives all the inputs, but all these connections are given initial random weights. When a set of data is presented to the system, each node will have a different level of activation. To train the SOFM, the 'winner' node is found, by finding the node whose input weights are closest to the input pattern. The weighting of the node's inputs are altered to make it respond to the input data set more strongly.

Similarly, the connections of the surrounding neurons in the grid are strengthened in proportion to how near each node is to the winner. This is repeated with different inputs in the data set until the weight changes are nominal.



	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	18
	0	1	1	0	1	0	0	1	1	0	0	1	0	1	1	0

The pixilated image is converted to a binary set of inputs

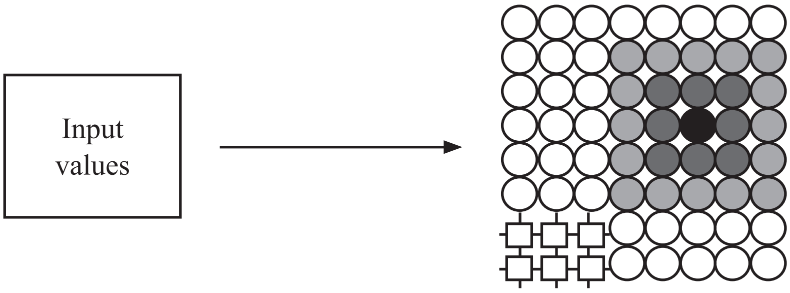


Fig. 1.10 Self Organising feature maps

The red node is the winner and its input weights and those of its neighbours are altered to increase the response to the input in the above case.

The effect of this training is to create topographical regions in the SOFM that respond strongly to certain cases of input data. For example if the SOFM were trained on pixilated alphanumeric characters, different regions would be activated when presented different characters. Characters that bear similarities to each other would have activation regions close to one another, for example O and C.

Within a region the site of greatest activation can be used to indicate how strongly similar the presented data is to the data the region has been trained to detect. Activation at or near the centre would be more certain than peripheral activation. This can be used to detect blurred or partially obscured characters, or to detect how similar a previously unseen character is to those shown in the training set. This method has been successfully used to read the licence plate numbers of cars on petrol forecourts from surveillance camera recordings, which are notoriously noisy sources of data.

1.16 Multi-layer Networks

Another common type of network is a multiple-layer network. These are made up of at least three layers. The input layer, which is the raw input data, connected to a hidden layer of neurons, which is further connected to an output layer of neurons. The number of hidden layers can be increased, and is one of the parameters that can be adjusted when trying to train a network

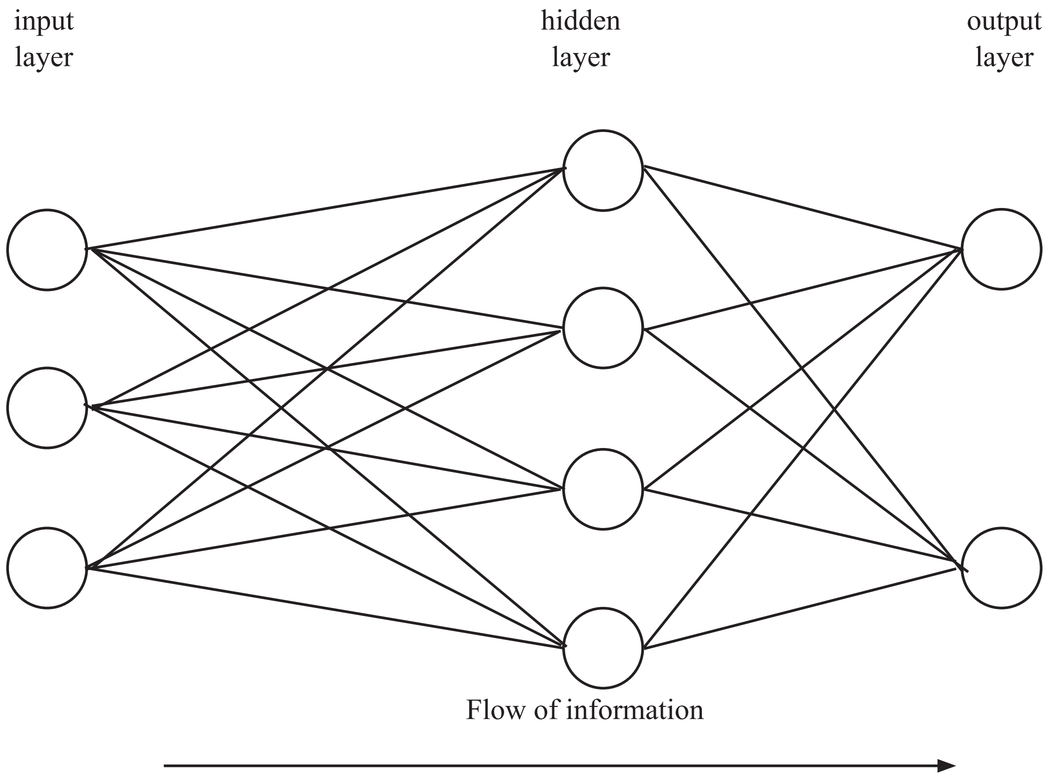


Fig. 1.11 Multilayer networks

More complicated algorithms are employed to train these networks, the most common of which is called the back-propagation algorithm. This works in a similar way to the algorithm for training a single neuron, but it propagates the weight changes back into the hidden layers. Networks like this can solve problems that are not linearly separable, and are very good at calculations that need patterns to be detected in noisy data, for example detecting the metabolic profile of a particular disease from the metabolites in urine samples.

1.17 Uses of Artificial Neural Networks

Neural networks can, in principle, compute any computable function. In particular neural networks are suited to problems where the input data is unavoidably going to be noisy or where there is a lot of available data that cannot be easily solved using mathematical techniques. An example of such a situation is the evaluation of medical images for pathological diseases. A recent breast cancer study used a neural network trained on ultrasound images of tumours that had been biopsied and diagnosed as malignant or benign.

Physicians collected the images by identifying the location of the tumours on the ultrasound image, selecting that section and then storing it for later use by the neural network. 140 training cases were stored, which were digitised and used to train the network, using the results of the biopsies to feedback the expected result. The fully trained network displayed a reasonably high level of accuracy and was helpful in guiding inexperienced operators to determine which patients should be given biopsies.

1.17.1 Bayesian Statistics

This technique is a statistical method that allows observed data to feed back into the process of statistical analysis. Where classical statistics takes sample data from a population and applies a statistical analysis method to the whole sample group, Bayesian statistical methods look at each case in turn and use the observed probabilities to update statistical assessment of the next cases.

This allows the analysis to be performed in an iterative manner, which can help reduce the number of cases needed to produce statistical results. If, for example, you wanted to analyse the dose-response curve of a drug, traditionally you would test the responses shown by a range of doses in large numbers of patients. Obviously the greater the number of patients who are involved in the trial, the more costly it is. Bayesian statistics can therefore offer an attractive alternative for assessing results. The initial drug dosages given can be analysed during the course of the trial and used to update the future doses given so that they are concentrated around the optimum dosage.

1.17.2 Hybrid Techniques

Each method described above can be used completely independently, or the techniques can be combined to exploit the strengths that each possess. There are numerous ways of hybridising these methods, of which three popular systems are explained below.

1.17.3 Neuro-fuzzy Systems

One powerful combination of soft computing techniques is the field of neuro-fuzzy systems. One of the main drawbacks of neural networks is that they are 'black boxes'. Data goes in, a calculation is performed and the answer is produced, but it is difficult to find out exactly what the calculation is doing or why. Fuzzy logic systems do not suffer from this problem, their rule sets can always be 'translated' into easily understandable rules, but they are not capable of learning in the same way that neural networks are. This usually means that the rules and membership functions used need to be written and tuned by hand.

When expert knowledge can be used to write these systems then this task is relatively simple, but this is not always possible to find experts in the problem domain. Neuro-fuzzy systems address both these problems by applying the learning algorithms of multi-layer neural networks to adjust the distribution of the fuzzy logic sets used in a fuzzy logic system. This means that the fuzzy logical system can be created with no expert knowledge of the system that is being modelled, and by supplying the network with good training data; it will learn rules that can be used to model the system in general. These rules can be interpreted easily, making the system much more transparent and understandable.

This branch of Soft Computing has expanded rapidly in recent years, and it has been used in a wide variety of applications. In one such example a neuro-fuzzy system was trained to predict the next 48 hours of electricity demand in Victoria, Australia. When tested on previously unseen data, this system predicted the demand with an error rate of only 0.0092.

1.17.4 Genetic Fuzzy Systems

Another way of tuning fuzzy systems is to use Evolutionary Computing techniques to evolve the rule set and membership functions. This is particularly useful when a system cannot be trained using expected or previously measured output values, for example when a controller for a dynamic system is being developed. The controllers can, however, be judged on how well they perform, and this can be used as the fitness function in an evolutionary algorithm.

Much like the neuro-fuzzy system, the various aspects of the fuzzy logical system are parameterised. These aspects include the membership functions and information on the fuzzy rule set. This information is then encoded into a string or strings that represent a candidate's genetic material. Genetic algorithms can then be applied to evolve the fuzzy system until an optimum solution is found. The combination of these methodologies is a relatively recent development, but it has already been used in a number of applications. One success story is a system for diagnosing the cause of structural cracks in stone buildings. This is an area which requires a great deal of expert knowledge and which relies on multiple observable features.

The fuzzy logic system was used to take 42 of these observable features, and used fuzzy rules to determine the root cause from 6 possibilities. As the number of parameters in the system was so large, manually tuning the membership functions and the weights given to the rules to give the optimum results would have been difficult. For this reason a genetic algorithm was used, which encoded these values and evolved them, using the accuracy of the resulting system as the fitness function to guide the evolution. Once this tuning was complete the final system showed a good match with actual cases, showing an error rate of less than 4%.

1.18 Using Neural Networks with Genetic Algorithms

One of the most important parts of a genetic algorithm is the specification of a good fitness function. If this is badly implemented then the evolved solutions will not be adapted to the task required. In many cases it is difficult to write a computationally efficient fitness function, as assessing the qualities of the individual designs can be an incredibly complex or non-linear problem. In cases like these, neural networks can be used to assess the fitness of solutions by training them on experimentally obtained data.

An example of this technique can be seen in the steel industry. Predicting the mechanical properties of steel alloys given their metal composition and tempering temperature is not very easy. Researchers at Sheffield University trained a neural network on experimental data to produce an effective predictor of alloy properties. This network was then used as the fitness function in a genetic algorithm to evolve new alloy designs with specific pre-subscribed properties.

The genetic algorithm produced reliable and practical solutions to the design problem, which were verified later by a number of metallurgists.

1.19 When Should you Use Soft Computing?

As shown by the examples described above, soft computing can be used to address a very wide range of problems in all industries and business sectors. In general though, Soft Computing is a good option for complex systems, where:

- The system is non-linear, time-variant or ill defined.
- The variables are continuous.
- A mathematical model is either too difficult to encode, does not exist or is too complicated and expensive to be evaluated.
- There are noisy or numerous inputs.
- An expert is available who can outline the rules-of-thumb that should determine the system behaviour.

General areas that need these kinds of systems are:

- Classification
- Optimisation
- Data mining
- Prediction
- Control
- Scheduling
- Decision support or auto-decision making

The most appropriate technique to use depends on the type of problem and the available data. Neural Networks are good at classification, data-mining and prediction systems, where there is lots of available potentially noisy input data, which either needs to be classified into groups or which needs to be mapped to an expected output.

They do, however, suffer from a lack of transparency, as the calculation is encoded in the weights and thresholds of multiply connected networks. Evolutionary Computing techniques are adapted to optimisation and design problems, where a good solution can be recognised, but where there is no “correct” answer. This technique is essentially an efficient search technique that deals well with the problem of getting “stuck” with a sub-optimal solution. The main difficulty with the technique is defining how the solutions should be encoded and assessed for their fitness. Badly written fitness functions will result in answers that do not fit the problem being solved.

Fuzzy Logic systems are best suited to decision making and control systems that have “rules-of-thumb” that cannot be translated to hard mathematical formulae. These rules are used to perform a logical, non-linear mapping between inputs and outputs, which simulates the decision-making processes in humans. These systems rely very heavily on being able to determine rules that accurately describe the system. They also need the membership functions for the fuzzy sets used to be tuned correctly, which can also be difficult due to their non-linear nature.

With such a large topic, it has only really been possible to scratch the surface of this technology and for further information the reader is referred to the websites listed below and to the numerous books now available on this area of computing.

Summary

- Soft computing (SC) is a branch, in which, it is tried to build intelligent and wiser machines. Intelligence provides the power to derive the answer and not simply arrive to the answer.
- Intuitive consciousness/wisdom is also one of the important area in the soft computing, which is always cultivated by meditation.
- Soft computing is an umbrella term for a collection of computing techniques.
- Evolutionary computing uses the Darwinian principle of ‘survival of the fittest’ to evolve optimum solutions to problems.
- The fitness function evaluates the solution and returns a numerical answer, which rates the solution’s suitability.
- Re-combination/cross-over involves combining the ‘genetic’ information from two parent solutions to produce an offspring solution
- The mutation operator usually performs point mutations.
- Genetic programming works in a very similar way to genetic algorithms.
- Usually re-combination involves selecting a sub-tree at random from each parent, and then swapping them.
- Evolution strategies were developed by researchers at the Technical Institute of Berlin, who wanted to automate the design of aerodynamic jointed plates.
- Evolutionary computing has already proved itself as a useful tool in a range of industries.
- Due to the complexity of the search space, conventional rule-based or numerical methods tend to be unsuccessful.
- In Boolean logic, the only values allowed are completely true and completely false.
- A fuzzy inference system uses fuzzy logical rules to produce output values from input values.
- The degree to which the output variable belongs to that fuzzy set is defined by the antecedent.
- Fuzzy logic provides an alternative solution, by using the sensors already in place to feedback the condition of the road once braking has started.
- In human brains large numbers of neurons with numerous connections to one another are able to calculate highly complex problems and react to new situations by learning from previous experiences.
- Artificial neurons, or nodes, are built on very simple models of real neurons. In a real neuron the cell receives electrical inputs through its dendrites.
- One powerful combination of soft computing techniques is the field of neuro-fuzzy systems.

References

- Chaturvedi, K. D., 2008. *Soft Computing: Techniques and Its Applications in Electrical Engineering*, Springer.
- Konar, A., 2000. *Artificial Intelligence and Soft Computing: Behavioral and Cognitive Modeling of the Human Brain*, Volume 1, CRC Press.
- Chakraborty, C. R., *Introduction Basics of Soft Computing*, [Online] Available at: <http://www.myreaders.info/01_Introduction_to_Soft_Computing.pdf> [Accessed 18 July 2012].
- Bower, A., *SOFT COMPUTING*, [Online] Available at: <http://www.tessella.com/wp-content/uploads/2008/05/softcomputing.pdf> [Accessed 18 July 2012].
- Kala, R., 2012. *Soft computing lecture - hour 1*, [Video Online] Available at: <http://www.youtube.com/watch?v=DzidvDNE-Ik> [Accessed 18 July 2012].
- Kala, R., 2012. *Soft computing lecture - hour 2*, [Video Online] Available at: <http://www.youtube.com/watch?v=O5kZOqw_tfc&feature=relmfu> [Accessed 18 July 2012].

Recommended Reading

- Akerkar, R.& Sajja, P., 2010. *Knowledge-Based Systems*, Jones & Bartlett Learning.
- Varshney, M. A. R., *An Introduction to Soft Computing*, A. B. Publication.
- Gupta, M. M., 2002. *Soft Computing and Intelligent Systems: Theory and Applications*, Elsevier.

Self Assessment

1. Soft computing (SC) is a branch, in which, it is tried to build _________and ________machines.
    a. intelligent, wiser
    b. dimensions, complexity
    c. fuzziness handling, capability increase
    d. freedom, work

2. ________ provides the power to derive the answer and not simply arrive to the answer.
    a. Human beings
    b. Intelligence
    c. Hierarchy
    d. Complexity

3. Which of the following statements is false?
    a. Wisdom is also one of the important area in the soft computing, which is always cultivated by meditation.
    b. Soft computing is an emerging collection of methodologies, which aim to exploit tolerance for imprecision.
    c. Parallel computer architectures not simulate biological processes, they can perform mapping from inputs to the outputs faster.
    d. Computational load and consequent increase of computation speeds that permit more robust system.

4. Soft computing is a ________term for a collection of computing techniques.
    a. computation
    b. umbrella
    c. uncertainty
    d. tractability

5. Which of the following is not guiding principle of soft computing?
    a. Exploit the tolerance for imprecision
    b. Uncertainty and partial truth to achieve tractability
    c. Robustness
    d. High solution cost

6. ____________ uses the Darwinian principle of 'survival of the fittest' to evolve optimum solutions to problems.
    a. Soft computing
    b. Biological processes
    c. Evolutionary computing
    d. Evolutionary techniques

7. Fuzzy set output now needs to be converted into a crisp output value in order to be of any use a process called ________.
    a. Candidate solutions
    b. De-fuzzification
    c. Genetic programming
    d. Computer programs

8. The basic unit of an artificial neural network is an _________.
    a. artificial neuron
    b. artificial numerical
    c. threshold function
    d. sigmoid

9. Which of the following statements is false?
    a. Nodes are built on very simple models of real neurons.
    b. Human brains large numbers of neurons with numerous connections to one another.
    c. Evolutionary computing has already not proved itself as a useful tool in a range of industries.
    d. Evolutionary programming the only genetic operator used is mutation.

10. _________technique is a statistical method that allows observed data to feed back into the process of statistical analysis.
    a. Neural networks
    b. Bayesian Statistics
    c. Hybrid Techniques
    d. Optimum dosage

Chapter II

Artificial Intelligence and Agents

Aim

The aim of this chapter is to:

- introduce artificial intelligence and its agents
- elucidate various disciplines of artificial intelligence
- explain intelligent tutorial system

Objectives

The objectives of this chapter are to:

- explain role of knowledge representation in solving problems
- explicate anytime algorithm to calculate computation time
- elucidate solution quality as a function of time for an anytime algorithm

Learning outcome

At the end of this chapter, you will be able to:

- understand agents in different environment
- identify the typical laboratory environment for various problems
- solve the problem using different AI solutions

2.1 Introduction

John McCarthy, coined the term Artificial Intelligence (AI) in the year 1956. He defines it as “the science and engineering of making intelligent machines. Artificial Intelligence is the science and engineering of making intelligent machines, especially intelligent computer programs. It is related to the similar task of using computers to understand human intelligence, but AI does not have to confine itself to methods that are biologically observable.

Artificial Intelligence has been a recognised discipline for over 50 years. This field has built on though and imagination. We have the tools to test hypotheses about the nature of thought itself, as well as to solve practical problems. Many scientific and engineering problems have solved by AI. Many practical applications are currently deployed and the potential exists for an almost unlimited number of future applications. In this book we will see the principles that underlie intelligent computational agents.

2.2 What is Artificial Intelligence?

Artificial intelligence, or AI, is the field that studies the synthesis and analysis of computational agents that act intelligently.

The description of the above definition tells that:
An agent is something that acts in an environment. An agent is a means by which something is done or caused; instrument. Agents include worms, dogs, thermostats, airplanes, robots, humans, companies, and countries.

An agent acts intelligently when

- what it does is appropriate for its circumstances and its goals?
- it is flexible to changing environments and changing goals
- it learns from experience, and
- it makes appropriate choices given its perceptual and computational limitations

An agent has only a finite memory and it has limited time to act. It typically cannot observe the state of the world directly. A computational agent is an agent whose decisions about its actions can be explained in terms of computation. That is, the decision can be broken down into primitive operation that can be implemented in a physical device. This computation can take many forms in different mediums. In humans this computation is carried out in “wetware”; in computers it is carried out in “hardware.” There are some agents that are possibly not computational, such as the wind and rain which erodes a landscape, it is an open question whether all intelligent agents are computational or not.

To make intelligent behaviour possible in natural or artificial systems we have to understand the principles of AI. This is done by

- the analysis of natural and artificial agents
- formulating and testing hypotheses about what it takes to construct intelligent agents
- designing, building, and experimenting with computational systems that perform tasks commonly viewed as requiring intelligence

As part of science, researchers build empirical systems to test hypotheses or to explore the space of possibilities. These are quite distinct from applications that are built to be useful for an application domain.

Note that the definition is not for intelligent thought. We are only interested in thinking intelligently insofar as it leads to better performance. The role of thought is to affect action.

2.2.1 Artificial and Natural Intelligence

Artificial intelligence (AI) may be interpreted as the opposite of real intelligence. This is the established name for the field, but the term "artificial intelligence" is a source of much confusion because artificial intelligence. For any phenomenon, we can distinguish real phenomenon versus fake phenomenon, where the fake is non-real. We can also distinguish natural versus artificial. Natural means occurring in nature and artificial means made by people.

Consider an example to understand this; a tsunami is a natural calamity happens in an ocean which is caused by an earthquake or a landslide. Natural tsunamis occur from time to time. And an artificial tsunami made by people, such as explosion of a bomb in the ocean, also causes tsunami. This is the difference which we can see in natural and artificial phenomenon.

It is debatable that intelligence is different, we cannot have fake intelligence. If an agent behaves intelligently, it is intelligent. It is only the external behaviour that defines intelligence; acting intelligently is being intelligent. Thus, artificial intelligence, if and when it is achieved, will be real intelligence created artificially.

This idea of intelligence being defined by external behaviour was the motivation for a test for intelligence designed by Turing (1950), which has become known as the Turing test. The Turing test consists of an imitation game where an interrogator can ask a witness, via a text interface, any question. If the interrogator cannot distinguish the witness from a human, the witness must be intelligent. Example below shows the possible dialog that Turing suggested. An agent that is not really intelligent could not fake intelligence for arbitrary topics.

Interrogator:

In the first line of your sonnet which reads "Shall I compare thee to a summer's day," would not "a spring day" do as well or better?

Witness:
It wouldn't scan.

Interrogator:
How about "a winter's day," That would scan all right.

Witness:
Yes, but nobody wants to be compared to a winter's day.

Interrogator:
Would you say Mr. Pickwick reminded you of Christmas?

Witness:
In a way.

Interrogator:
Yet Christmas is a winter's day, and I do not think Mr. Pickwick would mind the comparison.

Witness:
I don't think you're serious. By a winter's day one means a typical winter's day, rather than a special one like Christmas.

Unfortunately, Turing test provide a test for how to recognize intelligence, but it does not provide a way to get there; trying each year to fake it does not seem like a useful avenue of research.

The obvious naturally intelligent agent is the human being. There can be debate between some people. Some people would say worms, insects, or bacteria are intelligent, but more people would say that dogs, whales, or monkeys are intelligent. One class of intelligent agents that may be more intelligent than humans is the class of organizations. Ant colonies are a prototypical example of organizations. Each individual ant may not be very intelligent, but an ant colony can act more intelligently than any individual ant. The colony can discover food and exploit it very effectively as well as adapt to changing circumstances.

Similarly, companies can develop, manufacture, and distribute products where these skills are required. Now day's modern computers are more complicated than any human can understand, yet they are manufactured daily by organizations of humans. Human society viewed as an agent is arguably the most intelligent agent known.

We can consider that human intelligence comes from three main sources:

Biology:
Humans have evolved into adaptable animals that can survive in various habitats.

Culture:
Culture provides not only language, but also useful tools, useful concepts, and the wisdom that is passed from parents and teachers to children.

Life-long learning:
Humans learn throughout their life and accumulate knowledge and skills.

2.3 A Brief History of AI

About 400 years ago people started to write about the nature of thought and reason. Hobbes, who has been described by Haugeland as the "Grandfather of AI," espoused the position that thinking was symbolic reasoning like talking out loud or working out an answer with pen and paper. The idea of symbolic reasoning was further developed by Descartes, Pascal, Spinoza, Leibniz, and others who were pioneers in the philosophy of mind.

Development of computers briefs the idea of symbolic operations. The first general-purpose computer designed was the Analytical Engine by Charles Babbage. In the early part of the 20th century, there was much work done on understanding computation. Several models of computation were proposed, including the Turing machine by Alan Turing, a theoretical machine that writes symbols on an infinitely long tape, and the lambda calculus of Church, which is a mathematical formalism for rewriting formulas. It can be shown that these very different formalisms are equivalent in that any function computable by one is computable by the others. This leads to the Church-Turing thesis:

Any effectively computable function can be carried out on a Turing machine. This thesis says that all computation can be carried out on a Turing machine or one of the other equivalent computational machines. The Church-Turing thesis cannot be proved but it is a hypothesis that has stood the test of time. No one has built a machine that has carried out computation that cannot be computed by a Turing machine. There is no evidence that people can compute functions that are not Turing computable. An agent's actions are a function of its abilities, its history, and its goals or preferences. This provides an argument that computation is more than just a metaphor for intelligence; reasoning is computation and computation can be carried out by a computer.

Once real computers were built, some of the first applications of computers were AI programs. For example, Samuel (1959) built a checkers program in 1952 and implemented a program that learns to play checkers in the late 1950s. Newell and Simon (1956) built a program, Logic Theorist that discovers proofs in propositional logic. In addition to that for high-level symbolic reasoning, there was also much work on low-level learning inspired by how neurons work. McCulloch and Pitts (1943) showed how a simple thresholding "formal neuron" could be the basis for a Turing-complete machine. The first learning for these neural networks was described by Minsky (1952). One of the early significant works was the Perceptron of Rosenblatt (1958).

The work on neural networks went into decline for a number of years after the 1968 book by Minsky and Papert (1988), which argued that the representations learned were inadequate for intelligent action.

These early programs concentrated on learning and search as the foundations of the field. It became apparent early that one of the main problems was how to represent the knowledge needed to solve a problem. Before learning, an agent must have an appropriate target language for the learned knowledge. There have been many proposals for representations from simple feature-based representations to complex logical representations of [McCarthy and Hayes (1969)] and many in between such as the frames of Minsky (1975).

During the 1960s and 1970s, success was had in building natural language understanding systems in limited domains. For example, the STUDENT program of Daniel Bobrow (1967) could solve high school algebra problems expressed in natural language. [Winograd (1972)]'s SHRDLU system could, using restricted natural language, discuss and carry out tasks in a simulated blocks world. CHAT-80 [Warren and Pereira (1982)] could answer geographical questions placed to it in natural language. The example below shows some questions that CHAT-80 answered based on a database of facts about countries, rivers, and so on. All of these systems could only reason in very limited domains using restricted vocabulary and sentence structure.

Does Afghanistan border China?
What is the capital of Upper_Volta?
Which country's capital is London?
Which is the largest African country?
How large is the smallest American country?
What is the ocean that borders African countries and that borders Asian countries?
What are the capitals of the countries bordering the Baltic?
How many countries does the Danube flow through?
What is the total area of countries south of the Equator and not in Australasia?
What is the average area of the countries in each continent?
Is there more than one country in each continent?
What are the countries from which a river flows into the Black_Sea?
What are the continents no country in which contains more than two cities whose population exceeds 1 million?
Which country bordering the Mediterranean borders a country that is bordered by a country whose population exceeds the population of India?
Which countries with a population exceeding 10 million border the Atlantic?

During the 1970s and 1980s, there was a large body of work on expert systems, where the aim was to capture the knowledge of an expert in some domain so that a computer could carry out expert tasks. For example, DENDRAL [Buchanan and Feigenbaum (1978)], developed from 1965 to 1983 in the field of organic chemistry, proposed plausible structures for new organic compounds. MYCIN [Buchanan and Shortliffe (1984)], developed from 1972 to 1980, diagnosed infectious diseases of the blood, prescribed antimicrobial therapy, and explained its reasoning. The 1970s and 1980s were also a period when AI reasoning became widespread in languages such as Prolog [Colmerauer and Roussel (1996)][Kowalski (1988)].

During the 1990s and the 2000s there was great growth in the sub disciplines of AI such as perception, probabilistic and decision-theoretic reasoning, planning, embodied systems, machine learning, and many other fields. There has also been much progress on the foundations of the field; these form the foundations of this book.

2.3.1 Relationship to Other Disciplines

AI is a very much expanding discipline. Other disciplines such as philosophy, neurobiology, evolutionary biology, psychology, economics, political science, sociology, anthropology, control engineering, and many more have been studying intelligence from the decades.

The science of AI could be described as "synthetic psychology," "experimental philosophy," or "computational epistemology". Epistemology generally is the study of knowledge. We can say that AI provides the way to study the old problem of the nature of knowledge and intelligence, but with a more powerful experimental tool. Instead of being able to observe only the external behaviour of intelligent systems, as philosophy, psychology, economics, and sociology have traditionally been able to do, AI researchers experiment with executable models of intelligent behaviour. Most important, such models are open to inspection, redesign, and experiment in a complete and rigorous way. Modern computers provide a way to construct the models about which philosophers have only been able to theorize. AI researchers can experiment with these models as opposed to just discussing their abstract properties. AI theories can be empirically grounded in implementation. Moreover, we are often surprised when simple agents exhibit complex behaviour. We would not have known this without implementing the agents.

AI takes an approach analogous to that of aerodynamics. AI researchers are interested in testing general hypotheses about the nature of intelligence by building machines that are intelligent and that do not necessarily mimic humans or organizations. This also offers an approach to the question, "Can computers really think?" by considering the analogous question, "Can airplanes really fly?"

AI is intimately linked with the discipline of computer science. Although there are many non-computer scientists who are doing AI research, much, if not most, AI research is done within computer science departments. This is appropriate because the study of computation is central to AI. It is essential to understand algorithms, data structures, and combinatorial complexity to build intelligent machines. It is also surprising how much of computer science systems started as a spinoff from AI, from timesharing to computer algebra.

Finally, AI can be seen as coming under the umbrella of cognitive science. Cognitive science links various disciplines that study cognition and reasoning, from psychology to linguistics to anthropology to neuroscience. AI distinguishes itself within cognitive science by providing tools to build intelligence rather than just studying the external behaviour of intelligent agents or dissecting the inner workings of intelligent systems.

2.4 Agents Situated in Environments

AI is about practical reasoning in order to do something. A coupling of perception, reasoning, and acting comprise an agent. An agent acts in an environment. An agent's environment may well include other agents. An agent together with its environment is called a world.

If the environment is a physical setting an agent could be considered as a coupling of a computational engine with physical sensors and actuators, called a robot. It could be the advice to a computer from an expert system with a human who provides perceptual information and carries out the task. An agent could be a program that acts in a purely computational environment like a software agent.

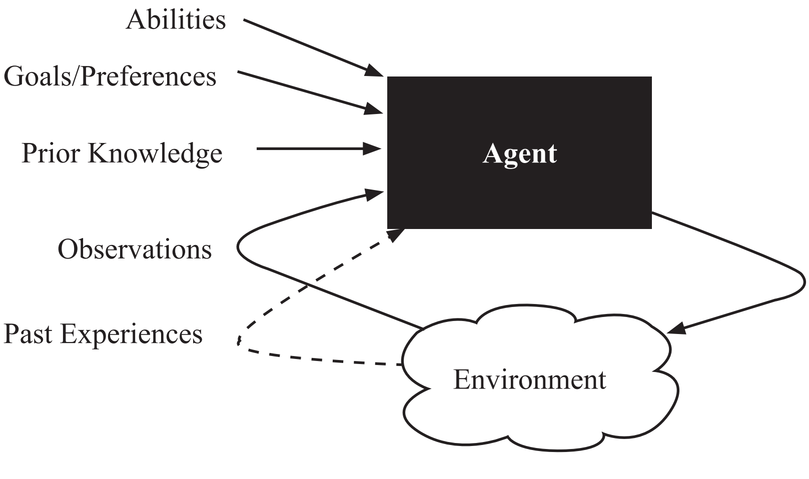


Fig. 2.1 An agent interacting with an environment

Figure above shows the inputs and outputs of an agent. The agents activities depend on its:

- prior knowledge about the agent and the environment
- history of interaction with the environment, which is composed of observations of the current environment and past experiences of previous actions and observations, or other data, from which it can learn
- goals that it must try to achieve or preferences over states of the world
- abilities, which are the primitive actions it is capable of carrying out

Two distinct agents with the same knowledge, history, abilities, and goals should do the same thing. If any one of these changes it can result in different actions. Each agent has some internal state about its environment and itself. It also has goals to achieve, ways to act in the environment to achieve those goals, and various means to modify its beliefs by reasoning, perception, and learning.

2.5 Knowledge Representation

Knowledge is a progression that starts with data which is of limited services. It is a general term. After processing the data or by organising or analysing the data, we understand what the data means, and this data becomes information then.

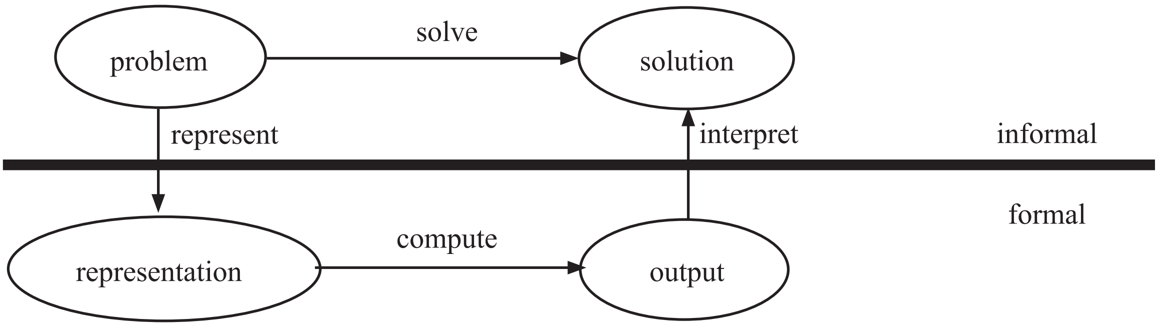


Fig. 2.2 The role of representations in solving problems

The general framework for solving problems by computer is given in Figure 2.2. To solve a problem, the designer of a system must

- flesh out the task and determine what constitutes a solution
- represent the problem in a language with which a computer can reason
- use the computer to compute an output, which is an answer presented to a user or a sequence of actions to be carried out in the environment
- interpret the output as a solution to the problem.

Knowledge can be used to solve problems in its own domain. To solve many problems we require much knowledge that is the information about a domain, and this knowledge must be represented in the computer. While designing a program or to solve the problems, the knowledge should be defined first. A representation scheme is the form of the knowledge that is used in an agent. A representation of some piece of knowledge is the internal representation of the knowledge. A representation scheme specifies the form of the knowledge. A knowledge base is the representation of all of the knowledge that is stored by an agent.

A good representation scheme is a compromise among many competing objectives. A representation should be

- rich enough to express the knowledge needed to solve the problem
- as close to the problem as possible
- it should be compact, natural, and maintainable
- It should be easy to see the relationship between the representation and the domain being represented

- It should be amenable to efficient computation, which usually means that it is able to express features of the problem that can be exploited for computational gain and able to trade off accuracy and computation time
- It should be able to be acquired from people, data and past experiences

A good representation scheme makes easy to determine whether the knowledge represented is correct. A small change in the problem should result in a small change in the representation of the problem.

Many of the representation schemes start with some of the above objectives and are then expanded to include the other objectives. Some representation schemes are designed for learning and then expanded to allow richer problem solving and inference abilities.

Some questions should be keep in mind while giving a task or solving a problem. For example, what is a solution to the problem? How good must a solution be? How can the problem be represented? What distinctions in the world are needed to solve the problem? What specific knowledge about the world is required? How can an agent acquire the knowledge from experts or from experience? How can the knowledge be debugged, maintained, and improved? How can the agent compute an output that can be interpreted as a solution to the problem? Is worst-case performance or average-case performance the critical time to minimize? Is it important for a human to understand how the answer was derived?

2.5.1 Defining a Solution

Problems are not well specified. If we ask a trading agent to find out all the information about resorts that may have health issues, we do not want the agent to return the information about all resorts, even though all of the information requested is in the result. However, if the trading agent does not have complete knowledge about the resorts, returning all of the information may be the only way for it to guarantee that all of the requested information is there. AI is motivated by reasonable reasoning; we want the computer to be able to make commonsense conclusions about the unstated assumptions.

There are four common classes of solutions:

Optimal solution

An optimal solution to a problem is one that is the best solution according to some measure of solution quality. This measure is typically specified as an ordinal, where only the order matters. However, in some situations, such as combining multiple criteria or reasoning under uncertainty, the relative magnitudes matters. An example of an ordinal measure is for the robot to take out as much trash as possible; the more trash it can take out, the better. As an example of a cardinal measure, you may want the delivery robot to take as much of the trash as possible to the garbage can, minimizing the distance travelled, and explicitly specify a trade-off between the effort required and the proportion of the trash taken out. It may be better to miss some trash than to waste too much time. One general cardinal measure of desirability, known as utility, is used in decision theory.

Satisfying solution

A satisfying solution is one that is good enough according to some description of which solutions are adequate. For example, a person may tell a robot that it must take all of trash out, or tell it to take out three items of trash.

Approximately optimal solution

One of the advantages of a cardinal measure of success is that it allows for approximations. An approximately optimal solution is one whose measure of quality is close to the best that could theoretically be obtained. Typically agents do not need optimal solutions to problems; they only must get close enough. For example, the robot may not need to travel the optimal distance to take out the trash but may only need to be within, say, 10% of the optimal distance.

For some problems, it is much easier computationally to get an approximately optimal solution than to get an optimal solution. However, for other problems, it is just as difficult to guarantee finding an approximately optimal solution as it is to guarantee finding an optimal solution. Some approximation algorithms guarantee that a solution is within some range of optimal, but for some algorithms no guarantees are available.

Probable solution

A probable solution is one that, even though it may not actually be a solution to the problem, is likely to be a solution. This is one way to approximate, in a precise manner, a satisfying solution. For example, in the case where the delivery robot could drop the trash or fail to pick it up when it attempts to, you may need the robot to be 80% sure that it has picked up three items of trash. Often you want to distinguish the false-positive error rate from the false-negative error rate. Some applications are much more tolerant of one of these errors than of the other.

2.5.2 Representations

According to problems and the nature of a solution, we can have some requirements. Also the problem must be represented so that a computer can solve it. Computers and human minds are examples of physical symbol systems. A symbol is a meaningful pattern that can be manipulated. Examples of symbols are written words, sentences, gestures, marks on paper, or sequences of bits. A symbol system creates copies, modifies, and destroys symbols. The term physical is used, because symbols in a physical symbol system are physical objects that are part of the real world, even though they may be internal to computers and brains. They may also need to physically affect action or motor control.

AI rests on the physical symbol system hypothesis of Newell and Simon (1976). A physical symbol system gives the necessary and sufficient means for general intelligent action. This is a strong hypothesis that any intelligent agent is necessarily a physical symbol system. It also means that a physical symbol system is all that is needed for intelligent action; there is no magic or an as-yet-to-be-discovered quantum phenomenon required. It does not imply that a physical symbol system does not need a body to sense and act in the world. The physical symbol system hypothesis is an empirical hypothesis that, like other scientific hypotheses, is to be judged by how well it fits the evidence, and what alternative hypotheses exist. Indeed, it could be false.

An intelligent agent manipulates symbols to produce action. Many of these symbols are used to refer to stuffs in the world. Other symbols may be useful concepts that may or may not have external meaning. Yet other symbols may refer to internal states of the agent.

An agent can use physical symbol systems to model the world. A model of a world is a representation of the specifics of what is true in the world or of the dynamic of the world. The world does not have to be modelled at the most detailed level to be useful. All models are abstractions; they represent only part of the world and leave out many of the details. An agent can have a very simplistic model of the world, or it can have a very detailed model of the world. The level of abstraction provides a partial ordering of abstraction. A lower-level abstraction includes more details than a higher-level abstraction. An agent can have multiple, even contradictory, models of the world. The models are judged not by whether they are correct, but by whether they are useful.

Consider an example, a delivery robot can model the environment at a high level of abstraction in terms of rooms, corridors, doors, and obstacles, ignoring distances, its size, the steering angles needed, the slippage of the wheels, the weight of parcels, the details of obstacles, the political situation in Canada, and virtually everything else. The robot could model the environment at lower levels of abstraction by taking some of these details into account. Some of these details may be irrelevant for the successful implementation of the robot, but some may be crucial for the robot to succeed. For example, in some situations the size of the robot and the steering angles may be crucial for not getting stuck around a particular corner. In other situations, if the robot stays close to the centre of the corridor, it may not need to model its width or the steering angles.

Choosing an appropriate level of abstraction is difficult because

- A high-level description is easier for a human to specify and understand
- A low-level description can be more accurate and more predictive. Often high-level descriptions abstract away details that may be important for actually solving the problem
- The lower the level, the more difficult it is to reason with. This is because a solution at a lower level of detail involves more steps and many more possible courses of action exist from which to choose.

- You may not know the information needed for a low-level description. For example, the delivery robot may not know what obstacles it will encounter or how slippery the floor will be at the time that it must decide what to do.

Biological systems, and computers, can be described at multiple levels of abstraction. At successively lower levels are the neural level, the biochemical level (what chemicals and what electrical potentials are being transmitted), the chemical level (what chemical reactions are being carried out), and the level of physics (in terms of forces on atoms and quantum phenomena). The following are two levels that seem to be common to both biological and computational entities:

The knowledge level

It is a level of abstraction that considers what an agent knows and believes and what its goals are. The knowledge level considers what an agent knows, but not how it reasons. For example, the delivery agent's behaviour can be described in terms of whether it knows that a parcel has arrived or not and whether it knows where a particular person is or not. Both human and robotic agents can be described at the knowledge level. At this level, you do not specify how the solution will be computed or even which of the many possible strategies available to the agent will be used.

The symbol level

It is a level of description of an agent in terms of the reasoning it does. To implement the knowledge level, an agent manipulates symbols to produce answers. Many cognitive science experiments are designed to determine what symbol manipulation occurs during reasoning. Note that whereas the knowledge level is about what the agent believes about the external world and what its goals are in terms of the outside world, the symbol level is about what goes on inside an agent to reason about the external world.

2.5.3 Reasoning and Acting

The manipulation of symbols to produce action is called reasoning. One way that AI representations differ from computer programs in traditional languages is that an AI representation typically specifies what needs to be computed, not how it is to be computed. We might specify that the agent should find the most likely disease a patient has, or specify that a robot should get coffee, but not give detailed instructions on how to do these things. Much AI reasoning involves searching through the space of possibilities to determine how to complete a task.

In deciding what an agent will do, there are three aspects of computation that must be distinguished:

- The computation that goes into the design of the agent.
- The computation that the agent can do before it observes the world and needs to act.
- The computation that is done by the agent as it is acting.
- Design time reasoning is the reasoning that is carried out to design the agent. It is carried out by the designer of the agent, not the agent itself.
- Offline computation is the computation done by the agent before it has to act. It can include compilation and learning. Offline, the agent takes background knowledge and data and compiles them into a usable form called a knowledge base. Background knowledge can be given either at design time or offline.
- Online computation is the computation done by the agent between observing the environment and acting in the environment. A piece of information obtained online is called an observation. An agent typically must use both its knowledge base and its observations to determine what to do.

It is important to distinguish between the knowledge in the mind of the designer and the knowledge in the mind of the agent. Consider the extreme cases:

- At one extreme is a highly specialized agent that works well in the environment for which it was designed, but it is helpless outside of this niche. The designer may have done considerable work in building the agent, but the agent may not need to do very much to operate well. An example is a thermostat. It may be difficult to design a thermostat so that it turns on and off at exactly the right temperatures, but the thermostat itself does not have to do much computation. These very specialized agents do not adapt well to different environments or to changing goals.
- At the other extreme is a very flexible agent that can survive in arbitrary environments and accept new tasks at run time. Simple biological agents such as insects can adapt to complex changing environments, but they cannot carry out arbitrary tasks. Designing an agent that can adapt to complex environments and changing goals is a major challenge. The agent will know much more about the particulars of a situation than the designer. Even biology has not produced many such agents. Humans may be the only extant example, but even humans need time to adapt to new environments.

Agents building pursue two broad strategies explain below:

- The first is to simplify environments and build complex reasoning systems for these simple environments. For example, factory robots can do sophisticated tasks in the engineered environment of a factory, but they may be hopeless in a natural environment. Much of the complexity of the problem can be reduced by simplifying the environment. This is also important for building practical systems because many environments can be engineered to make them simpler for agents.
- The second strategy is to build simple agents in natural environments. This is inspired by seeing how insects can survive in complex environments even though they have very limited reasoning abilities. Researchers then make the agents have more reasoning abilities as their tasks become more complicated.

One of the advantages of simplifying environments is that it may enable us to prove properties of agents or to optimize agents for particular situations. Proving properties or optimization typically requires a model of the agent and its environment. The advantage of building agents for complex environments is that these are the types of environments in which humans live and want our agents to live.

2.6 Dimensions of Complexity

Agents acting in environments range in complexity from thermostats to companies with multiple goals acting in competitive environments. A number of dimensions of complexity exist in the design of intelligent agents. These dimensions may be considered separately but must be combined to build an intelligent agent. These dimensions define a design space of AI; different points in this space can be obtained by varying the values of the dimensions.

Here we present nine dimensions: modularity, representation scheme, planning horizon, sensing uncertainty, effect uncertainty, preference, number of agents, learning, and computational limits. These dimensions give a coarse division of the design space of intelligent agents. There are many other design choices that must be made to build an intelligent agent.

2.6.1 Modularity

The first dimension is the level of modularity. Modularity is the extent to which a system can be decomposed into interacting modules that can be understood separately. Modularity is important for reducing complexity. It is apparent in the structure of the brain, serves as a foundation of computer science, and is an important part of any large organization.

Modularity is typically expressed in terms of a hierarchical decomposition. For example, a human's visual cortex and eye constitute a module that takes in light and perhaps higher-level goals and outputs some simplified description of a scene. Modularity is hierarchical if the modules are organized into smaller modules, which, in turn, can be organized into even smaller modules, all the way down to primitive operations. This hierarchical organization is part of what biologists investigate. Large organizations have a hierarchical organization so that the top-level decision makers are not overwhelmed by details and do not have to micromanage all details of the organization. Procedural abstraction and object-oriented programming in computer science are designed to enable simplification of a system by exploiting modularity and abstraction.

In the modularity dimension, an agent's structure is one of the following:
Flat: there is no organizational structure
Modular: the system is decomposed into interacting modules that can be understood on their own; or
Hierarchical: the system is modular, and the modules themselves are decomposed into interacting modules, each of which are hierarchical systems, and this recursion grounds out into simple components.

In a flat or modular structure the agent typically reasons at a single level of abstraction. In a hierarchical structure the agent reasons at multiple levels of abstraction. The lower levels of the hierarchy involve reasoning at a lower level of abstraction.

For example, in taking a trip from home to a holiday location overseas, an agent, such as your self, must get from home to an airport, fly to an airport near the destination, then get from the airport to the destination. It also must make a sequence of specific leg or wheel movements to actually move. In a flat representation, the agent chooses one level of abstraction and reasons at that level. A modular representation would divide the task into a number of subtasks that can be solved separately for example, booking tickets, getting to the departure airport, getting to the destination airport, and getting to the holiday location. In a hierarchical representation, the agent will solve these subtasks in a hierarchical way, until the problem is reduced to simple problems such a sending an http request or taking a particular step.

A hierarchical decomposition is important for reducing the complexity of building an intelligent agent that acts in a complex environment. However, to explore the other dimensions, we initially ignore the hierarchical structure and assume a flat representation. Ignoring hierarchical decomposition is often fine for small or moderately sized problems, as it is for simple animals, small organizations, or small to moderately sized computer programs. When problems or systems become complex, some hierarchical organization is required.

2.6.2 Representation Scheme

The representation scheme dimension concerns how the world is described. The different ways the world could be to affect what an agent should do are called states. We can factor the state of the world into the agent's internal state (its belief state) and the environment state. At the simplest level, an agent can reason explicitly in terms of individually identified states.

Example: A thermostat for a heater may have two belief states: off and heating. The environment may have three states: cold, comfortable, and hot. There are thus six states corresponding to the different combinations of belief and environment states. These states may not fully describe the world, but they are adequate to describe what a thermostat should do. The thermostat should move to, or stay in, heating if the environment is cold and move to, or stay in, off if the environment is hot. If the environment is comfortable, the thermostat should stay in its current state. The agent heats in the heating state and does not heat in the off state.

Instead of enumerating states, it is often easier to reason in terms of the state's features or propositions that are true or false of the state. A state may be described in terms of features, where a feature has a value in each state. In the representation scheme dimension, the agent reasons in terms of:

- states
- features
- or relational descriptions, in terms of individuals and relations.

Some of the frameworks will be developed in terms of states, some in terms of features and some relationally.

2.6.3 Planning Horizon

The next dimension is how far ahead in time the agent plans. For example, when a dog is called to come, it should turn around to start running in order to get a reward in the future. It does not act only to get an immediate reward. Plausibly, a dog does not act for goals arbitrarily far in the future in a few months, whereas people do working hard now to get a holiday next year.

How far the agent "looks into the future" when deciding what to do is called the planning horizon. That is, the planning horizon is how far ahead the agent considers the consequences of its actions. For completeness, we include the non-planning case where the agent is not reasoning in time. The time points considered by an agent when planning are called stages.

In the planning horizon dimension, an agent is one of the following:

- A non-planning agent is an agent that does not consider the future when it decides what to do or when time is not involved.
- A finite horizon planner is an agent that looks for a fixed finite number of time steps ahead. For example, a doctor may have to treat a patient but may have time for some testing and so there may be two stages: a testing stage and a treatment stage to plan for. In the degenerate case where an agent only looks one time step ahead, it is said to be greedy or myopic.
- An indefinite horizon planner is an agent that looks ahead some finite, but not predetermined, number of steps ahead. For example, an agent that must get to some location may not know a priori how many steps it will take to get there.
- An infinite horizon planner is an agent that plans on going on forever. This is often called a process. For example, the stabilization module of a legged robot should go on forever; it cannot stop when it has achieved stability, because the robot has to keep from falling over.

2.6.4 Uncertainty

An agent could assume there is no uncertainty, or it could take uncertainty in the domain into consideration. Uncertainty is divided into two dimensions: one for uncertainty from sensing and one for uncertainty about the effect of actions.

Sensing uncertainty

In some cases, an agent can observe the state of the world directly. For example, in some board games or on a factory floor, an agent may know exactly the state of the world. In many other cases, it may only have some noisy perception of the state and the best it can do is to have a probability distribution over the set of possible states based on what it perceives. For example, given a patient's symptoms, a medical doctor may not actually know which disease a patient may have and may have only a probability distribution over the diseases the patient may have.

The sensing uncertainty dimension concerns whether the agent can determine the state from the observations:

- Fully observable is when the agent knows the state of the world from the observations.
- Partially observable is when the agent does not directly observe the state of the world. This occurs when many possible states can result in the same observations or when observations are noisy.

Effect uncertainty

In some cases an agent knows the effect of an action. That is, given a state and an action, it can accurately predict the state resulting from carrying out that action in that state. For example, an agent interacting with a file system may be able to predict the effect of deleting a file given the state of the file system. In many cases, it is difficult to predict the effect of an action, and the best an agent can do is to have a probability distribution over the effects. For example, a person may not know the effect of calling his dog, even if he knew the state of the dog, but, based on experience, he has some idea of what it will do. The dog owner may even have some idea of what another dog that he has never seen before will do if he calls it.

The effect uncertainty dimension is that the dynamics can be
Deterministic: when the state resulting from an action is determined by an action and the prior state or
Stochastic: when there is only a probability distribution over the resulting states.

This dimension only makes sense when the world is fully observable. If the world is partially observable, a stochastic system can be modelled as a deterministic system where the effect of an action depends on some unobserved feature. It is a separate dimension because many of the frameworks developed are for the fully observable, stochastic action case.

Preference

Agents act to have better outcomes for themselves. The only reason to choose one action over another is because the preferred action will lead to more desirable outcomes.

An agent may have a simple goal, which is a state to be reached or a proposition to be true such as getting its owner a cup of coffee. Other agents may have more complex preferences. For example, a medical doctor may be expected to take into account suffering, life expectancy, quality of life, monetary costs for the patient, the doctor, and society, the ability to justify decisions in case of a lawsuit, and many other desiderata. The doctor must trade these considerations off when they conflict, as they invariably do.

The preference dimension is whether the agent has:

- Goals, either achievement goals to be achieved in some final state or maintenance goals that must be maintained in all visited states. For example, the goals for a robot may be to get two cups of coffee and a banana, and not to make a mess or hurt anyone.
- Complex preferences involve trade-offs among the desirability of various outcomes, perhaps at different times. An ordinal preference is where only the ordering of the preferences is important. A cardinal preference is where the magnitude of the values matters. For example, an ordinal preference may be that Sam prefers cappuccino over black coffee and prefers black coffee over tea. A cardinal preference may give a trade-off between the wait time and the type of beverage, and a mess-taste trade-off, where Sam is prepared to put up with more mess in the preparation of the coffee if the taste of the coffee is exceptionally good.

2.6.5 Number of Agents

An agent reasoning about what it should do in an environment where it is the only agent is difficult enough. However, reasoning about what to do when there are other agents who are also reasoning is much more difficult. An agent in a multi agent setting should reason strategically about other agents; the other agents may act to trick or manipulate the agent or may be available to cooperate with the agent. With multiple agents, is often optimal to act randomly because other agents can exploit deterministic strategies. Even when the agents are cooperating and have a common goal, the problem of coordination and communication makes multi agent reasoning more challenging. However, many domains contain multiple agents and ignoring other agents' strategic reasoning is not always the best way for an agent to reason.

Taking the point of view of a single agent, the number of agents dimension considers whether the agent does

- Single agent reasoning, where the agent assumes that any other agents are just part of the environment. This is a reasonable assumption if there are no other agents or if the other agents are not going to change what they do based on the agent's action.
- Multiple agent reasoning, where the agent takes the reasoning of other agents into account. This happens when there are other intelligent agents whose goals or preferences depend, in part, on what the agent does or if the agent must communicate with other agents.

2.6.6 Learning

In some cases, a designer of an agent may have a good model of the agent and its environment. Often a designer does not have a good model, and an agent should use data from its past experiences and other sources to help it decide what to do.

The learning dimension determines whether

- knowledge is given or
- knowledge is learned (from data or past experience).

Learning typically means finding the best model that fits the data. Sometimes this is as simple as tuning a fixed set of parameters, but it can also mean choosing the best representation out of a class of representations. Learning is a huge field in itself but does not stand in isolation from the rest of AI. There are many issues beyond fitting data, including how to incorporate background knowledge, what data to collect, how to represent the data and the resulting representations, what learning biases are appropriate, and how the learned knowledge can be used to affect how the agent acts.

2.6.7 Computational Limits

Sometimes an agent can decide on its best action quickly enough for it to act. Often there are computational resource limits that prevent an agent from carrying out the best action. That is, the agent may not be able to find the best action quickly enough within its memory limitations to act while that action is still the best thing to do. For example, it may not be much use to take 10 minutes to derive what was the best thing to do 10 minutes ago, when the agent has to act now. Often, instead, an agent must trade off how long it takes to get a solution with how good the solution is; it may be better to find a reasonable solution quickly than to find a better solution later because the world will have changed during the computation.

The computational limits dimension determines whether an agent has

- Perfect rationality, where an agent reasons about the best action without taking into account its limited computational resources
- Bounded rationality, where an agent decides on the best action that it can find given its computational limitations.

Computational resource limits include computation time, memory, and numerical accuracy caused by computers not representing real numbers exactly. An anytime algorithm is an algorithm whose solution quality improves with time. In particular, it is one that can produce its current best solution at any time, but given more time it could produce even better solutions. We can ensure that the quality doesn’t decrease by allowing the agent to store the best solution found so far and return that when asked for a solution. However, waiting to act has a cost; it may be better for an agent to act before it has found what would have been the best solution.

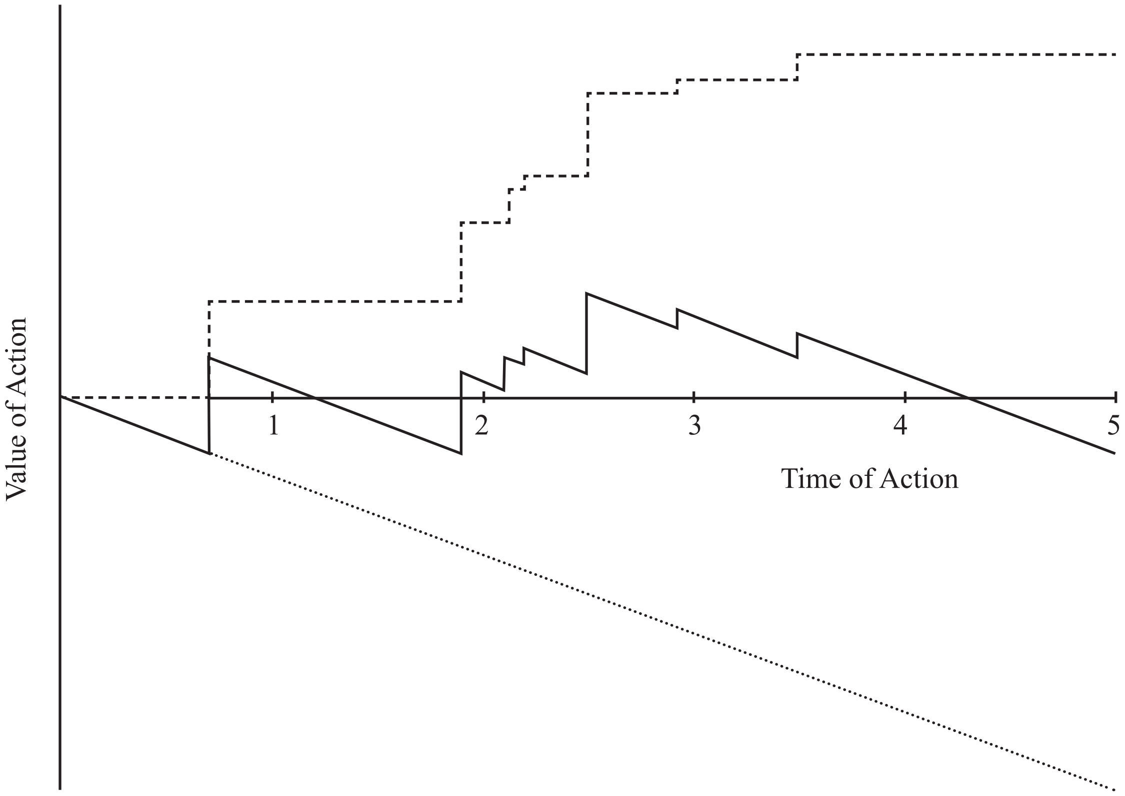


Fig 2.3 Solution quality as a function of time for an anytime algorithm

The agent has to choose an action. As time progresses, the agent can determine better actions. The value to the agent of the best action found so far, if it had been carried out initially, is given by the dashed line. The reduction in value to the agent by waiting to act is given by the dotted line. The net value to the agent, as a function of the time it acts, is given by the solid line.

Figure above shows how the computation time of an anytime algorithm can affect the solution quality. The agent has to carry out an action but can do some computation to decide what to do. The absolute solution quality, had the action been carried out at time zero, shown as the dashed line at the top, is improving as the agent takes time to reason.

However, there is a penalty associated with taking time to act. In this figure, the penalty is shown as the dotted line at the bottom, is proportional to the time taken before the agent acts. These two values can be added to get the discounted quality, the time-dependent value of computation; this is the solid line in the middle of the graph.

To take into account bounded rationality, an agent must decide whether it should act or think more. This is challenging because an agent typically does not know how much better off it would be if it only spent a little bit more time reasoning. Moreover, the time spent thinking about whether it should reason may detract from actually reasoning about the domain. However, bounded rationality can be the basis for approximate reasoning.

2.6.8 Interaction of the Dimensions

Table 1.1 summarises the dimensions of complexity. Unfortunately, we cannot study these dimensions independently because they interact in complex ways. Here we give some examples of the interactions.

Dimension	Values
Modularity	Flat, modular, hierarchical
Representation scheme	States, features, relations
Planning horizon	Non-planni9ng, finite stage, indefinite stage, infinite stage
Sensing uncertainty	Fully observable, partially observable
Effect uncertainty	Deterministic, stochastic
Preference	Goals, complex preferences
Learning	Knowledge is given, knowledge is learned
Number of agents	Single agent, multiple agents
Computational limits	Perfect rationality, bounded rationality

Table 2.1 Dimensions of complexity

The representation dimension interacts with the modularity dimension in that some modules in a hierarchy may be simple enough to reason in terms of a finite set of states, whereas other levels of abstraction may require reasoning about individuals and relations. For example, in a delivery robot, a module that maintains balance may only have a few states. A module that must prioritize the delivery of multiple parcels to multiple people may have to reason about multiple individuals for example, people, packages, and rooms and the relations between them. At a higher level, a module that reasons about the activity over the day may only require a few states to cover the different phases of the day for example, there might be three states:

- busy time
- available for requests
- and recharge time

The planning horizon interacts with the modularity dimension. For example, at a high level, a dog may be getting an immediate reward when it comes and gets a treat. At the level of deciding where to place its paws, there may be a long time until it gets the reward, and so at this level it may have to plan for an indefinite stage.

Sensing uncertainty probably has the greatest impact on the complexity of reasoning. It is much easier for an agent to reason when it knows the state of the world than when it doesn't. Although sensing uncertainty with states is well understood, sensing uncertainty with individuals and relations is an active area of current research.

The effect uncertainty dimension interacts with the modularity dimension: at one level in a hierarchy, an action may be deterministic, whereas at another level, it may be stochastic. As an example, consider the result of flying to Paris with a companion you are trying to impress. At one level you may know where you are; at a lower level, you may be quite lost and not know where you are on a map of the airport. At an even lower level responsible for maintaining balance, you may know where you are: you are standing on the ground. At the highest level, you may be very unsure whether you have impressed your companion.

Preference models interact with uncertainty because an agent must have a trade-off between satisfying a major goal with some probability or a less desirable goal with a higher probability. Multiple agents can also be used for modularity; one way to design a single agent is to build multiple interacting agents that share a common goal of making the higher-level agent act intelligently. Some researchers, such as Minsky (1986), argue that intelligence is an emergent feature from a "society" of unintelligent agents.

Learning is often cast in terms of learning with features determining which feature values best predict the value of another feature. However, learning can also be carried out with individuals and relations. Much work has been done on learning hierarchies, learning in partially observable domains, and learning with multiple agents, although each of these is challenging in its own right without considering interactions with multiple dimensions.

Two of these dimensions, modularity and bounded rationality, promise to make reasoning more efficient. Although they make the formalism more complicated, breaking the system into smaller components, and making the approximations needed to act in a timely fashion and within memory limitations, should help build more complex systems.

2.7 Prototypical Applications

AI applications are widespread and diverse and include medical diagnosis, scheduling factory processes, robots for hazardous environments, game playing, autonomous vehicles in space, natural language translation systems, and tutoring systems. Rather than treating each application separately, we abstract the essential features of such applications to allow us to study the principles behind intelligent reasoning and action.

This section outlines four application domains that will be developed in examples throughout the book. Although the particular examples presented are simple otherwise they would not fit into the book. The application domains are representative of the range of domains in which AI techniques can be, and are being, used.

The four application domains are as follows:

- An autonomous delivery robot roams around a building delivering packages and coffee to people in the building. This delivery agent should be able to find paths, allocate resources, receive requests from people, make decisions about priorities, and deliver packages without injuring people or itself.
- A diagnostic assistant helps a human troubleshoot problems and suggests repairs or treatments to rectify the problems. One example is an electrician's assistant that suggests what may be wrong in a house, such as a fuse blown, a light switch broken, or a light burned out, given some symptoms of electrical problems. Another example is a medical diagnostician that finds potential diseases, useful tests, and appropriate treatments based on knowledge of a particular medical domain and a patient's symptoms and history. This assistant should be able to explain its reasoning to the person who is carrying out the tests and repairs and who is ultimately responsible for their actions.
- A tutoring system interacts with a student, presenting information about some domain and giving tests of the student's knowledge or performance. This entails more than presenting information to students. Doing what a good teacher does, namely tailoring the information presented to each student based on his or her knowledge, learning preferences, and misunderstandings, is more challenging. The system must understand both the subject matter and how students learn.
- A trading agent knows what a person wants and can buy goods and services on her behalf. It should know her requirements and preferences and how to trade off competing objectives. For example, for a family holiday a travel agent must book hotels, airline flights, rental cars, and entertainment, all of which must fit together. It should determine a customer's trade-offs.

2.7.1 An Autonomous Delivery Robot

Imagine a robot that has wheels and can pick up objects and put them down. It has sensing capabilities so that it can recognize the objects that it must manipulate and can avoid obstacles. It can be given orders in natural language and obey them, making reasonable choices about what to do when its goals conflict. Such a robot could be used in an office environment to deliver packages, mail, and/or coffee, or it could be embedded in a wheelchair to help disabled people. It should be useful as well as safe.

In terms of the black box characterization of an agent, the autonomous delivery robot has the following as inputs:

- Prior knowledge, provided by the agent designer, about its own capabilities, what objects it may encounter and have to differentiate, what requests mean, and perhaps about its environment, such as a map.

- Past experience obtained while acting, for instance, about the effect of its actions, what objects are common in the world, and what requests to expect at different times of the day.
- Goals in terms of what it should deliver and when, as well as preferences that specify trade-offs, such as when it must forgo one goal to pursue another, or the trade-off between acting quickly and acting safely.
- Observations about its environment from such input devices as cameras, sonar, touch, sound, laser range finders, or keyboards.

The robot's outputs are motor controls that specify how its wheels should turn, where its limbs should move, and what it should do with its grippers. Other outputs may include speech and a video display.

Each dimension can add conceptual complexity to the task of reasoning:

- A hierarchical decomposition can allow the complexity of the overall system to be increased while allowing each module to be simple and able to be understood by itself.
- Modelling in terms of features allows for a much more comprehensible system than modelling explicit states. For example, there may be features for the robot's location, the amount of fuel it has, what it is carrying, and so forth. Reasoning in terms of the states, where a state is an assignment of a value to each feature loses the structure that is provided by the features.
- The planning horizon can be finite if the agent only looks ahead a few steps. The planning horizon can be indefinite if there is a fixed set of goals to achieve. It can be infinite if the agent has to survive for the long term, with ongoing requests and actions, such as delivering mail whenever it arrives and recharging its battery when its battery is low.
- There could be goals, such as "deliver coffee to Chris and make sure you always have power." A more complex goal may be to "clean up the lab, and put everything where it belongs." There can be complex preferences, such as "deliver mail when it arrives and service coffee requests as soon as possible, but it is more important to deliver messages marked as important, and Chris really needs her coffee quickly when she asks for it."
- There can be sensing uncertainty because the robot does not know what is in the world based on its limited sensors.
- There can be uncertainty about the effects of an action, both at the low level, such as due to slippage of the wheels, or at the high level in that the agent might not know if putting the coffee on Chris's desk succeeded in delivering coffee to her.
- There can be multiple robots, which can coordinate to deliver coffee and parcels and compete for power outlets. There may also be children out to trick the robot.
- A robot has lots to learn, such as how slippery floors are as a function of their shininess, where Chris hangs out at different parts of the day and when she will ask for coffee, and which actions result in the highest rewards.

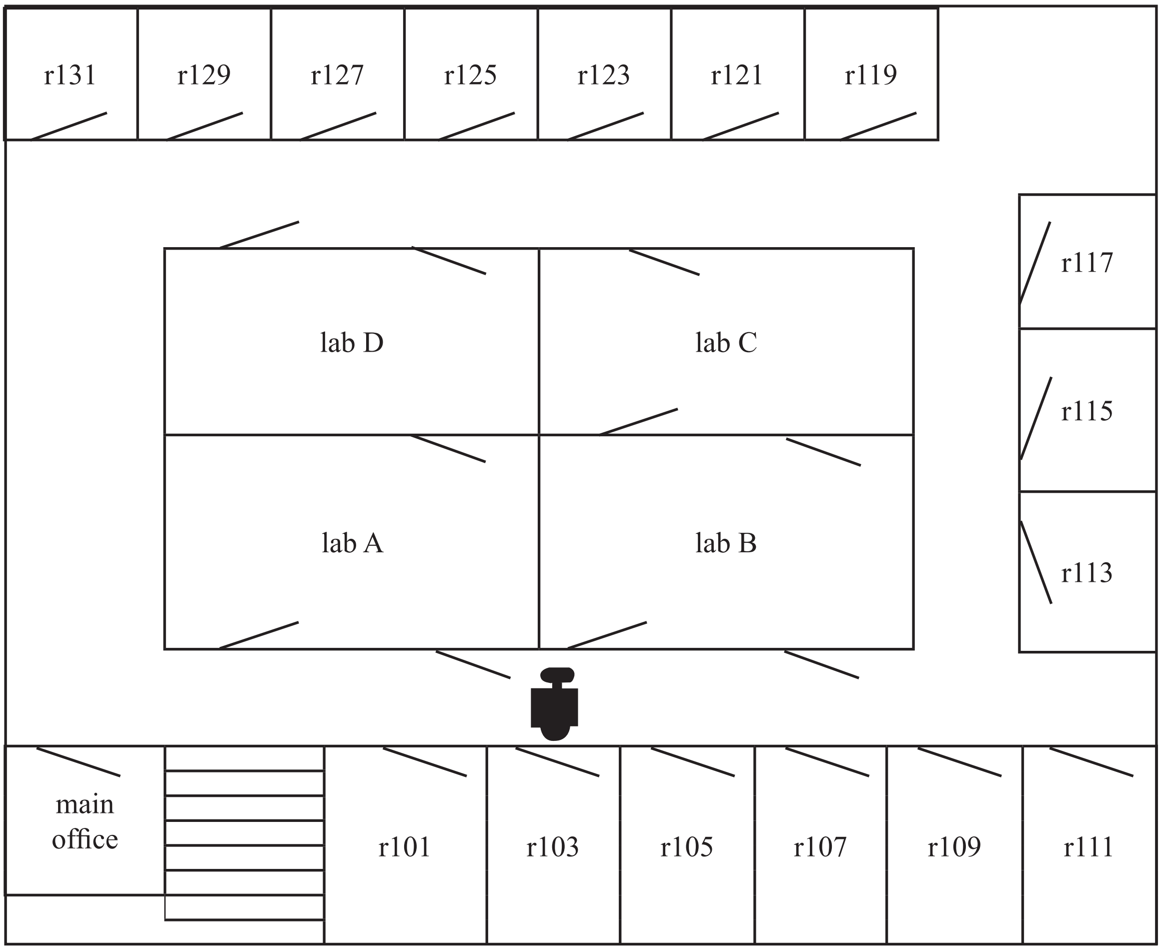


Fig. 2.4 An environment for the delivery robot, which shows a typical laboratory environment

Figure shows the locations of the doors and which way they open. It depicts a typical laboratory environment for a delivery robot. This environment consists of four laboratories and many offices. The robot can only push doors, and the directions of the doors in the diagram reflect the directions in which the robot can travel. Rooms require keys, and those keys can be obtained from various sources. The robot must deliver parcels, beverages, and dishes from room to room. The environment also contains a stairway that is potentially hazardous to the robot.

2.7.2 A Diagnostic Assistant

A diagnostic assistant is intended to advise a human about some particular system such as a medical patient, the electrical system in a house, or an automobile. The diagnostic assistant should advise about potential underlying faults or diseases, what tests to carry out, and what treatment to prescribe. To give such advice, the assistant requires a model of the system, including knowledge of potential causes, available tests, and available treatments, and observations of the system (which are often called symptoms).

To be useful, the diagnostic assistant must provide added value, be easy for a human to use, and not be more trouble than it is worth. A diagnostic assistant connected to the Internet can draw on expertise from throughout the world, and its actions can be based on the most up-to-date research. However, it must be able to justify why the suggested diagnoses or actions are appropriate. Humans are, and should be, suspicious of computer systems that are opaque and impenetrable. When humans are responsible for what they do, even if it is based on a computer system's advice, they should have reasonable justifications for the suggested actions.

In terms of the black box definition of an agent in Figure 1.3, the diagnostic assistant has the following as inputs:

- Prior knowledge, such as how switches and lights normally work, how diseases or malfunctions manifest themselves, what information tests provide, and the effects of repairs or treatments.
- Past experience, in terms of data of previous cases that include the effects of repairs or treatments, the prevalence of faults or diseases, the prevalence of symptoms for these faults or diseases, and the accuracy of tests. These data are usually about similar artifacts or patients, rather than the actual one being diagnosed.
- Goals of fixing the device and trade-offs, such as between fixing or replacing different components, or whether patients prefer to live longer if it means they will be in pain or be less coherent.
- Observations of symptoms of a device or patient.

The output of the diagnostic assistant is in terms of recommendations of treatments and tests, along with a rationale for its recommendations.

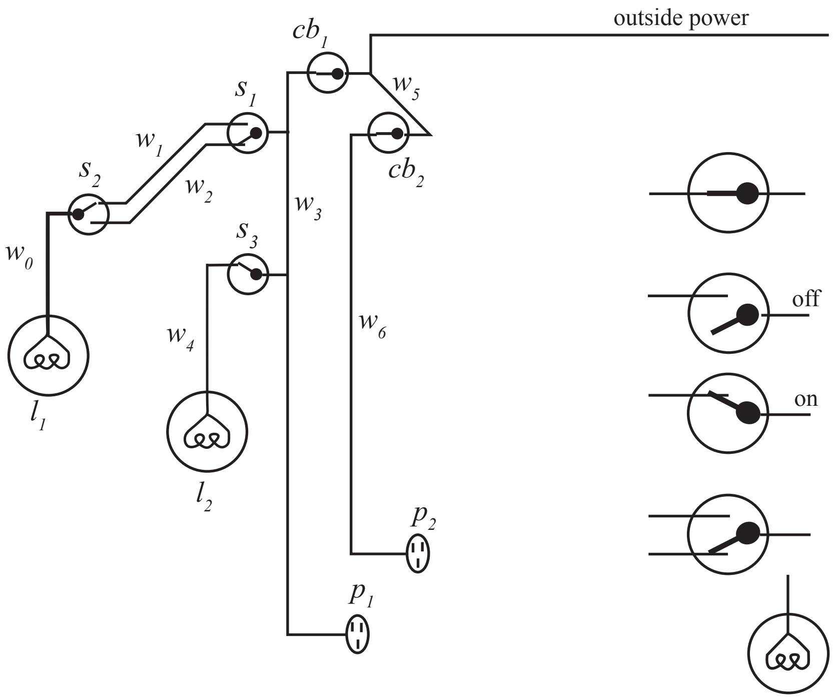


Fig. 2.5 An electrical environment for the diagnostic assistant

Figure above shows a depiction of an electrical distribution system in a house. In this house, power comes into the house through circuit breakers and then it goes to power outlets or to lights through light switches. For example, light l1 is on if there is power coming into the house, if circuit breaker cb1 is on, and if switches s1 and s2 are either both up or both down. This is the sort of model that normal householders may have of the electrical power in the house, and which they could use to determine what is wrong given evidence about the position of the switches and which lights are on and which are off. The diagnostic assistant is there to help a householder or an electrician troubleshoot electrical problems.

Each dimension is relevant to the diagnostic assistant:

- Hierarchical decomposition allows for very-high-level goals to be maintained while treating the lower-level causes and allows for detailed monitoring of the system. For example, in a medical domain, one module could take the output of a heart monitor and give higher-level observations such as notifying when there has been a change in the heart rate. Another module could take in this observation and other high-level observations and notice what other symptoms happen at the same time as a change in heart rate. In the electrical domain, Figure 2.5 is at one level of abstraction; a lower level could specify the voltages, how wires are spliced together, and the internals of switches.
- Most systems are too complicated to reason about in terms of the states, and so they are usually described in terms of the features or individual components and relations among them. For example, a human body may be described in terms of the values for features of its various components. Designers may want to model the dynamics without knowing the actual individuals. For example, designers of the electrical diagnosis system would model how lights and switches work before knowing which lights and switches exist in an actual house and, thus, before they know the features. This can be achieved by modelling in terms of relations and their interaction and by adding the individual components when they become known.
- It is possible to reason about a static system, such as reasoning about what could be wrong when a light is off given the position of switches. It is also possible to reason about a sequence of tests and treatments, where the agents keep testing and treating until the problem is fixed, or where the agent carries out ongoing monitoring of a system, continuously fixing whatever gets broken.
- Sensing uncertainty is the fundamental problem that faces diagnosis. Diagnosis is required if an agent cannot directly observe the internals of the system.
- Effect uncertainty also exists in that an agent may not know the outcome of a treatment and, often, treatments have unanticipated outcomes.
- The goal may be as simple as “fix what is wrong,” but often there are complex trade-offs involving costs, pain, life expectancy, the probability that the diagnosis is correct, and the uncertainty as to efficacy and side effects of the treatment.
- Although it is often a single-agent problem, diagnosis becomes more complicated when multiple experts are involved who perhaps have competing experience and models. There may be other patients with whom an agent must compete for resources for example, doctor’s time, surgery rooms.
- Learning is fundamental to diagnosis. It is through learning that we understand the progression of diseases and how well treatments work or do not work. Diagnosis is a challenging domain for learning, because all patients are different, and each individual doctor’s experience is only with a few patients with any particular set of symptoms. Doctors also see a biased sample of the population; those who come to see them usually have unusual or painful symptoms.
- Diagnosis often requires a quick response, which may not allow for the time to carry out exhaustive reasoning or perfect rationality.

2.7.3 An Intelligent Tutoring System

An intelligent tutoring system is a computer system that tutors students in some domain of study. For example, in a tutoring system to teach elementary physics, such as mechanics, the system may present the theory and worked-out examples. The system can ask the student questions and it must be able to understand the student’s answers, as well as determine the student’s knowledge based on what answers were given. This should then affect what is presented and what other questions are asked of the student. The student can ask questions of the system, and so the system should be able to solve problems in the physics domain.

In terms of the black box definition of an agent, an intelligent tutoring system has the following as inputs:

- Prior knowledge, provided by the agent designer, about the subject matter being taught, teaching strategies, possible errors, and misconceptions of the students.
- Past experience, which the tutoring system has acquired by interacting with students, about what errors students make, how many examples it takes to learn something, and what students forget. This can be information about students in general or about a particular student.
- Preferences about the importance of each topic, the level of achievement of the student that is desired, and costs associated with usability. There are often complex trade-offs among these.
- Observations of a student's test results and observations of the student's interaction or non-interaction with the system. Students can also ask questions or provide new examples with which they want help.

The output of the tutoring system is the information presented to the student, tests the students should take, answers to questions, and reports to parents and teachers. Each dimension is relevant to the tutoring system:

- There should be both a hierarchical decomposition of the agent and a decomposition of the task of teaching. Students should be taught the basic skills before they can be taught higher-level concepts. The tutoring system has high-level teaching strategies, but, at a much lower level, it must design the details of concrete examples and specific questions for a test.
- A tutoring system may be able to reason in terms of the state of the student. However, it is more realistic to have multiple features for the student and the subject domain.
- In terms of planning horizon, for the duration of a test, it may be reasonable to assume that the domain is static and that the student does not learn while taking a test. For some subtasks, a finite horizon may be appropriate. For example, there may be teach, test, re teaches sequence.
- Uncertainty will have to play a large role. The system cannot directly observe the knowledge of the student. All it has is some sensing input, based on questions the student asks or does not ask, and test results. The system will not know for certain the effect of a particular teaching episode.
- Although it may be possible to have a simple goal such as to teach some particular concept, it is more likely that complex preferences must be taken into account. One reason is that, with uncertainty.
- It may be appropriate to treat this as a single-agent problem. However, the teacher, the student, and the parent may all have different preferences that must be taken into account. Each of these agents may act strategically by not telling the truth.
- We would expect the system to be able to learn about what teaching strategies work, how well some questions work at testing concepts, and what common mistakes students make. It could learn general knowledge, or knowledge particular to a topic for example, learning about what strategies work for teaching mechanics or knowledge about a particular student, such as learning what works for Sam.
- One could imagine that choosing the most appropriate material to present would take a lot of computation time. However, the student must be responded to in a timely fashion. Bounded rationality would play a part in ensuring that the system does not compute for a long time while the student is waiting.

2.7.4 A Trading Agent

A trading agent is like an automaton, but instead of interacting with a physical environment, it interacts with an information environment. It procures goods and services for a user. The simplest trading agent involves proxy bidding for a user on an auction site, where the system will keep bidding until the user's price limit is reached. A more complicated trading agent will buy multiple complementary items, like booking a flight, a hotel, and a rental car that fit together, in addition to trading off competing preferences. Another example of a trading agent is one that monitors how much food and groceries are in a household, monitors the prices, and orders goods before they are needed, keeping costs to a minimum.

In terms of the black box definition of an agent the trading agent has the following as inputs:

- Prior knowledge about types of goods and services, selling practices, and how auctions work
- Past experience about where is the best place to look for specials, how prices vary in time in an auction, and when specials tend to turn up
- Preferences in terms of what the user wants and how to trade off competing goals
- Observations about what items are available, their price, and, perhaps, how long they are available

The output of the trading agent is either a proposal to the user that they can accept or reject or an actual purchase.

The trading agent should take all of the dimensions into account:

- Hierarchical decomposition is essential because of the complexity of domains. Study the example of making all of the arrangements and purchases for a custom holiday for a traveller. It is simpler to have a module that can purchase a ticket and optimize connections and timing, rather than to do this at the same time as determining what doors to go through to get to the taxi stand.
- The state space of the trading agent is too large to reason in terms of individual states. There are also too many individuals to reason in terms of features. The trading agent will have to reason in terms of individuals such as customers, days, hotels, flights, and so on.
- A trading agent typically does not make just one purchase, but must make a sequence of purchases, often a large number of sequential decisions for example, purchasing one hotel room may require booking ground transportation, which may in turn require baggage storage, and often plans for ongoing purchasing, such as for an agent that makes sure a household has enough food on hand at all times.
- There is often sensing uncertainty in that a trading agent does not know all of the available options and their availability, but must find out information that can become old quickly for example, if some hotel becomes booked up. A travel agent does not know if a flight will be cancelled or delayed or whether the passenger's luggage will be lost. This uncertainty means that the agent must plan for the unanticipated.
- There is also effect uncertainty in that the agent does not know if an attempted purchase will succeed.
- Complex preferences are at the core of the trading agent. The main problem is in allowing users to specify what they want. The preferences of users are typically in terms of functionality, not components. For example, typical computer buyers have no idea of what hardware to buy, but they know what functionality they want and they also want the flexibility to be able to use new features that might not yet exist. Similarly, in a travel domain, what activities a user may want may depend on the location. Users also may want the ability to participate in a local custom at their destination, even though they may not know what these customs are.
- A trading agent has to reason about other agents. In commerce, prices are governed by supply and demand; this means that it is important to reason about the other competing agents. This happens particularly in a world where many items are sold by auction. Such reasoning becomes particularly difficult when there are items that must complement each other, such as flights and hotel booking, and items that can substitute for each other, such as bus transport or taxis.
- A trading agent should learn about what items sell quickly, which of the suppliers are reliable, where to find good deals, and what unanticipated events may occur.
- A trading agent faces severe communication limitations. In the time between finding that some item is available and coordinating the item with other items, the item may have sold out. This can sometimes be alleviated by sellers agreeing to hold some items (not to sell them to someone else in the meantime), but sellers will not be prepared to hold an item indefinitely if others want to buy it.

Summary

- John McCarthy, coined the term Artificial Intelligence (AI) in the year 1956.
- Artificial Intelligence is the science and engineering of making intelligent machines, especially intelligent computer programs.
- Artificial intelligence, or AI, is the field that studies the synthesis and analysis of computational agents that act intelligently.
- AI takes an approach analogous to that of aerodynamics. AI researchers are interested in testing general hypotheses about the nature of intelligence by building machines that are intelligent and that do not necessarily mimic humans or organizations.
- An agent has only a finite memory and it has limited time to act. It typically cannot observe the state of the world directly.
- This idea of intelligence being defined by external behaviour was the motivation for a test for intelligence designed by Turing (1950), which has become known as the Turing test.
- The science of AI could be described as “synthetic psychology,” “experimental philosophy,” or “computational epistemology”.
- A representation scheme is the form of the knowledge that is used in an agent.
- If the environment is a physical setting an agent could be considered as a coupling of a computational engine with physical sensors and actuators, called a robot.
- Modularity is the extent to which a system can be decomposed into interacting modules that can be understood separately.
- Learning typically means finding the best model that fits the data.
- Agents acting in environments range in complexity from thermostats to companies with multiple goals acting in competitive environments.
- AI applications are widespread and diverse and include medical diagnosis, scheduling factory processes, robots for hazardous environments, game playing, autonomous vehicles in space, natural language translation systems, and tutoring systems.
- The knowledge level is a level of abstraction that considers what an agent knows and believes and what its goals are.
- The symbol level is a level of description of an agent in terms of the reasoning it does.
- The manipulation of symbols to produce action is called reasoning.
- Knowledge is a progression that starts with data which is of limited services.

References

- *Artificial Intelligence*, [Online] Available at: <http://artint.info/html/ArtInt_182.html>[Accessed 20 July 2012].
- Bobrow, D. G., *Artificial Intelligence*, [Online] Available at: <http://www2.parc.com/istl/groups/hdi/papers/stefik-molgen-part-1.pdf>[Accessed 20 July 2012].
- Russell, 2003. *Artificial Intelligence: A Modern Approach*, 2nd ed., Pearson Education India.
- Haugeland, J., 1989. *Artificial intelligence: The very idea,* MIT Press.
- 2008. *Lecture - 1 Introduction to Artificial Intelligence*,. [Video Online] Available at: <http://www.youtube.com/watch?v=fV2k2ivttL0>[Accessed 20 July 2012].
- 2012. *Artificial Intelligence Lecture No. 1*, [Video Online] Available at: <http://www.youtube.com/watch?v=katiy95_mxo>[Accessed 20 July 2012].

Recommended Reading

- Russell, S. & Norvig, P., 2009. *Artificial Intelligence: A Modern Approach*, 3rd ed., Prentice Hall.
- Jones, M. T., 2008. *Artificial Intelligence: A Systems Approach (Computer Science Series)*, 1st ed., Jones and Bartlett Publishers.
- Jackson, P. C., 1985. *Introduction to Artificial Intelligence*, 2nd ed., Dover Publications.

Self Assessment

1. Which of the following statements is false?
   a. AI may be interpreted as the opposite of real intelligence.
   b. Artificial intelligence does not have any source.
   c. We can distinguish real phenomenon versus fake phenomenon.
   d. We can also distinguish natural versus artificial.

2. The idea of Artificial Intelligence was designed by __________.
   a. Turing
   b. Newell
   c. Simon
   d. Pitts

3. ___________is used to calculate the computation time for solution quality.
   a. Anytime algorithm
   b. Computation algorithm
   c. Trading agent
   d. Optimal algorithm

4. __________can be used to solve problems in its own domain.
   a. Anytime algorithm
   b. Computation algorithm
   c. Agent
   d. Knowledge

5. Match the columns

Dimension	**Value**
1. Modularity	A. Deterministic, stochastic
2. Effect uncertainty	B. States, features, relations
3. Preference	C. Flat, modular, hierarchical
4. Representation scheme	D. Goals, complex preferences

   a. 1-A, 2-B, 3-C, 4-D
   b. 1D, 2-C, 3-B, 4-A
   c. 1-C, 2-A, 3-D, 4-B
   d. 1-B, 2-D, 3-A, 4-C

6. Which of the following statement is true?
   a. Diagnosis is required if an agent cannot directly observe the internals of the system.
   b. Diagnosis is necessary for a quick response.
   c. Diagnosis is very simple even if multiple experts are involved.
   d. Knowledge is fundamental to diagnosis.

7. _________ is fundamental to diagnosis.
   a. Knowledge
   b. Algorithm
   c. Learning
   d. Doctor

8. ____________procures goods and services for a user.
   a. Intelligent tutoring system
   b. A trading agent
   c. An Autonomous Delivery Robot
   d. Diagnostic system

9. The effect uncertainty dimension is dynamics when;
   a. There is only a probability distribution over the resulting states.
   b. The dimension makes sense when the world is fully observable.
   c. There is only a probability distribution over the resulting prior states.
   d. It is a separate dimension.

10. The_________ scheme dimension concerns how the world is described.
    a. knowledge
    b. learning
    c. solution
    d. representation

Chapter III

Neural Networks

Aim

The aim of this chapter is to:

- introduce natural and artificial neural networks
- elucidate biological model and cellular automata
- explain models of computation

Objectives

The objectives of this chapter are to:

- explain the elements of a computing model
- explicate the hodgkin–huxley model of a cell membrane
- elucidate information processing at the neurons and synapses

Learning outcome

At the end of this chapter, you will be able to:

- understand the structure of the neurons
- identify transmission of information
- describe the storage of information

3.1 Introduction

Research in the field of neural networks has been attracting increasing attention in recent years. Since 1943, when Warren McCulloch and Walter Pitts presented the first model of artificial neurons, new and more sophisticated proposals have been made from decade to decade. Mathematical analysis has solved some of the mysteries posed by the new models but has left many questions open for future investigations. Needless to say, the study of neurons, their interconnections, and their role as the brain's elementary building blocks is one of the most dynamic and important research fields in modern biology.

We can illustrate the relevance of this endeavour by pointing out that between 1901 and 1991 approximately ten percent of the Nobel Prizes for Physiology and Medicine were awarded to scientists who contributed to the understanding of the brain. It is not an exaggeration to say that we have learned more about the nervous system in the last fifty years than ever before. In this chapter we deal with artificial neural networks, and therefore the first question to be clarified is their relation to the biological paradigm. What do we abstract from real neurons for our models? What is the link between neurons and artificial computing units? This chapter gives a preliminary answer to these important questions.

3.2 Natural and Artificial Neural Networks

Artificial neural networks are an attempt at modelling the information processing capabilities of nervous systems. Thus, first of all, we need to consider the essential properties of biological neural networks from the viewpoint of information processing. This will allow us to design abstract models of artificial neural networks, which can then be simulated and analyzed. Although the models which have been proposed to explain the structure of the brain and the nervous systems of some animals are different in many respects, there is a general consensus that the essence of the operation of neural ensembles is "control through communication".

Animal nervous systems are composed of thousands or millions of interconnected cells. Each one of them is a very complex arrangement which deals with incoming signals in many different ways. However, neurons are rather slow when compared to electronic logic gates. These can achieve switching times of a few nanoseconds, whereas neurons need several milliseconds to react to a stimulus. Nevertheless the brain is capable of solving problems which no digital computer can yet efficiently deal with.

Massive and hierarchical networking of the brain seems to be the fundamental precondition for the emergence of consciousness and complex behaviour. So far, however, biologists and neurologists have concentrated their research on uncovering the properties of individual neurons. Today, the mechanisms for the production and transport of signals from one neuron to the other are well-understood physiological phenomena, but how these individual systems cooperate to form complex and massively parallel systems capable of incredible information processing feats has not yet been completely elucidated. Mathematics, physics, and computer science can provide invaluable help in the study of these complex systems. It is not surprising that the study of the brain has become one of the most interdisciplinary areas of scientific research in recent years.

However, we should be careful with the metaphors and paradigms commonly introduced when dealing with the nervous system. It seems to be a constant in the history of science that the brain has always been compared to the most complicated contemporary artefact produced by human industry. In ancient times the brain was compared to a pneumatic machine, in the Renaissance to clockwork and at the end of the last century to the telephone network. There are some today who consider computers the paradigm par excellence of a nervous system. It is rather paradoxical that when John von Neumann wrote his classical description of future universal computers, he tried to choose terms that would describe computers in terms of brains, not brains in terms of computers.

The nervous system of an animal is an information processing totality. The sensory inputs that are signals from the environment are coded and processed to evoke the appropriate response. Biological neural networks are just one of many possible solutions to the problem of processing information. The main difference between neural networks and conventional computer systems is the massive parallelism and redundancy which they exploit in order to deal with the unreliability of the individual computing units.

Moreover, biological neural networks are self-organising systems and each individual neuron is also a delicate self-organising structure capable of processing information in many different ways.

In this chapter we will study the information processing capabilities of complex hierarchical networks of simple computing units. We deal with systems whose structure is only partially predetermined. Some parameters modify the capabilities of the network and it is our task to find the best combination for the solution of a given problem. The adjustment of the parameters will be done through a learning algorithm that is not through explicit programming but through an automatic adaptive method.

A cursory review of the relevant literature on artificial neural networks leaves the impression of a chaotic mixture of very different network topologies and learning algorithms. Commercial neural network simulators sometimes offer several dozens of possible models. The large number of proposals has led to a situation in which each single model appears as part of a big puzzle whereas the bigger picture is absent. We will try to solve this puzzle by systematically introducing and discussing each of the neural network models in relation to the others.

Our approach consists of stating and answering the following questions: what information processing capabilities emerge in hierarchical systems of primitive computing units? What can be computed with these networks? How can these networks determine their structure in a self-organising manner? We start by considering biological systems. Artificial neural networks have aroused so much interest in recent years, not only because they exhibit interesting properties, but also because they try to mirror the kind of information processing capabilities of nervous systems. Since information processing consists of transforming signals, we deal with the biological mechanisms for their generation and transmission in this chapter.

We discuss those biological processes by which neurons produce signals, and absorb and modify them in order to retransmit the result. In this way biological neural networks give us a clue regarding the properties which would be interesting to include in our artificial networks.

3.3 Models of Computation

Artificial neural networks can be considered as just another approach to the problem of computation. The first formal definitions of computability were proposed in the 1930s and ’40s and at least five different alternatives were studied at the time. The computer era was started, not with one single approach, but with a contest of alternative computing models. We all know that the von Neumann computer emerged as the undisputed winner in this confrontation, but its triumph did not lead to the dismissal of the other computing models. Figure 3.1 shows the five principal contenders:

3.3.1 The Mathematical Model

Mathematicians avoided dealing with the problem of a function’s computability until the beginning of this century. This happened not just because existence theorems were considered sufficient to deal with functions, but mainly because nobody had come up with a satisfactory definition of computability, certainly a relative concept which depends on the specific tools that can be used. The general solution for algebraic equations of degree five, for example, cannot be formulated using only algebraic functions, yet this can be done if a more general class of functions is allowed as computational primitives.

The squaring of the circle, to give another example, is impossible using ruler and compass, but it has a trivial real solution. If we want to talk about computability we must therefore specify which tools are available. We can start with the idea that some primitive functions and composition rules are “obviously” computable. All other functions which can be expressed in terms of these primitives and composition rules are then also computable.

David Hilbert, the famous German mathematician, was the first to state the conjecture that a certain class of functions contains all intuitively computable functions. Hilbert was referring to the primitive recursive functions, the class of functions which can be constructed from the zero and successor function using composition, projection, and a deterministic number of iterations (primitive recursion). However, in 1928, Wilhelm Ackermann was able to find a computable function which is not primitive recursive. This led to the definition of the general recursive functions.

In this formalism, a new composition rule has to be introduced, the so-called μ operator, which is equivalent to an indeterminate recursion or a lookup in an infinite table. At the same time Alonzo Church and collaborators developed the lambda calculus, another alternative to the mathematical definition of the computability concept. In 1936, Church and Kleene were able to show that the general recursive functions can be expressed in the formalism of the lambda calculus.

This led to the Church thesis that computable functions are the general recursive functions. David Deutsch has recently added that this thesis should be considered to be a statement about the physical world and be given the same status as a physical principle. He thus speaks of a "Church principle".

3.3.2 The Logic-Operational Model (Turing Machines)

Alan Turing introduced another kind of computing model. The advantage of his approach is that it consists in an operational, mechanical model of computability. A Turing machine is composed of an infinite tape, in which symbols can be stored and read again. A read-write head can move to the left or to the right according to its internal state, which is updated at each step. The Turing thesis states that computable functions are those which can be computed with this kind of device. It was formulated concurrently with the Church thesis and Turing was able to show almost immediately that they are equivalent. The Turing approach made clear for the first time what "programming" means, curiously enough at a time when no computer had yet been built.

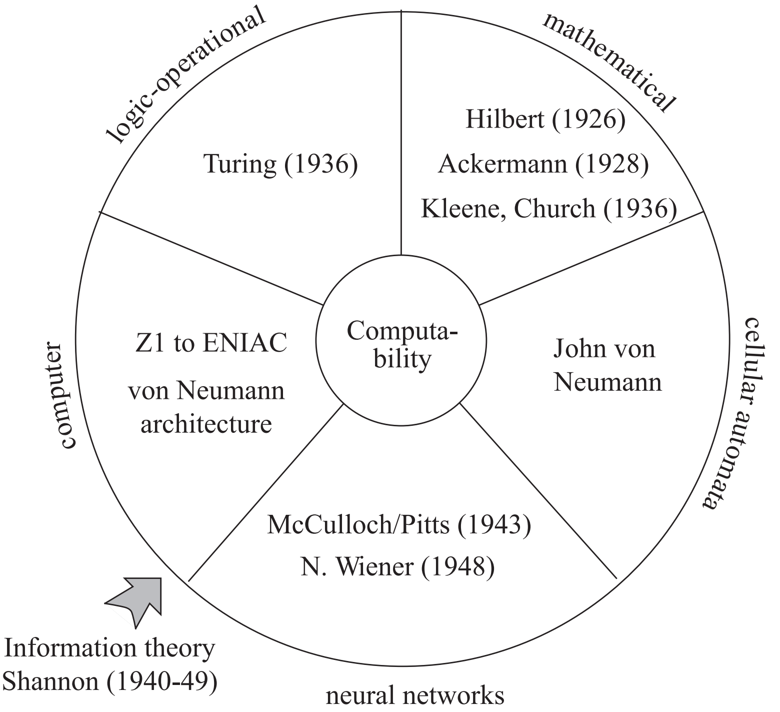


Fig. 3.1 Five models of computation

3.3.3 The Computer Model

The first electronic computing devices were developed in the 1930s and '40s. Since then, "computation-with-the-computer" has been regarded as computability itself. However the first engineers developing computers were for the most part unaware of Turing's or Church's research. Konrad Zuse, for example, developed in Berlin between 1938 and 1944 the computing machines Z1 and Z3 which were programmable but not universal, because they could not reach the whole space of the computable functions. Zuse's machines were able to process a sequence of instructions but could not iterate. Other computers of the time, like the Mark I built at Harvard, could iterate a constant number of times but were incapable of executing open-ended iterations (WHILE loops).

Therefore the Mark I could compute the primitive but not the general recursive functions. Also the ENIAC, which is usually hailed as the world's first electronic computer, was incapable of dealing with open-ended loops, since iterations were determined by specific connections between modules of the machine. It seems that the first universal computer was the Mark I built in Manchester. This machine was able to cover all computable functions by making use of conditional branching and self-modifying programs, which is one possible way of implementing indexed addressing.

3.3.4 Cellular Automata

The history of the development of the first mechanical and electronic computing devices shows how difficult it was to reach a consensus on the architecture of universal computers. Aspects such as the economy or the dependability of the building blocks played a role in the discussion, but the main problem was the definition of the minimal architecture needed for universality. In machines like the Mark I and the ENIAC there was no clear separation between memory and processor, and both functional elements were intertwined. Some machines still worked with base 10 and not 2, some were sequential and others parallel.

John Von Neumann, who played a major role in defining the architecture of sequential machines, analyzed at that time a new computational model which he called cellular automata. Such automata operate in a "computing space" in which all data can be processed simultaneously. The main problem for cellular automata is communication and coordination between all the computing cells.

This can be guaranteed through certain algorithms and conventions. It is not difficult to show that all computable functions, in the sense of Turing, can also be computed with cellular automata, even of the one-dimensional type, possessing only a few states. Turing himself considered this kind of computing model at one point in his career. Cellular automata as computing model resemble massively parallel multiprocessor systems of the kind that has attracted considerable interest recently.

3.3.5 The Biological Model (Neural Networks)

The explanation of important aspects of the physiology of neurons set the stage for the formulation of artificial neural network models which do not operate sequentially, as Turing machines do. Neural networks have a hierarchical multilayered structure which sets them apart from cellular automata, so that information is transmitted not only to the immediate neighbours but also to more distant units. In artificial neural networks one can connect each unit to any other. In contrast to conventional computers, no program is handed over to the hardware such a program has to be created, that is, the free parameters of the network have to be found adaptively.

Although, neural networks and cellular automata are potentially more efficient than conventional computers in certain application areas, at the time of their conception they were not yet ready to take center stage. The necessary theory for harnessing the dynamics of complex parallel systems is still being developed right before our eyes. In the meantime, conventional computer technology has made great strides.

There is no better illustration for the simultaneous and related emergence of these various computability models than the life and work of John von Neumann himself. He participated in the definition and development of at least three of these models: in the architecture of sequential computers the theory of cellular automata and the first neural network models. He also collaborated with Church and Turing in Princeton.

Artificial neural networks have, as initial motivation, the structure of biological systems, and constitute an alternative computability paradigm. For that reason we will review some aspects of the way in which biological systems perform information processing. The fascination which still pervades this research field has much to do with the points of contact with the surprisingly elegant methods used by neurons in order to process information at the cellular level. Several million years of evolution have led to very sophisticated solutions to the problem of dealing with an uncertain environment.

3.4 Elements of a Computing Model

What are the elementary components of any conceivable computing model? In the theory of general recursive functions, for example, it is possible to reduce any computable function to some composition rules and a small set of primitive functions. For a universal computer, we ask about the existence of a minimal and sufficient instruction set. For an arbitrary computing model the following metaphoric expression has been proposed:

Computation = storage + transmission + processing

The mechanical computation of a function presupposes that these three elements are present, that is, that data can be stored, communicated to the functional units of the model and transformed. It is implicitly assumed that a certain coding of the data has been agreed upon. Coding plays an important role in information processing because, as Claude Shannon showed in 1948, when noise is present information can still be transmitted without loss, if the right code with the right amount of redundancy is chosen.

Modern computers transform storage of information into a form of information transmission. Static memory chips store a bit as a circulating current until the bit is read. Turing machines store information in an infinite tape, whereas transmission is performed by the read-write head. Cellular automata store information in each cell, which at the same time is a small processor.

3.5 Networks of Neurons

In biological neural networks information is stored at the contact points between different neurons, the so-called synapses. Later we will discuss what role these elements play for the storage, transmission, and processing of information. Other forms of storage are also known, because neurons are themselves complex systems of self-organising signalling. In the next few pages we cannot do justice to all this complexity, but we analyse the most salient features and, with the metaphoric expression given above in mind, we will ask: how do neurons compute?

3.6 Structure of the Neurons

Nervous systems possess global architectures of variable complexity, but all are composed of similar building blocks, the neural cells or neurons. They can perform different functions, which in turn leads to a very variable morphology. If we analyze the human cortex under a microscope, we can find several different types of neurons. Although the neurons have very different forms, it is possible to recognise a hierarchical structure of six different layers. Each one has specific functional characteristics. Sensory signals, for example, are transmitted directly to the fourth layer and from there processing is taken over by other layers. Neurons receive signals and produce a response.

3.7 Transmission of Information

The fundamental problem of any information processing system is the transmission of information, as data storage can be transformed into a recurrent transmission of information between two points. Biologists have known for more than 100 years that neurons transmit information using electrical signals. Because we are dealing with biological structures, this cannot be done by simple electronic transport as in metallic cables.

Evolution arrived at another solution involving ions and semi permeable membranes. Our body consists mainly of water, 55% of which is contained within the cells and 45% forming its environment. The cells preserve their identity and biological components by enclosing the protoplasm in a membrane made of a double layer of molecules that form a diffusion barrier. Some salts, present in our body, dissolve in the intracellular and extracellular fluid and dissociate into negative and positive ions. Sodium chloride, for example, dissociates into positive sodium ions (Na+) and negative chlorine ions (Cl−).

Other positive ions present in the interior or exterior of the cells are potassium (K+) and calcium (Ca2+). The membranes of the cells exhibit different degrees of permeability for each one of these ions. The permeability is determined by the number and size of pores in the membrane, the so-called ionic channels. These are macromolecules with forms and charges which allow only certain ions to go from one side of the cell membrane to the other.

Channels are selectively permeable to sodium, potassium or calcium ions. The specific permeability of the membrane leads to different distributions of ions in the interior and the exterior of the cells and this, in turn, to the interior of neurons being negatively charged with respect to the extracellular fluid.

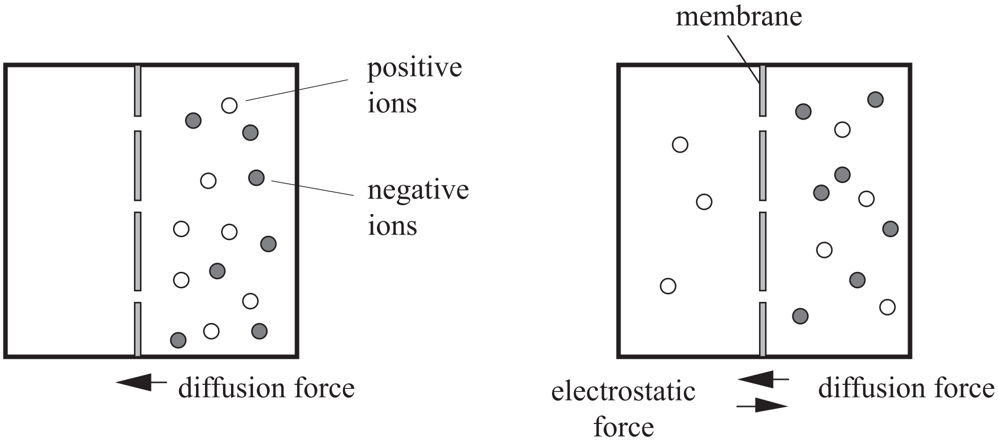


Fig. 3.2 Diffusion of ions through a membrane

Above figure illustrates this phenomenon. A box is divided into two parts separated by a membrane permeable only to positive ions. Initially the same number of positive and negative ions is located in the right side of the box. Later, some positive ions move from the right to the left through the pores in the membrane. This occurs because atoms and molecules have a thermo-dynamical tendency to distribute homogeneously in space by the process called diffusion. The process continues until the electrostatic repulsion from the positive ions on the left side balances the diffusion potential.

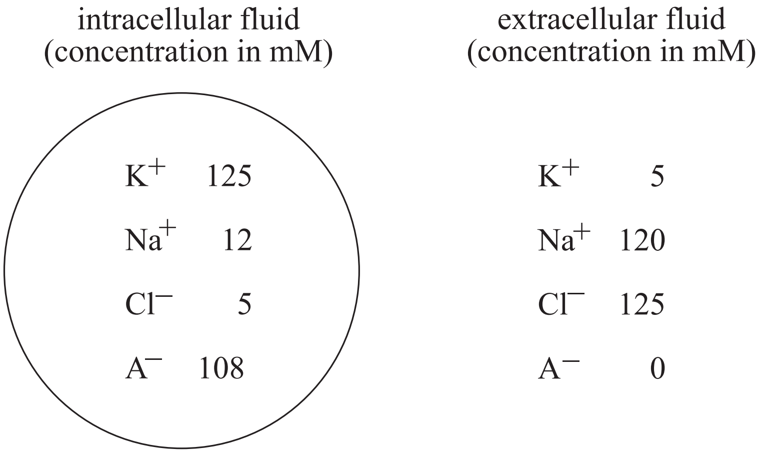
A potential difference, called the reversal potential, is established and the system behaves like a small electric battery. In a cell, if the initial concentration of potassium ions in its interior is greater than in its exterior, positive potassium ions will diffuse through the open potassium-selective channels. If these are the only ionic channels, negative ions cannot disperse through the membrane. The interior of the cell becomes negatively charged with respect to the exterior, creating a potential difference between both sides of the membrane. This balances the diffusion potential, and, at some point, the net flow of potassium ions through the membrane falls to zero. The system reaches a steady state. The potential difference E for one kind of ion is given by the Nernst formula

$$E = k\,(\ln(c_o)\text{-}\ln(c_i))$$

Where, c_i is the concentration inside the cell, co the concentration in the extracellular fluid and k is proportionality constant. For potassium ions the equilibrium potential is −80 mV.

Because there are several different concentrations of ions inside and outside of the cell, the question is, ‘what is the potential difference which is finally reached?’ The exact potential in the interior of the cell depends on the mixture of concentrations. A typical cell’s potential is −70 mV, which is produced mainly by the ion concentrations shown in figure below (A− designates negatively charged bio-molecules). The two main ions in the cell are sodium and potassium. Equilibrium potential for sodium lies around 58 mV.

The cell reaches a potential between −80 mV and 58 mV. The cell’s equilibrium potential is nearer to the value induced by potassium, because the permeability of the membrane to potassium is greater than to sodium. There is a net outflow of potassium ions at this potential and a net inflow of sodium ions. However, the sodium ions are less mobile because fewer open channels are available. In the steady state the cell membrane experiences two currents of ions trying to reach their individual equilibrium potential. An ion pump guarantees that the concentration of ions does not change with time.



intracellular fluid (concentration in mM)		extracellular fluid (concentration in mM)	
K^+	125	K^+	5
Na^+	12	Na^+	120
Cl^-	5	Cl^-	125
A^-	108	A^-	0

Fig. 3.3 Ion concentrations inside and outside a cell

The British scientists Alan Hodgkin and Andrew Huxley were able to show that it is possible to build an electric model of the cell membrane based on very simple assumptions. The membrane behaves as a capacitor made of two isolated layers of lipids. It can be charged with positive or negative ions. The different concentrations of several classes of ions in the interior and exterior of the cell provide an energy source capable of negatively polarising the interior of the cell. Figure below shows a diagram of the model proposed by Hodgkin and Huxley. The specific permeability of the membrane for each class of ion can be modelled like a conductance (the reciprocal of resistance).

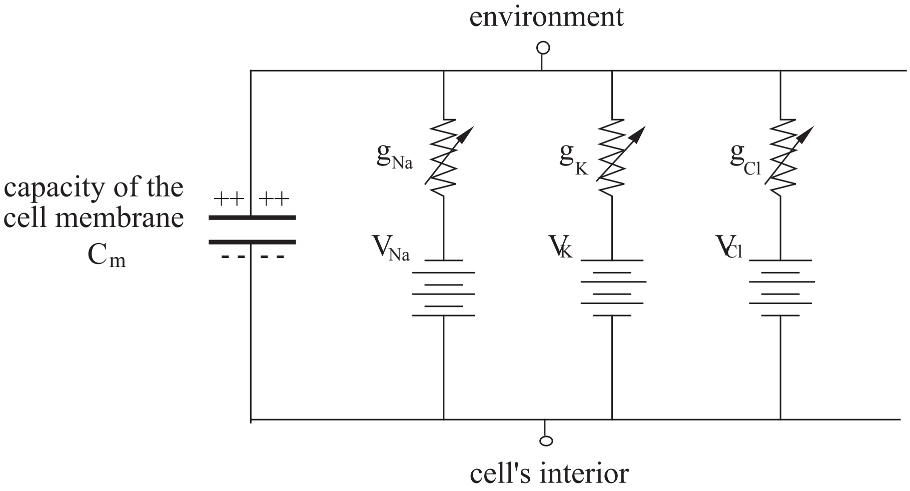


Fig. 3.4 The Hodgkin–Huxley model of a cell membrane

The electric model is a simplification, because there are other classes of ions and electrically charged proteins present in the cell. In the model, three ions compete to create a potential difference between the interior and exterior of the cell. The conductances g_{Na}, g_K, and g_L reflect the permeability of the membrane to sodium, potassium, and leakages that is the number of open channels of each class. A signal can be produced by modifying the polarity of the cell through changes in the conductances g_{Na} and g_K. By making g_{Na} larger and the mobility of sodium ions greater than the mobility of potassium ions, the polarity of the cell changes from −70 mV to a positive value, nearer to the 58 mV at which sodium ions reach equilibrium.

If the conductance g_K then becomes larger and g_{Na} falls back to its original value, the interior of the cell becomes negative again, overshooting in fact by going below −70 mV. To generate a signal, a mechanism for depolarising and polarising the cell in a controlled way is necessary. The conductance and resistance of a cell membrane in relation to the different classes of ions depends on its permeability. This can be controlled by opening or closing excitable ionic channels. In addition to the static ionic channels already mentioned, there is another class which can be electrically controlled. These channels react to a depolarisation of the cell membrane.

When this happens, that is, when the potential of the interior of the cell in relation to the exterior reaches a threshold, the sodium-selective channels open automatically and positive sodium ions flow into the cell making its interior positive. This in turn leads to the opening of the potassium-selective channels and positive potassium ions flow to the exterior of the cell, restoring the original negative polarisation.

Figure below shows a diagram of an electrically controlled sodium-selective channel which lets only sodium ions flow across. This effect is produced by the small aperture in the middle of the channel which is negatively charged (at time t = 1). If the interior of the cell becomes positive relative to the exterior, some negative charges are displaced in the channel and this produces the opening of a gate (t = 2). Sodium ions flow through the channel and into the cell. After a short time the second gate is closed and the ionic channel is sealed (t = 3). The opening of the channel corresponds to a change of membrane conductivity as explained above.

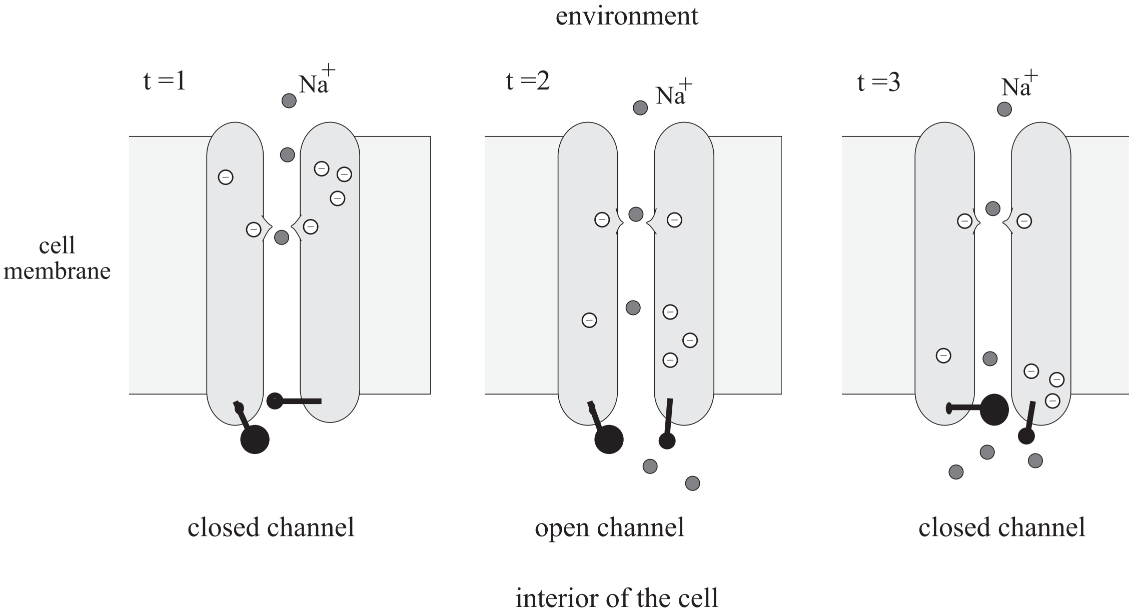


Fig. 3.5 Electrically controlled ionic channels

Static and electrically controlled ionic channels are not only found in neurons. As in any electrical system there are charge losses which have to be continuously balanced. A sodium ion pump has shown in figure below transports the excess of sodium ions out of the cell and, at the same time, potassium ions into its interior. The ion pump consumes adenosine triphosphate (ATP), a substance produced by the mitochondria, helping to stabilise the polarisation potential of −70 mV. The ion pump is an example of a self-regulating system, because it is accelerated or decelerated by the differences in ion concentrations on both sides of the membrane. Ion pumps are constantly active and account for a considerable part of the energy requirements of the nervous system.

Neural signals are produced and transmitted at the cell membrane. The signals are represented by depolarisation waves travelling through the axons in a self-regenerating manner. Figure below shows the form of such a depolarisation wave, called an action potential. The x-dimension is shown horizontally and the diagram shows the instantaneous potential in each segment of the axon. An action potential is produced by an initial depolarisation of the cell membrane. The potential increases from −70 mV up to +40 mV. After some time the membrane potential becomes negative again but it overshoots, going as low as −80 mV.

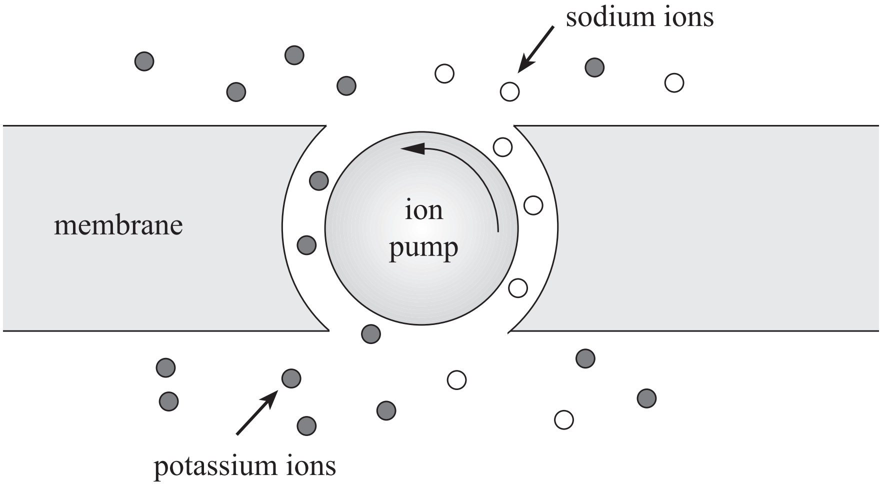


Fig. 3.6 Sodium and potassium ion pump

The cell recovers gradually and the cell membrane returns to the initial potential. The switching time of the neurons is determined, as in any resistor-capacitor configuration, by the RC constant. In neurons, 3.4 milliseconds is a typical value for this constant.

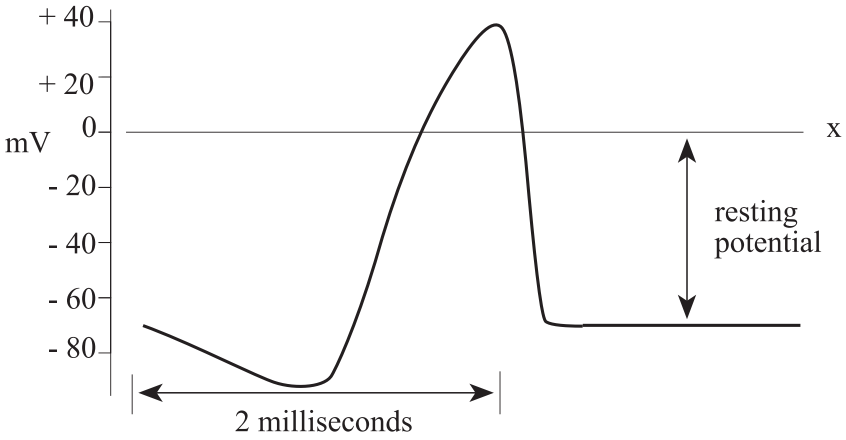


Fig. 3.7 Typical form of the action potential

Figure above shows an action potential travelling through an axon. A local perturbation, produced by the signals arriving at the dendrites, leads to the opening of the sodium-selective channels in a certain region of the cell membrane. The membrane is thus depolarised and positive sodium ions flow into the cell. After a short delay, the outward flow of potassium ions compensates the depolarisation of the membrane. Both perturbations the opening of the sodium and potassium-selective channels – are transmitted through the axon like falling dominos. In the entire process only local energy is consumed, that is, only the energy stored in the polarised membrane itself. The action potential is thus a wave of Na+ permeability increase followed by a wave of K+ permeability increase. It is easy to see that charged particles only move a short distance in the direction of the perturbation, only as much as is necessary to perturb the next channels and bring the next “domino” to fall.

Figure below also shows how impulse trains are produced in the cells. After a signal is produced a new one follows. Each neural signal is an all-or nothing self-propagating regenerative event as each signal has the same for and amplitude. At this level we can safely speak about digital transmission of information.

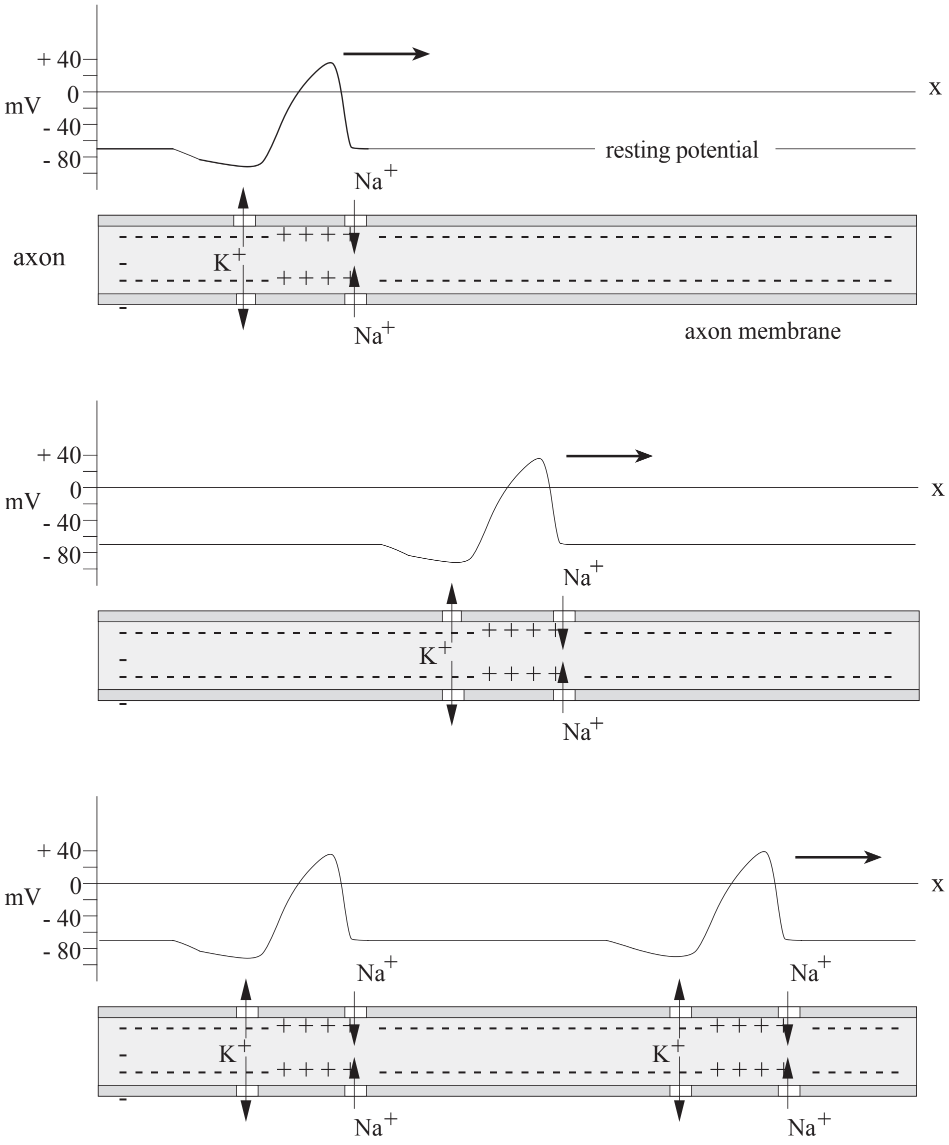


Fig. 3.8 Transmission of an action potential

With this picture of the way an action potential is generated in mind, it is easy to understand the celebrated Hodgkin–Huxley differential equation which describes the instantaneous variation of the cell's potential V as a function of the conductance of sodium, potassium and leakages (g_{Na}, g_K, g_L) and of the equilibrium potentials for all three groups of ions called V_{Na}, V_K and V_L with respect to the current potential:

$$\frac{dV}{dt} = \frac{1}{cm}(I - g\mathrm{N}_a(V - V_{Na}) - g_K(V - V_K) - g_L(V - V_L))$$

In this equation Cm is the capacitance of the cell membrane. The terms $V - V_{Na}$, $V - V_K$, $V - V_L$ are the electromotive forces acting on the ions. Any variation of the conductances translates into a corresponding variation of the cell's potential V. The variations of g_{Na} and g_K are given by differential equations which describe their oscillations. The conductance of the leakages g_L can be taken as a constant.

A neuron codes its level of activity by adjusting the frequency of the generated impulses. This frequency is greater for a greater stimulus. In some cells the mapping from stimulus to frequency is linear in a certain interval. This means that information is transmitted from cell to cell using what engineers call frequency modulation. This form of transmission helps to increase the accuracy of the signal and to minimise the energy consumption of the cells.

3.8 Information Processing at the Neurons and Synapses

Neurons transmit information using action potentials. The processing of this information involves a combination of electrical and chemical processes, regulated for the most part at the interface between neurons, the synapses. Neurons transmit information not only by electrical perturbations. Although electrical synapses are also known, most synapses make use of chemical signalling.

The synapse appears as a thickening of the axon. The small vacuoles in the interior, the synaptic vesicles, contain chemical transmitters. The small gap between a synapse and the cell to which it is attached is known as the synaptic gap. When an electric impulse arrives at a synapse, the synaptic vesicles fuse with the cell membrane; study the figure below. The transmitters flow into the synaptic gap and some attach themselves to the ionic channels, as in our example. If the transmitter is of the right kind, the ionic channels are opened and more ions can now flow from the exterior to the interior of the cell.

The cell's potential is altered in this way. If the potential in the interior of the cell is increased, this helps prepare an action potential and the synapse causes an excitation of the cell. If negative ions are transported into the cell, the probability of starting an action potential is decreased for some time and we are dealing with an inhibitory synapse. Synapses determine a direction for the transmission of information. Signals flow from one cell to the other in a well-defined manner. This will be expressed in artificial neural networks models by embedding the computing elements in a directed graph.

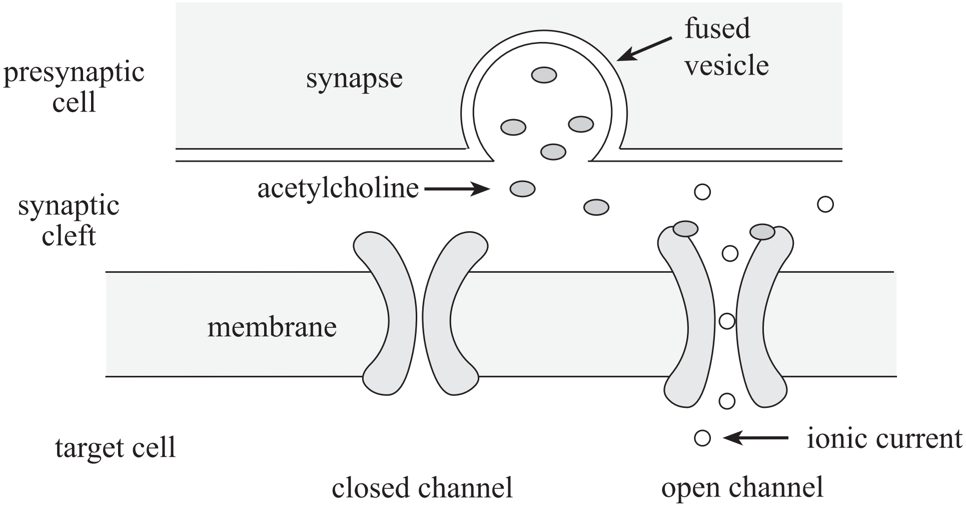


Fig. 3.9 Chemical signalling at the synapse

A well-defined direction of information flow is a basic element in every computing model, and is implemented in digital systems by using diodes and directional amplifiers. The interplay between electrical transmission of information in the cell and chemical transmission between cells is the basis for neural information processing. Cells process information by integrating incoming signals and by reacting to inhibition. The flow of transmitters from an excitatory synapse leads to a depolarisation of the attached cell.

The depolarisation must exceed a threshold, that is, enough ionic channels have to be opened in order to produce an action potential. This can be achieved by several pulses arriving simultaneously or within a short time interval at the cell. If the quantity of transmitters reaches a certain level and enough ionic channels are triggered, the cell reaches its activation threshold. At this moment an action potential is generated at the axon of this cell.

In most neurons, action potentials are produced at the so-called axon hillock, the part of the axon nearest to the cell body. In this region of the cell, the number of ionic channels is larger and the cell's threshold lower. The dendrites collect the electrical signals which are transmitted electronically through the cytoplasm. The transmission of information at the dendrites makes use of additional electrical effects. Streams of ions are collected at the dendrites and brought to the axon hillock. There is spatial summation of information when signals coming from different dendrites are collected and temporal summation when signals arriving consecutively are combined to produce a single reaction. In some neurons not only the axon hillock but also the dendrites can produce action potentials. In this case information processing at the cell is more complex than in the standard case.

It can be shown that digital signals combined in an excitatory or inhibitory way can be used to implement any desired logical function. The number of computing units required can be reduced if the information is not only transmitted but also weighted. This can be achieved by multiplying the signal by a constant. Such is the kind of processing we find at the synapses. Each signal is an all-or-none event but the number of ionic channels triggered by the signal is different from synapse to synapse. It can happen that a single synapse can push a cell to fire an action potential, but other synapses can achieve this only by simultaneously exciting the cell. With each synapse i $(1\leq i \leq n)$ we can therefore associate a numerical weight w_i. If all synapses are activated at the same time, the information which will be transmitted is $w_1 + w_2 + \cdot \cdot \cdot + w_n$. If this value is greater than the cell's threshold, the cell will fire a pulse.

It follows from this description that neurons process information at the membrane. The membrane regulates both transmission and processing of information. Summation of signals and comparison with a threshold is a combined effect of the membrane and the cytoplasm. If a pulse is generated, it is transmitted and the synapses set some transmitter molecules free. From this description an abstract neuron can be modelled which contains dendrites, a cell body and an axon. The same three elements will be present in our artificial computing units.

3.9 Storage of Information – Learning

In neural networks information is stored at the synapses. Some other forms of information storage may be present, but they are either still unknown or not very well understood. A synapse's efficiency in eliciting the depolarisation of the contacted cell can be increased if more ionic channels are opened. In recent years NMDA receptors have been studied because they exhibit some properties which could help explain some forms of learning in neurons.

NMDA receptors are ionic channels permeable for different kinds of molecules, like sodium, calcium, or potassium ions. These channels are blocked by a magnesium ion in such a way that the permeability for sodium and calcium is low. If the cell is brought up to a certain excitation level, the ionic channels lose the magnesium ion and become unblocked. The permeability for Ca2+ ions increases immediately. Through the flow of calcium ions a chain of reactions is started which produces a durable change of the threshold level of the cell. Figure below shows a diagram of this process.

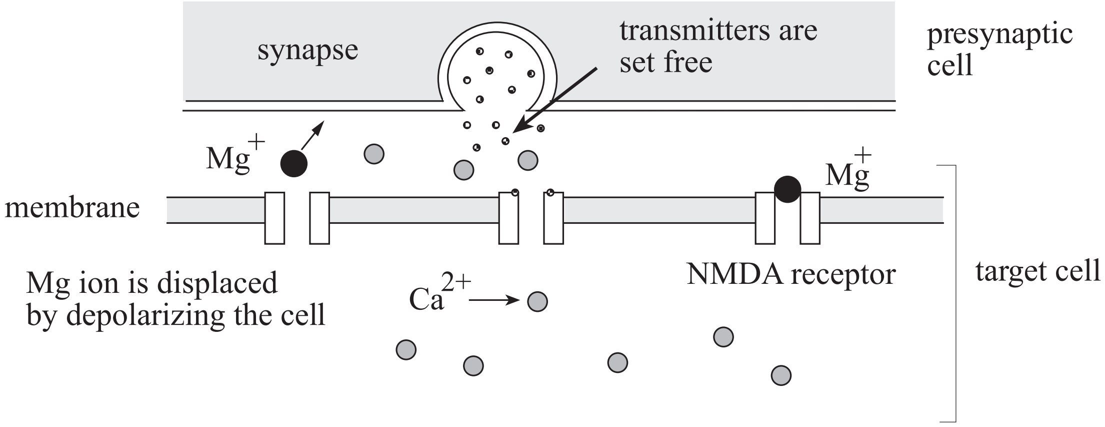

synapse
transmitters are set free
presynaptic cell
Mg+
Mg+
membrane
Mg ion is displaced by depolarizing the cell
Ca2+
NMDA receptor
target cell

Fig. 3.10 Unblocking of an NMDA receptor

NMDA receptors are just one of the mechanisms used by neurons to increase their plasticity that is their adaptability to changing circumstances. Through the modification of the membrane's permeability a cell can be trained to fire more often by setting a lower firing threshold. NMDA receptors also offer an explanation for the observed phenomenon that cells which are not stimulated to fire tend to set a higher firing threshold. The stored information must be refreshed periodically in order to maintain the optimal permeability of the cell membrane.

This kind of information storage is also used in artificial neural networks. Synaptic efficiency can be modelled as a property of the edges of the network. The networks of neurons are thus connected through edges with different transmission efficiencies. Information flowing through the edges is multiplied by a constant which reflects their efficiency. One of the most popular learning algorithms for artificial neural networks is Hebbian learning. The efficiency of synapses is increased any time the two cells which are connected through this synapse fire simultaneously and is decreased when the firing states of the two cells are uncorrelated. The NMDA receptors act as coincidence detectors of presynaptic and postsynaptic activity, which in turn leads to greater synaptic efficiency.

Summary

- Since 1943, when Warren McCulloch and Walter Pitts presented the first model of artificial neurons, new and more sophisticated proposals have been made from decade to decade.
- Artificial neural networks are an attempt at modelling the information processing capabilities of nervous systems.
- Animal nervous systems are composed of thousands or millions of interconnected cells.
- The nervous system of an animal is an information processing totality.
- The main difference between neural networks and conventional computer systems is the massive parallelism and redundancy which they exploit in order to deal with the unreliability of the individual computing units.
- Mathematicians avoided dealing with the problem of a function's computability until the beginning of this century.
- David Hilbert, the famous German mathematician, was the first to state the conjecture that a certain class of functions contains all intuitively computable functions.
- A Turing machine is composed of an infinite tape, in which symbols can be stored and read again.
- A read-write head can move to the left or to the right according to its internal state, which is updated at each step.
- The first electronic computing devices were developed in the 1930s and '40s.
- The history of the development of the first mechanical and electronic computing devices shows how difficult it was to reach a consensus on the architecture of universal computers.
- John von Neumann, who played a major role in defining the architecture of sequential machines, analyzed at that time a new computational model which he called cellular automata.
- In biological neural networks information is stored at the contact points between different neurons, the so-called synapses.
- Nervous systems possess global architectures of variable complexity, but all are composed of similar building blocks, the neural cells or neurons.
- A well-defined direction of information flow is a basic element in every computing model, and is implemented in digital systems by using diodes and directional amplifiers.
- The depolarisation must exceed a threshold, that is, enough ionic channels have to be opened in order to produce an action potential.

References

- Gurney, K. &Gurney, N. K., 1997. *An Introduction to Neural Networks*, UCL Press.
- Dreyfus, G., 2005. *Neural Networks: Methodology And Applications*, Birkhäuser.
- *The Biological Paradigm*, [Online] Available at: <http://page.mi.fu-berlin.de/rojas/neural/chapter/K1.pdf> [Accessed 18 July 2012].
- Prof. Smith, L., *An Introduction to Neural Networks*, [Online] Available at: <http://www.cs.stir.ac.uk/~lss/NNIntro/InvSlides.html> [Accessed 18 July 2012].
- 2009. *Intro to Neural Networks*, [Video Online] Available at: <http://www.youtube.com/watch?v=DG5-UyRBQD4> [Accessed 18 July 2012].
- Prof. Sengupta, S., 2009. *Lec-1 Introduction to Artificial Neural Networks*, [Video Online] Available at: <http://www.youtube.com/watch?v=xbYgKoG4x2g> [Accessed 18 July 2012].

Recommended Reading

- Rojas, R., 1996. *Neural Networks: A Systematic Introduction*, Springer.
- Haykin, S. S., 1999. *Neural networks: A comprehensive foundation*, 2nd ed. Prentice Hall.
- Freeman, 1991. *Neural Networks: Algorithms, Applications, and Programming Techniques*, Pearson Education India.

Self Assessment

1. Artificial neural networks are an attempt at modelling the information processing capabilities of __________.
   a. neural computation
   b. nervous systems
   c. neural ensembles
   d. artificial neural

2. Which of the following statement is false?
   a. Animal nervous systems are composed of thousands or millions of interconnected Artificial neural.
   b. The structure of the brain and the nervous systems of some animals are different in many respects.
   c. Massive and hierarchical networking of the brain seems to be the fundamental precondition.
   d. Renaissance to clockwork and at the end of the last century to the telephone network.

3. ___________ was the first to state the conjecture that a certain class of functions contains all intuitively computable functions.
   a. David Hilbert
   b. Newton
   c. Wilhelm Ackermann
   d. Alonzo Church

4. Match the columns:

1. Alan Turing	A. General recursive functions
2. Church	B. Primitive recursive functions
3. Alonzo Church	C. Computing model
4. Hilbert	D. Lambda calculus

   a. 1-B, 2-A, 3-C, 4-D
   b. 1-D, 2-C, 3-B, 4-A
   c. 1-C, 2-A, 3-D, 4-B
   d. 1-C, 2-A, 3-C, 4-B

5. A read-write head can move to the left or to the right according to its ________ state.
   a. external
   b. input
   c. output
   d. internal

6. The first electronic computing devices were developed in the________.
   a. 1930
   b. 1975
   c. 1920
   d. 1990

7. __________ played a major role in defining the architecture of sequential machines.
   a. Konrad Zuse
   b. John von Neumann
   c. Alan Turing
   d. Hilbert

8. The fundamental problem of any information processing system is the transmission of __________.
   a. environment
   b. membranes
   c. information
   d. channels

9. Neural signals are produced and transmitted at the _______ membrane.
   a. cell
   b. neuron
   c. axons
   d. mitochondria

10. A well-defined direction of __________ flow is a basic element in every computing model.
   a. output
   b. data
   c. input
   d. information

Chapter IV

Artificial Neural Networks

Aim

The aim of this chapter is to:

- introduce artificial neural networks
- elucidate networks of primitive functions
- explain applications of threshold logic

Objectives

The objectives of this chapter are to:

- explain approximation of functions
- explicate synthesis of boolean functions
- elucidate the hadamard–walsh transform

Learning outcome

At the end of this chapter, you will be able to:

- distinguish between networks of functions
- understand classification of neural networks
- describe harmonic analysis of logical functions

4.1 Introduction

The short review of the properties of biological neurons in the previous sections is necessarily incomplete and can offer only a rough description of the mechanisms and processes by which neurons deal with information. Nerve cells are very complex self-organising systems which have evolved in the course of millions of years. How was this exquisitely fine-tuned information processing organs developed? Where do we find the evolutionary origin of consciousness?

The information processing capabilities of neurons depend essentially on the characteristics of the cell membrane. Ionic channels appeared very early in evolution to allow unicellular organisms to get some kind of feedback from the environment. Consider the case of a paramecium, a protozoan with cilia, which are hair like processes which provide it with locomotion. A paramecium has a membrane cell with ionic channels and its normal state is one in which the interior of the cell is negative with respect to the exterior. In this state the cilia around the membrane beat rhythmically and propel the paramecium forward.

If an obstacle is encountered, some ionic channels sensitive to contact open, let ions into the cell, and depolarise it. The depolarisation of the cell leads in turn to a reversing of the beating direction of the cilia and the paramecium swims backward for a short time. After the cytoplasm returns to its normal state, the paramecium swims forward, changing its direction of movement. If the paramecium is touched from behind, the opening of ionic channels leads to a forward acceleration of the protozoan. In each case, the paramecium escapes its enemies.

From these humble origins, ionic channels in neurons have been perfected over millions of years of evolution. In the protoplasm of the cell, ionic channels are produced and replaced continually. They attach themselves to those regions of the neurons where they are needed and can move laterally in the membrane, like icebergs in the sea. The regions of increased neural sensitivity to the production of action potentials are thus changing continuously according to experience. The electrical properties of the cell membrane are not totally predetermined. They are also a result of the process by which action potentials are generated.

Consider also the interior of the neurons. The number of biochemical reaction chains and the complexity of the mechanical processes occurring in the neuron at any given time have led some authors to look for its control system. Stuart Hameroff, for example, has proposed that the cytoskeleton of neurons does not just perform a static mechanical function, but in some way provides the cell with feedback control. It is well known that the proteins that form the microtubules in axons coordinate to move synaptic vesicles and other materials from the cell body to the synapses. This is accomplished through a coordinated movement of the proteins, configured like a cellular automaton.

Consequently, transmission, storage, and processing of information are performed by neurons exploiting many effects and mechanisms which we still do not understand fully. Each individual neuron is as complex as or more complex than any of our computers. For this reason, we will call the elementary components of artificial neural networks simply "computing units" and not neurons. In the mid-1980s, the PDP (Parallel Distributed Processing) group already agreed to this convention at the insistence of Francis Crick.

4.2 Artificial Neural Networks

The discussion in the last section is only an example of how important it is to define the primitive functions and composition rules of the computational model. If we are computing with a conventional von Neumann processor, a minimal set of machine instructions is needed in order to implement all computable functions. In the case of artificial neural networks, the primitive functions are located in the nodes of the network and the composition rules are contained implicitly in the interconnection pattern of the nodes, in the synchrony or asynchrony of the transmission of information, and in the presence or absence of cycles.

4.2.1 Networks of Primitive Functions

Figure below shows the structure of an abstract neuron with n inputs. Each input channel *i* can transmit a real value x_i. The primitive function f computed in the body of the abstract neuron can be selected arbitrarily. Usually the input channels have an associated weight, which means that the incoming information x_i is multiplied by the corresponding weight w_i. The transmitted information is integrated at the neuron (usually just by adding the different signals) and the primitive function is then evaluated.

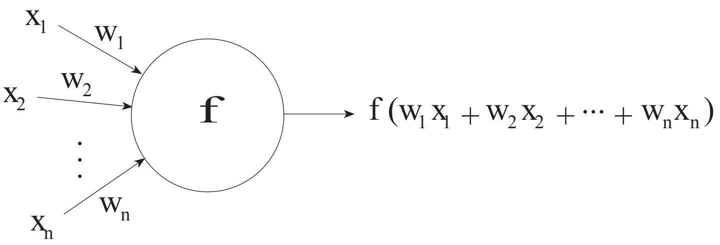


Fig. 4.1 An abstract neuron

If we conceive of each node in an artificial neural network as a primitive function capable of transforming its input in a precisely defined output, then artificial neural networks are nothing but networks of primitive functions. Different models of artificial neural networks differ mainly in the assumptions about the primitive functions used, the interconnection pattern, and the timing of the transmission of information.

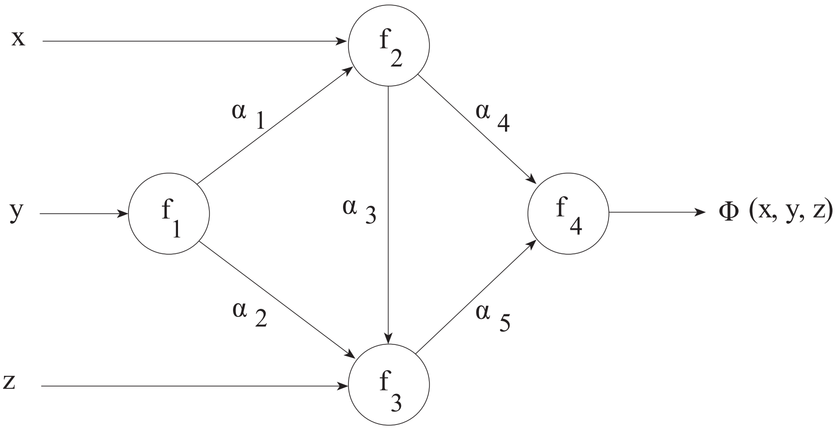


Fig. 4.2 Functional model of an artificial neural network

Typical artificial neural networks have the structure shown in figure above. The network can be thought of as a function which is evaluated at the point (*x, y, z*). The nodes implement the primitive functions f_1, f_2, f_3, f_4 which are combined to produce. The function implemented by a neural network will be called the network function. Different selections of the weights produce different network functions. Therefore, tree elements are particularly important in any model of artificial neural networks:

- The structure of the nodes
- The topology of the network
- The learning algorithm used to find the weights of the network

4.2.2 Approximation of Functions

An old problem in approximation theory is to reproduce a given function F: IR→IR either exactly or approximately by evaluating a given set of primitive functions. A classical example is the approximation of one-dimensional functions using polynomials or Fourier series. The Taylor series for a function *F* which is being approximated around the point x_0 is

$$F(x) = a_0 + a_1(x - x_0) + a_2(x - x_0)^2 + \cdots + a_n(x - x_0)^n + \cdots,$$

Whereby the constants $a_0, ..., a_n$ depend on the function F and its derivatives at x_0. Figure below shows how the polynomial approximation can be represented as a network of functions. The primitive functions z → 1, z → z^1, . . . , z → z^n are computed at the nodes. The only free parameters are the constants $a_0, ..., a_n$. The output node additively collects all incoming information and produces the value of the evaluated polynomial. The weights of the network can be calculated in this case analytically, just by computing the first $n + 1$ terms of the Taylor series of F. They can also be computed using a learning algorithm, which is the usual case in the field of artificial neural networks.

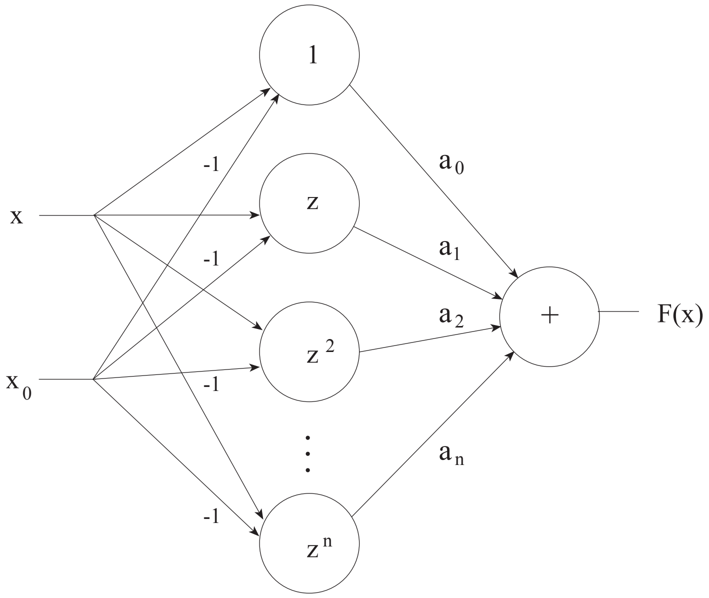


Fig. 4.3 A Taylor network

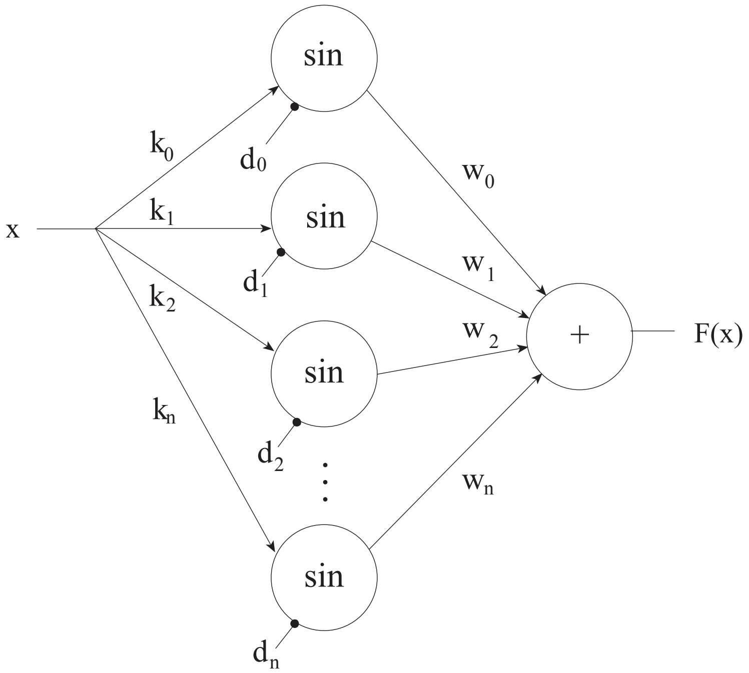


Fig. 4.4 A Fourier network

Figure above shows how a Fourier series can be implemented as a neural network. If the function F is to be developed as a Fourier series it has the form

$$F(x) = \sum_{i=0}^{\infty}(a_i \cos(ix) + b_i \sin(ix))$$

An artificial neural network with the sine as primitive function can implement a finite number of terms in the above expression. In Figure above shows the constants $k_0, \ldots, k_n$ determine the wave numbers for the arguments of the sine functions. The constants d0, . . . , d_n play the role of phase factors (with d0 = $\pi/2$, for example, we have $\sin(x+d_0)$ = $\cos(x)$) and we do not need to implement the cosine explicitly in the network. The constants $w_0, \ldots, w_n$ are the amplitudes of the Fourier terms. The network is indeed more general than the conventional formula because non-integer wave numbers are allowed as are phase factors which are not simple integer multiples of $\pi/2$.

The main difference between Taylor or Fourier series and artificial neural networks is, however, that the function *F* to be approximated is given not explicitly but implicitly through a set of input-output examples. We know *F* only at some points but we want to generalise as well as possible. This means that we try to adjust the parameters of the network in an optimal manner to reflect the information known and to extrapolate to new input patterns which will be shown to the network afterwards. This is the task of the learning algorithm used to adjust the network's parameters.

4.2.3 Caveat

At this point we must issue a warning to the reader: in the theory of artificial neural networks we do not consider the whole complexity of real biological neurons. We only abstract some general principles and content ourselves with different levels of detail when simulating neural ensembles. The general approach is to conceive each neuron as a primitive function producing numerical results at some points in time. These will be the kinds of model that we will discuss in the first chapters of this book.

However we can also think of artificial neurons as computing units which produce pulse trains in the way that biological neurons do. We can then simulate this behaviour and look at the output of simple networks. This kind of approach, although more closely related to the biological paradigm, is still a very rough approximation of the biological processes.

4.3 Historical and Bibliographical Remarks

Philosophical reflection on consciousness and the organ in which it could possibly be localised spans a period of more than two thousand years. Greek philosophers were among the first to speculate about the location of the soul. Several theories were held by the various philosophical schools of ancient times. Galenus, for example, identified nerve impulses with pneumatic pressure signals and conceived the nervous system as a pneumatic machine. Several centuries later Newton speculated that nerves transmitted oscillations of the ether.

Our present knowledge of the structure and physiology of neurons is the result of 100 years of special research in this field. The facts presented in this chapter were discovered between 1850 and 1950, with the exception of the NMDA receptors which were studied mainly in the last decade. The electrical nature of nerve impulses was postulated around 1850 by Emil du Bois- Reymond and Hermann von Helmholtz. The latter was able to measure the velocity of nerve impulses and showed that it was not as fast as was previously thought. Signals can be transmitted in both directions of an axon, but around 1901 Santiago Ram´on y Cajal postulated that the specific networking of the nervous cells determines a direction for the transmission of information.

This discovery made it clear that the coupling of the neurons constitutes a hierarchical system. Ram´on y Cajal was also the most celebrated advocate of the neuron theory. His supporters conceived the brain as a highly differentiated hierarchical organ, while the supporters of the reticular theory thought of the brain as a grid of undifferentiated axons and of dendrites as organs for the nutrition of the cell. Ram´on y Cajal perfected Golgi's staining method and published the best diagrams of neurons of his time, so well indeed that they are still in use.

The word neuron (Greek for nerve) was proposed by the Berlin Professor Wilhelm Waldeger after he saw the preparations of Ram´on y Cajal. The chemical transmission of information at the synapses was studied from 1920 to 1940. From 1949 to 1956, Hodgkin and Huxley explained the mechanism by which depolarisation waves are produced in the cell membrane. By experimenting with the giant axon of the squid they measured and explained the exchange of ions through the cell membrane, which in time led to the now famous Hodgkin–Huxley differential equations.

The Hodgkin–Huxley model was in some ways one of the first artificial neural models, because the postulated dynamics of the nerve impulses could be simulated with simple electric networks. At the same time the mathematical properties of artificial neural networks were being studied by researchers like Warren McCulloch, Walter Pitts, and John von Neumann. Ever since that time, research in the neurobiological field has progressed in close collaboration with the mathematics and computer science community.

4.4 Networks of Functions

We deal in this chapter with the simplest kind of computing units used to build artificial neural networks. These computing elements are a generalisation of the common logic gates used in conventional computing and, since they operate by comparing their total input with a threshold, this field of research is known as threshold logic.

4.4.1 Feed-forward and Recurrent Networks

The characteristics and structure of biological neural networks provides us with the initial motivation for a deeper inquiry into the properties of networks of abstract neurons. From the viewpoint of the engineer, it is important to define how a network should behave, without having to specify completely all of its parameters, which are to be found in a learning process. Artificial neural networks are used in many cases as a black box: a certain input should produce a desired output, but how the network achieves this result is left to a self-organising process.

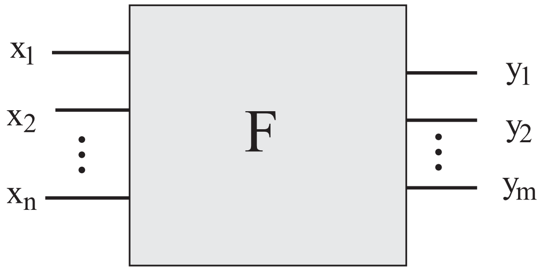


Fig. 4.5 A neural network as a black box

In general we are interested in mapping an *n*-dimensional real input (x_1, x_2, . . . , x_n) to an *m*-dimensional real output (y_1, y_2, . . . , y_m). A neural network thus behaves as a “mapping machine”, capable of modelling a function F: $\mathrm{IR}^n \rightarrow \mathrm{IR}^m$. If we look at the structure of the network being used, some aspects of its dynamics must be defined more precisely. When the function is evaluated with a network of primitive functions, information flows through the directed edges of the network. Some nodes compute values which are then transmitted as arguments for new computations. If there are no cycles in the network, the result of the whole computation is well-defined and we do not have to deal with the task of synchronising the computing units. We just assume that the computations take place without delay.

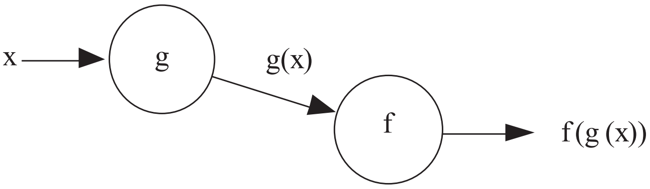


Fig. 4.6 Function composition

If the network contains cycles, however, the computation is not uniquely defined by the interconnection pattern and the temporal dimension must be considered. When the output of a unit is fed back to the same unit, we are dealing with a recursive computation without an explicit halting condition. We must define what we expect from the network: is the fixed point of the recursive evaluation the desired result or one of the intermediate computations? To solve this problem we assume that every computation takes a certain amount of time at each node (for example a time unit). If the arguments for a unit have been transmitted at time t, its output will be produced at time *t + 1*. A recursive computation can be stopped after a certain number of steps and the last computed output taken as the result of the recursive computation.

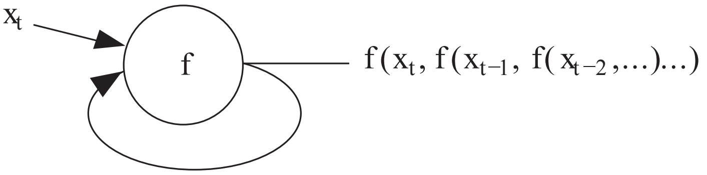


Fig. 4.7 Recursive evaluation

In this chapter we deal first with networks without cycles, in which the time dimension can be disregarded. Then we deal with recurrent networks and their temporal coordination. The first model we consider was proposed in 1943 by Warren McCulloch and Walter Pitts. Inspired by neurobiology they put forward a model of computation oriented towards the computational capabilities of real neurons and studied the question of abstracting universal concepts from specific perceptions.

We will avoid giving a general definition of a neural network at this point. So many models have been proposed which differ in so many respects that any definition trying to encompass this variety would be unnecessarily clumsy. As we show in this chapter, it is not necessary to start building neural networks with "high powered" computing units, as some authors do. We will start our investigations with the general notion that a neural network is a network of functions in which synchronisation can be considered explicitly or not.

4.4.2 The Computing Units

The nodes of the networks we consider will be called computing elements or simply units. We assume that the edges of the network transmit information in a predetermined direction and the number of incoming edges into a node is not restricted by some upper bound. This is called the unlimited fan-in property of our computing units.

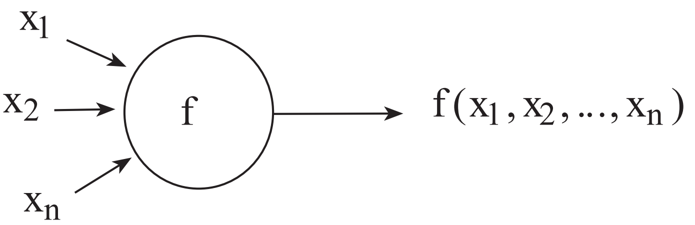


Fig. 4.8 Evaluation of a function of n arguments

The primitive function computed at each node is in general a function of n arguments. Normally, however, we try to use very simple primitive functions of one argument at the nodes. This means that the incoming n arguments have to be reduced to a single numerical value. Therefore computing units are split into two functional parts: an integration function g reduces the n arguments to a single value and the output or activation function f produces the output of this node taking that single value as its argument. Figure below shows this general structure of the computing units. Usually the integration function g is the addition function.

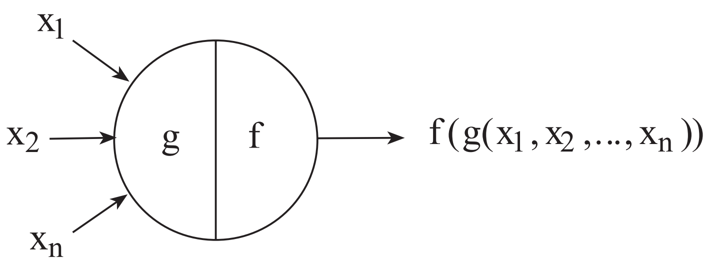


Fig. 4.9 Generic computing unit

McCulloch–Pitts networks are even simpler than this, because they use solely binary signals, that are ones or zeros. The nodes produce only binary results and the edges transmit exclusively ones or zeros. The networks are composed of directed un-weighted edges of excitatory or of inhibitory type. The latter are marked in diagrams using a small circle attached to the end of the edge. Each McCulloch–Pitts unit is also provided with a certain threshold value. At first sight the McCulloch–Pitts model seems very limited, since only binary information can be produced and transmitted, but it already contains all necessary features to implement the more complex models. Figure below shows an abstract McCulloch–Pitts computing unit.

Following Minsky it will be represented as a circle with a black half. Incoming edges arrive at the white half; outgoing edges leave from the black half. Outgoing edges can fan out any number of times.

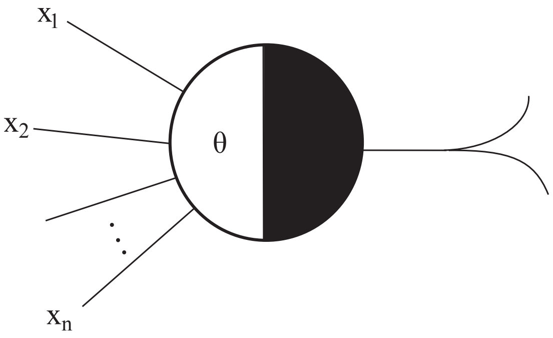


Fig. 4.10 Diagram of a McCulloch–Pitts unit

The rule for evaluating the input to a McCulloch–Pitts unit is the following:

- Assume that a McCulloch–Pitts unit gets an input $x_1, x_2, \ldots, x_n$ through n excitatory edges and an input $y_1, y_2, \ldots, y_m$ through m inhibitory edges.
- If m ≥ 1 and at least one of the signals $y_1, y_2, \ldots, y_m$ is 1, the unit is inhibited and the result of the computation is 0.
- Otherwise the total excitation $x = x_1 + x_2 + \cdots + x_n$ is computed and compared with the threshold θ of the unit (if $n = 0$ then x = 0). If $x \geq \theta$ the unit fires a 1, if $x < \theta$ the result of the computation is 0.

This rule implies that a McCulloch–Pitts unit can be inactivated by a single inhibitory signal, as is the case with some real neurons. When no inhibitory signals are present, the units act as a threshold gate capable of implementing many other logical functions of n arguments. Figure below shows the activation function of a unit, the so-called step function. This function changes discontinuously from zero to one at. When is zero and no inhibitory signals are present, we have the case of a unit producing the constant output one. If is greater than the number of incoming excitatory edges, the unit will never fire.

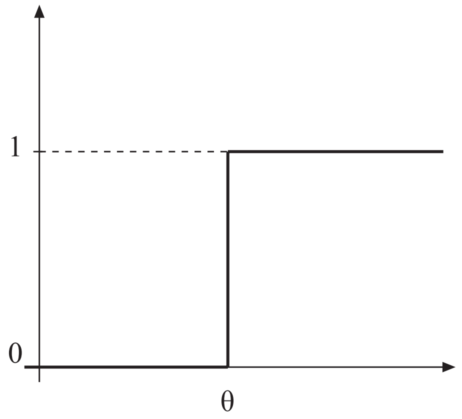


Fig. 4.11 The step function with threshold

In the following subsection we assume provisionally that there is no delay in the computation of the output.

4.5 Synthesis of Boolean Functions

The power of threshold gates of the McCulloch–Pitts type can be illustrated by showing how to synthesise any given logical function of n arguments. We deal firstly with the more simple kind of logic gates.

4.5.1 Conjunction, Disjunction, Negation

Mappings from $\{0, 1\}^n$ onto $\{0, 1\}$ are called logical or Boolean functions. Simple logical functions can be implemented directly with a single McCulloch– Pitts unit. The output value 1 can be associated with the logical value *true* and 0 with the logical value *false*. It is straightforward to verify that the two units of Figure below compute the functions AND and OR respectively.

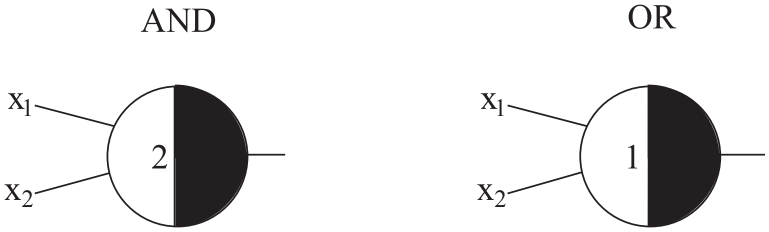


Fig. 4.12 Implementation of AND and OR gates

A single unit can compute the disjunction or the conjunction of n arguments as is shown in Study the figure below where the conjunction of three and four arguments is computed by two units. The same kind of computation requires several conventional logic gates with two inputs. It should be clear from this simple example that threshold logic elements can reduce the complexity of the circuit used to implement a given logical function.

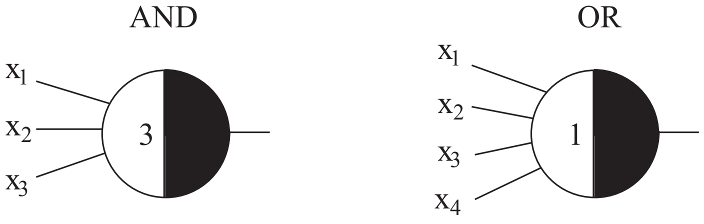


Fig. 4.13 Generalised AND and OR gates

As is well known, AND and OR gates alone cannot be combined to produce all logical functions of n variables. Since uninhibited threshold logic elements are capable of implementing more general functions than conventional AND or OR gates, the question of whether they can be combined to produce all logical functions arises. Stated another way: is inhibition of McCulloch–Pitts units' necessary or can it be dispensed with? The following proposition shows that it is necessary. A monotonic logical function f of n arguments is one whose value at two given n-dimensional points $x = (x_1, \dots, x_n)$ and $y = (y_1, \dots, y_n)$ is such that $f(x) \geq f(y)$ whenever the number of ones in the input y is a subset of the ones in the input *x*. An example of a non-monotonic logical function of one argument is logical negation.

Proposition 1: Uninhibited threshold logic elements of the McCulloch–Pitts type can only implement monotonic logical functions.

Proof: An example shows the kind of argumentation needed. Assume that the input vector (1, 1, . . . , 1) is assigned the function value 0. Since no other vector can set more edges in the network to 1 than this vector does, any other input vector can also only be evaluated to 0. In general, if the ones in the input vector y are a subset of the ones in the input vector *x*, then the first cannot set more edges to 1 than *x* does. This implies that f(x) ≥ f(y), as had to be shown.

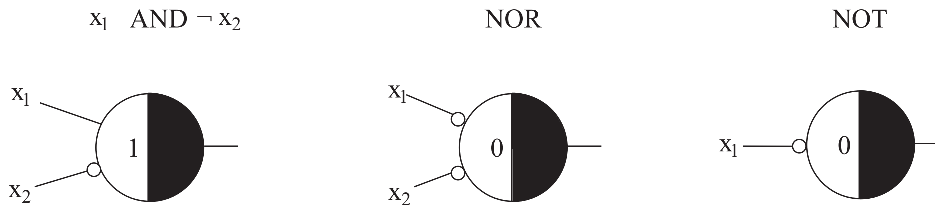


Fig. 4.14 Logical functions and their realisation

The units of above figure show the implementation of some non-monotonic logical functions requiring inhibitory connections. Logical negation, for example, can be computed using a McCulloch–Pitts unit with threshold 0 and an inhibitory edge. The other two functions can be verified by the reader.

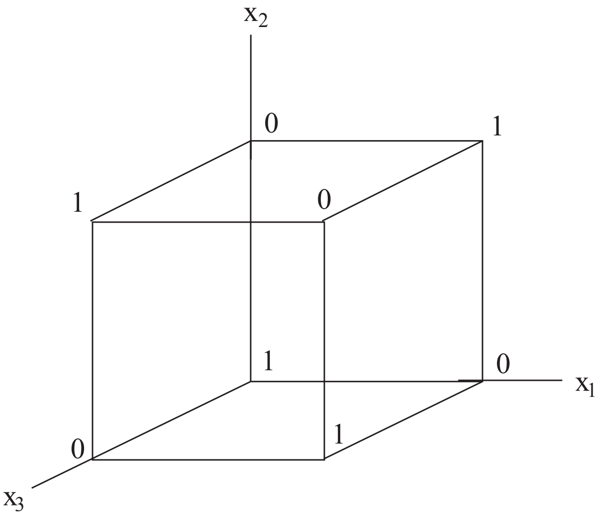


Fig. 4.15 Function values of a logical function of three variables

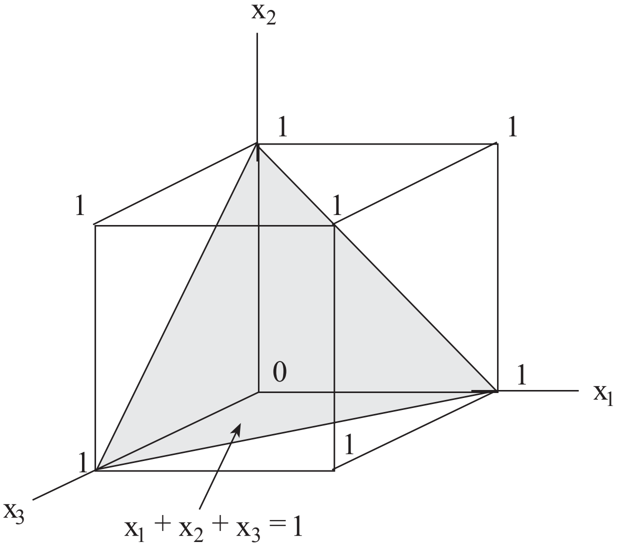


Fig. 4.16 Separation of the input space for the OR function

4.5.2 Geometric Interpretation

It is very instructive to visualise the kind of functions that can be computed with McCulloch–Pitts cells by using a diagram. Figure below shows the eight vertices of a three-dimensional unit cube. Each of the three logical variables x_1, x_2 and x_3 can assume one of two possible binary values. There are eight possible combinations, represented by the vertices of the cube. A logical function is just an assignment of a 0 or a 1 to each of the vertices. The figure shows one of these assignments. In the case of *n* variables, the cube consists of 2^n vertices and admits 2^{2^n} different binary assignments.

McCulloch–Pitts units divide the input space into two half-spaces. For a given input (x_1, x_2, x_3) and a threshold the condition $x_1 + x_2 + x_3 \geq \theta$ is tested, which is true for all points to one side of the plane with the equation $x_1 + x_2 + x_3 = \theta$ and false for all points to the other side (without including the plane itself in this case). Figure above shows this separation for the case in which θ = 1 that is for the OR function. Only those vertices above the separating plane are labelled 1.

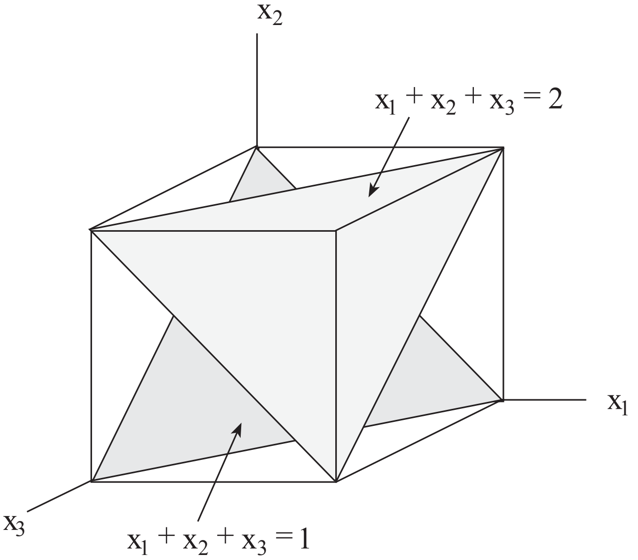


Fig. 4.17 Separating planes of the OR and majority functions

The majority function of three variables divides input space in a similar manner, but the separating plane is given by the equation $x_1 + x_2 + x_3 = 2$. Figure above shows the additional plane. The planes are always parallel in the case of McCulloch–Pitts units. Non-parallel separating planes can only be produced using weighted edges.

4.5.3 Constructive Synthesis

Every logical function of n variables can be written in tabular form. The value of the function is written down for every one of the possible binary combinations of the n inputs. If we want to build a network to compute this function, it should have n inputs and one output. The network must associate each input vector with the correct output value. If the number of computing units is not limited in some way, it is always possible to build or synthesise a network which computes this function. The constructive proof of this proposition profits from the fact that McCulloch–Pitts units can be used as binary decoders.

Consider for example the vector (1, 0, 1). It is the only one which fulfils the condition x_1^¬x_2^x_3. This condition can be tested by a single computing unit. Since only the vector (1, 0, 1) makes this unit fire, the unit is a decoder for this input. Assume that a function *F* of three arguments has been defined according to the following table: To compute this function it is only necessary to decode all those vectors for which the function's value is 1. Synthesis of the function *F* shows a network capable of computing the function F.

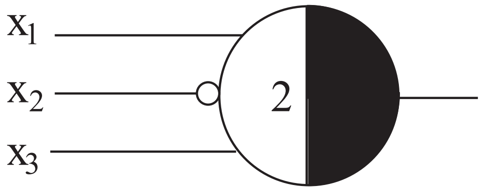


Fig. 4.18 Decoder for the vector (1, 0, 1)

input vectors	F
(0,0,1)	1
(0,1,0)	1
all others	0

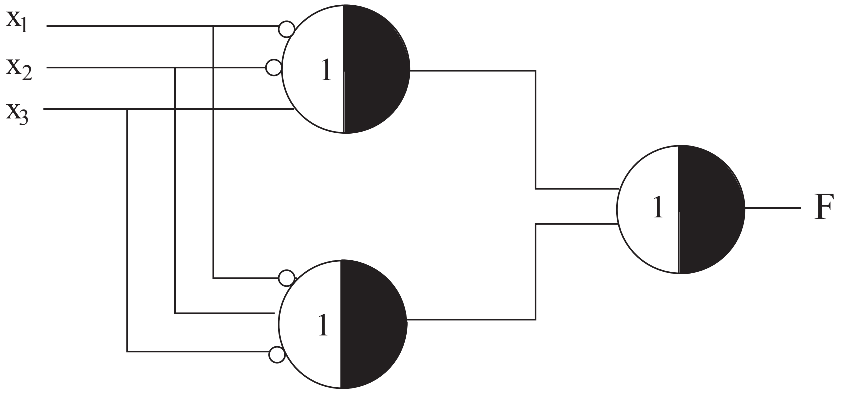


Fig. 4.19 Synthesis of the function F

The individual units in the first layer of the composite network are decoders. For each vector for which F is 1 a decoder is used. In our case we need just two decoders. Components of each vector which must be 0 are transmitted with inhibitory edges, components which must be 1 with excitatory ones. The threshold of each unit is equal to the number of bits equal to 1 that must be present in the desired input vector. The last unit to the right is a disjunction: if any one of the specified vectors can be decoded this unit fires a 1.

It is straightforward to extend this constructive method to other Boolean functions of any other dimension. This leads to the following proposition:

Proposition 2: Any logical function *F: {0, 1}^{n}* → *{0, 1}* can be computed with a McCulloch–Pitts network of two layers.

No attempt has been made here to minimise the number of computing units. In fact, we need as many decoders as there are ones in the table of function values. We can also consider the minimal possible set of building blocks needed to implement arbitrary logical functions when the fan-in of the units is bounded in some way. The circuits of above two figures use decoders of n inputs. These decoders can be built of simpler cells, for example, two units capable of respectively implementing the AND function and negation. Inhibitory connections in the decoders can be replaced with a negation gate. The output of the decoders is collected at a conjunctive unit. The decoder of vector (1, 0, 1) can be implemented as shown in figure below.

The only difference from the previous decoder are the negated inputs and the higher threshold in the AND unit. All decoders for a row of the table of a logical function can be designed in a similar way. This immediately leads to the following proposition:

Proposition 3: All logical functions can be implemented with a network composed of units which exclusively compute the AND, OR, and NOT functions. The three units AND, NOT and OR are called a logical basis because of this property. Since OR can be implemented using AND and NOT units, these two alone constitute a logical basis. The same happens with OR and NOT units. John von Neumann showed that through a redundant coding of the inputs (each variable is transmitted through two lines) AND and OR units alone can constitute a logical basis.

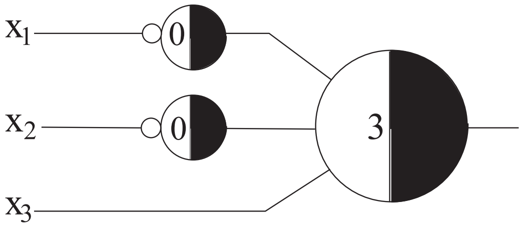


Fig. 4.20 A composite decoder for the vector (0, 0, 1)

4.6 Equivalent Networks

We can build simpler circuits by using units with more general properties, for example weighted edges and relative inhibition. However, as we show in this section, circuits of McCulloch–Pitts units can emulate circuits built out of high-powered units by exploiting the trade-off between the complexities of the network versus the complexity of the computing units.

4.6.1 Weighted and Un-weighted Networks

Since McCulloch–Pitts networks do not use weighted edges the question of whether weighted networks are more general than un-weighted ones must be answered. A simple example shows that both kinds of networks are equivalent. Assume that three weighted edges converge on the unit shown below.

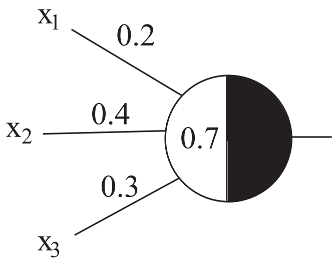


Fig. 4.21 Weighted unit

The unit computes

$$0.2x_1 + 0.4x_2 + 0.3_{x3} \geq 0.7$$

But this is equivalent to

$$2x_1 + 4x_2 + 3_{x3} \geq 7$$

and this computation can be performed with the network of figure below.

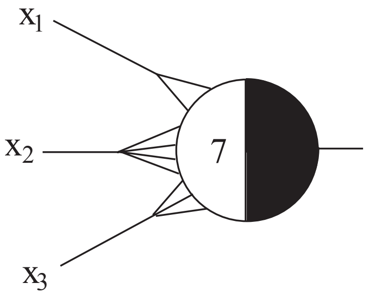


Fig. 4.22 Equivalent computing unit

The figure shows that positive rational weights can be simulated by simply fanning-out the edges of the network the required number of times. This means that we can either use weighted edges or go for a more complex topology of the network, with many redundant edges. The same can be done in the case of irrational weights if the number of input vectors is finite.

4.6.2 Absolute and Relative Inhibition

In the last subsection we dealt only with the case of positive weights. Two classes of inhibition can be identified: absolute inhibition corresponds to the one used in McCulloch–Pitts units. Relative inhibition corresponds to the case of edges weighted with a negative factor and whose effect is to lower the firing threshold when a 1 is transmitted through this edge.

Proposition 4: Networks of McCulloch–Pitts units are equivalent to net- works with relative inhibition.

Proof: It is only necessary to show that each unit in a network where relative inhibition is used is equivalent to one or more units in a network where absolute inhibition is used. It is clear that it is possible to implement absolute inhibition with relative inhibitory edges. If the threshold of a unit is the integer m and if n excitatory edges impinge on it then the maximum possible total excitation for this unit are $n - m$. If m ≥ n the unit never fires and the inhibitory edge is irrelevant. It suffices to fan out the inhibitory edge $n{-}m{+}1$ times and make all these edges meet at the unit. When a 1 is transmitted through the inhibitory edges the total amount of inhibition is $n - m + 1$ and this shuts down the unit. To prove that relative inhibitory edges can be simulated with absolute inhibitory ones, study figure below. The network to the left contains a relative inhibitory edge, the network to the right absolute inhibitory ones. The reader can verify that the two networks are equivalent.

Relative inhibitory edges correspond to edges weighted with −1. We can also accept any other negative weight *w*. In that case the threshold of the unit to the right of figure below should be *m+w* instead of *m+1*. Therefore networks with negative weights can be simulated using un-weighted McCulloch–Pitts elements.

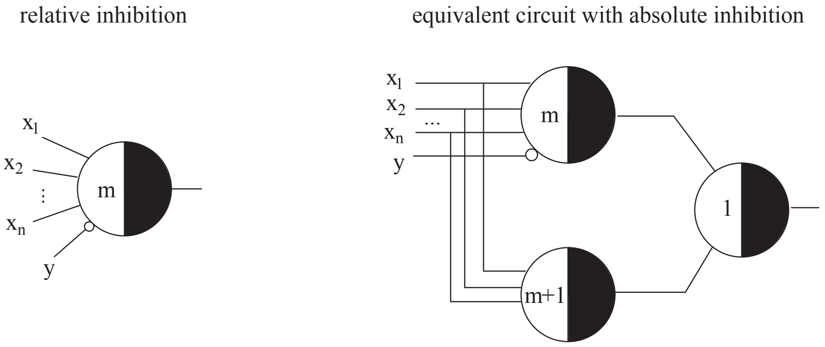


Fig. 4.23 Two equivalent networks

As shown above, we can implement any kind of logical function using un-weighted networks. What we trade is the simplicity of the building blocks for a more convoluted topology of the network. Later we will always use weighted networks in order to simplify the topology.

4.6.3 Binary Signals and Pulse Coding

An additional question which can be raised is whether binary signals are not a very limited coding strategy. Are networks in which the communication channels adopt any of ten or fifteen different states more efficient than channels which adopt only two states, as in McCulloch–Pitts networks? To give an answer we must consider that unit states have a price, in biological networks as well as in artificial ones. The transmitted information must be optimised using the number of available switching states.

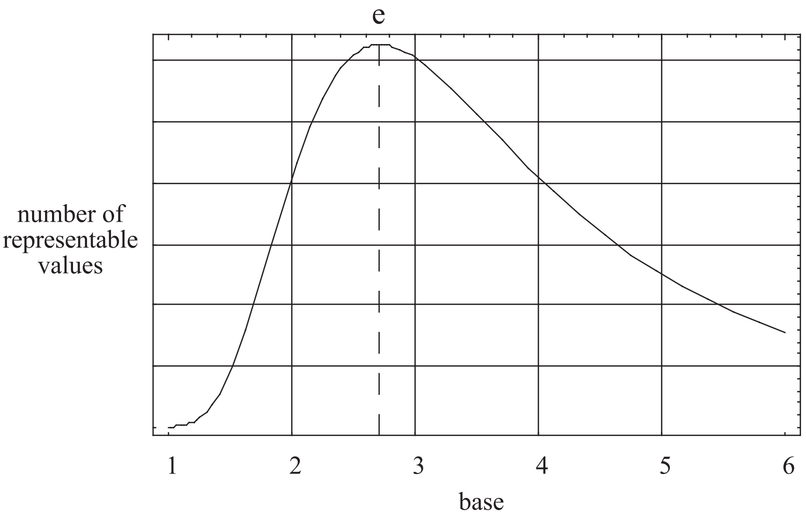


Fig. 4.24 Number of representable values as a function of the base

Assume that the number of states per communication channel is *b* and that *c* channels are used to input information. The cost *K* of the implementation is proportional to both quantities that is $K = \gamma bc$, where γ is a proportionality constant. Using *c* channels with *b* states, b^c different numbers can be represented.

This means that $c = K/\gamma b$ and, if we set $k = K/\gamma$, we are seeking the numerical base *b* which optimises the function bk/b. Since we assume constant cost, *k* is a constant. Above figure shows that the optimal value for *b* is the Euler constant *e*. Since the number of channel states must be an integer, three states would provide a good approximation to the optimal coding strategy.

However, in electronic and biological systems decoding of the signal plays such an important role that the choice of two states per channel becomes a better alternative. Wiener arrived at a similar conclusion through a somewhat different argument. The binary nature of information transmission in the nervous system seems to be an efficient way to transport signals. However, in the next chapters we will assume that the communication channels can transport arbitrary real numbers. This makes the analysis simpler than when we have to deal explicitly with frequency modulated signals, but does not lead to a minimisation of the resources needed for a technical implementation. Some researchers prefer to work with so-called weightless networks which operate exclusively with binary data.

4.7 Recurrent Networks

We have already shown that feed-forward networks can implement arbitrary logical functions. In this case the dimension of the input and output data is predetermined. In many cases, though, we want to perform computations on an input of variable length, for example, when adding two binary numbers being fed bit for bit into a network, which in turn produces the bits of the result one after the other.

A feed-forward network cannot solve this problem because it is not capable of keeping track of previous results and, in the case of addition, the carry bit must be stored in order to be reused. This kind of problem can be solved using recurrent networks that are networks whose partial computations are recycled through the network itself. Cycles in the topology of the network make storage and reuse of signals possible for a certain amount of time after they are produced.

4.7.1 Stored State Networks

McCulloch–Pitts units can be used in recurrent networks by introducing a temporal factor in the computation. We will assume that computation of the activation of each unit consumes a time unit. If the input arrives at time t the result is produced at time $t + 1$. Up to now, we have been working with units which produce results without delay. The numerical capabilities of any feed-forward network with instantaneous computation at the nodes can be reproduced by networks of units with delay. We only have to take care to coordinate the arrival of the input values at the nodes. This could make the introduction of additional computing elements necessary, whose sole mission is to insert the necessary delays for the coordinated arrival of information. This is the same problem that any computer with clocked elements has to deal with.

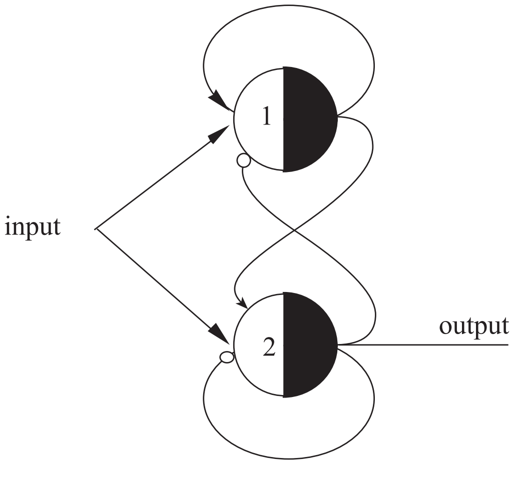


Fig. 4.25 Network for a binary scaler

The above figure shows a simple example of a recurrent circuit. The network processes a sequence of bits, giving off one bit of output for every bit of input, but in such a way that any two consecutive ones are transformed into the sequence 10. The binary sequence 00110110 is transformed for example into the sequence 00100100. The network recognises only two consecutive ones separated by at least a zero from a similar block.

4.7.2 Finite Automata

The network discussed in the previous subsection is an example of an automaton. This is an abstract device capable of assuming different states which change according to the received input. The automaton also produces an output according to its momentary state. In the previous example, the state of the automaton is the specific combination of signals circulating in the network at any given time. The set of possible states corresponds to the set of all possible combinations of signals travelling through the network.

state transitions

input \ state	Q_0	Q_1
0	Q_0	Q_0
1	Q_1	Q_1

output table

input \ state	Q_0	Q_1
0	0	1
1	0	1

Fig. 4.26 State tables for a binary delay

Finite automata can take only a finite set of possible states and can react only to a finite set of input signals. The state transitions and the output of an automaton can be specified with a table, like the one shown in figure above. This table defines an automaton which accepts a binary signal at time t and produces an output at time *t+1*. The automaton has two states, Q_0 and Q_1, and accepts only the values 0 or 1. The first table shows the state transitions, corresponding to each input and each state. The second table shows the output values corresponding to the given state and input. From the table we can see that the automaton switches from state Q_0 to state Q_1 after accepting the input 1. If the input bit is a 0, the automaton remains in state Q_0. If the state of the automaton is Q_1 the output at time *t + 1* is 1 regardless of whether 1 or 0 was given as input at time t. All other possibilities are covered by the rest of the entries in the two tables.

The diagram below shows how the automaton works. The values at the beginning of the arrows represent an input bit for the automaton. The values in the middle of the arrows are the output bits produced after each new input. An input of 1, for example, produces the transition from state Q_0 to state Q_1 and the output 0. The input 0 produces a transition to state Q_0. The automaton is thus one that stores only the last bit of input in its current state.

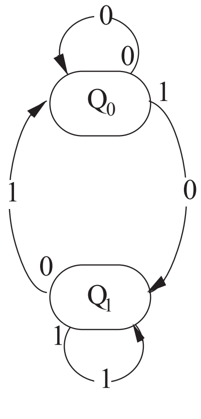


Fig. 4.27 Diagram of a finite automaton

Finite automata without input from the outside, i.e., free-wheeling automata, unavoidably fall in an infinite loop or reach a final constant state. This is why finite automata cannot cover all computable functions, for whose computation an infinite number of states are needed. A Turing machine achieves this through an infinite storage band which provides enough space for all computations. Even a simple problem like the multiplication of two arbitrary binary numbers presented sequentially cannot be solved by a finite automaton. Although our computers are finite automata, the number of possible states is so large that we consider them as universal computing devices for all practical purposes.

4.7.3 Finite Automata and Recurrent Networks

We now show that finite automata and recurrent networks of McCulloch–Pitts units are equivalent. We use a variation of a constructive proof due to Minsky.

Proposition 5: Any finite automaton can be simulated with a network of McCulloch–Pitts units.

Proof: Figure below is a diagram of the network needed for the proof. Assume that the input signals are transmitted through the input lines I_1 to I_m and at each moment t only one of these lines is conducting a 1. All other input lines are passive (set to 0). Assume that the network starts in a well-defined state Q_i.

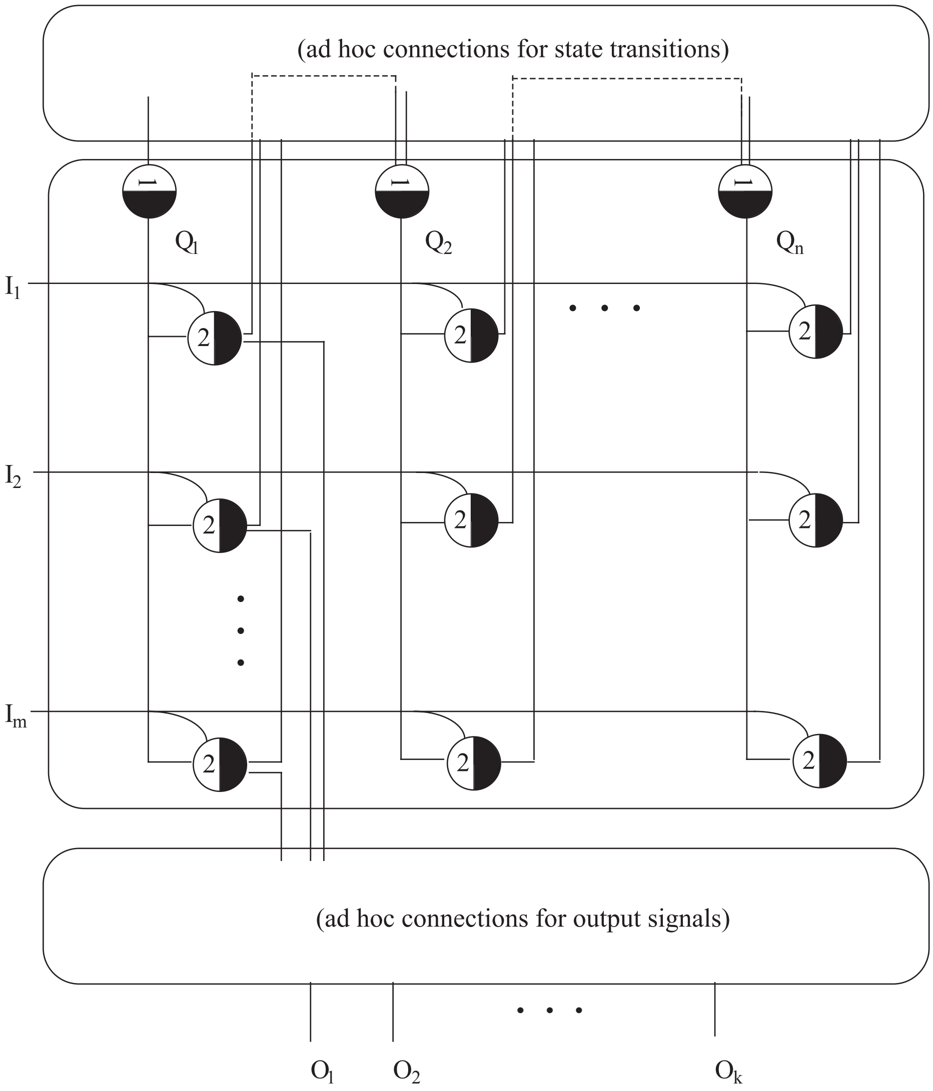

(ad hoc connections for state transitions)
1
1
1
Q1
Q2
Qn
I1
I2
Im
2
2
2
2
2
2
2
2
2
. . .
(ad hoc connections for output signals)
. . .
O1
O2
Ok

Fig. 4.28 Implementation of a finite automaton with McCulloch–Pitts units

This means that one, and only one, of the lines labelled $Q_1, \ldots, Q_n$ is set to 1 and the others to 0. At time *t + 1* only one of the AND units can produce a 1, namely the one in which both input and state line are set to 1. The state transition is controlled by the ad hoc connections defined by the user in the upper box. If, for example, the input I_1 and the state Q_1 at time t produce the transition to state Q_2 at time *t+ 1*, then we have to connect the output of the upper left AND unit to the input of the OR unit with the output line named Q_2 (dotted line in the diagram). This output will become active at time *t + 2*. At this stage a new input line must be set to 1 (for example I_2) and a new state transition will be computed (*Qn* in our example).

The connections required to produce the desired output are defined in a similar way. This can be controlled by connecting the output of each AND unit to the corresponding output line $O_1, \ldots, O_k$ using a box of ad hoc connections similar to the one already described. A disadvantage of this constructive method is that each simulated finite automaton requires a special set of connections in the upper and lower boxes. It is better to define a universal network capable of simulating any other finite automaton without having to change the topology of the network (under the assumption of an upper bound for the size of the simulated automata). This is indeed an active field of research in which networks learn to simulate automata. The necessary network parameters are found by a learning algorithm. In the case of McCulloch–Pitts units the available degrees of freedom are given by the topology of the network.

4.7.4 A First Classification of Neural Networks

The first clear separation line runs between weighted and un-weighted networks. It has already been shown that both classes of models are equivalent. The main difference is the kind of learning algorithm that can be used. In un-weighted networks only the thresholds and the connectivity can be adapted. In weighted networks the topology is not usually modified during learning (although we will see some algorithms capable of doing this) and only an optimal combination of weights is sought.

The second clear separation is between synchronous and asynchronous models. In synchronous models the output of all elements is computed instantaneously. This is always possible if the topology of the network does not contain cycles. In some cases the models contain layers of computing units and the activity of the units in each layer is computed one after the other, but in each layer simultaneously. Asynchronous models compute the activity of each unit independently of all others and at different stochastically selected times (as in Hopfield networks). In these kinds of models, cycles in the underlying connection graph pose no particular problem.

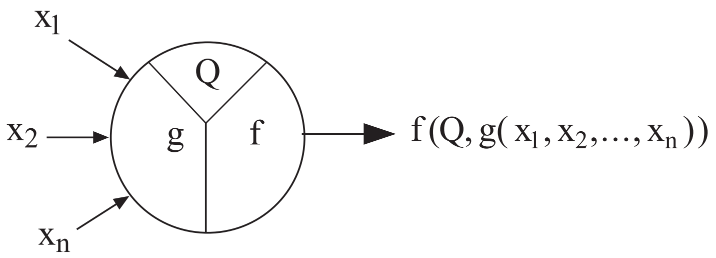


Fig. 4.29 A unit with stored state Q

Finally, we can distinguish between models with or without stored unit states. In above figure we gave an example of a unit without stored state. It shows a unit in which a state *Q* is stored after each computation. The state *Q* can modify the output of the unit in the following activation. If the number of states and possible inputs is finite, we are dealing with a finite automaton. Since any finite automaton can be simulated by a network of computing elements without memory, these units with a stored state can be substituted by a network of McCulloch–Pitts units. Networks with stored state units are thus equivalent to networks without stored-state units. Data is stored in the network itself and in its pattern of recursion.

It can be also shown that time varying weights and thresholds can be implemented in a network of McCulloch–Pitts units using cycles, so that networks with time varying weights and thresholds are equivalent to networks with constant parameters, whenever recursion is allowed.

4.8 Harmonic Analysis of Logical Functions

An interesting alternative for the automatic synthesis of logic functions and for a quantification of their implementation complexity is to do an analysis of the distribution of its non-zero values using the tools of harmonic analysis. Since we can tabulate the values of a logical function in a sequence, we can think of it as a one-dimensional function whose fundamental “frequencies” can be extracted using the appropriate mathematical tools. We will first deal with this problem in a totally general setting and then show that the Hadamard–Walsh transform is the tool we are looking for.

4.8.1 General Expression

Assume that we are interested in expressing a function f: IR^m → IR as a linear combination of n functions $f_1, f_2, . . . , f_n$ using the n constants $a_1, a_2, . . . , a_n$ in the following form

$$f = a_1f_1 + a_2f_2 + \cdot\cdot\cdot + a_nf_n$$

The domain and range of definition are the same for f and the base functions. We can determine the quadratic error E of the approximation in the whole domain of definition V for given constants $a_1, . . . , a_n$ by computing

$$E = \int_v (f - (a_1f_1 + a_2f_2 + \ldots + a_nf_n))^2\, dV$$

Here we are assuming that f and the functions f_i, $i = 1, . . . , n$, are integrable in its domain of definition V . Since we want to minimise the quadratic error E we compute the partial derivatives of E with respect to $a_1, a_2, ..., a_n$ and set them to zero:

$$\frac{dE}{da_i} = -2 \int_v f_i\ (f - a_1f_1 - a_2f_2 - \cdots - a_nf_n))^2\, dV = 0, \text{ for } i = 1, ... , n$$

This leads to the following set of n equations expressed in a simplified notation:

$$a_1 \int f_if_1 + a_2 \int f_i\, f_2 + \cdots + a_n \int f_if_n = \int f_if, \text{ for } i = 1, ..., n$$

Expressed in matrix form the set of equations becomes:

$$\begin{pmatrix} \int f_1f_1 & \int f_1f_2 & \cdots & \int f_1f_n \\ \int f_2f_1 & \int f_2f_2 & & \int f_2f_n \\ \vdots & & \ddots & \vdots \\ \int f_nf_1 & \int f_nf_2 & \cdots & \int f_nf_n \end{pmatrix} \begin{pmatrix} a_1 \\ a_2 \\ \vdots \\ a_n \end{pmatrix} = \begin{pmatrix} \int f_1f \\ \int f_2f \\ \vdots \\ \int f_nf \end{pmatrix}$$

This expression is very general. The only assumption we have used is the integrability of the partial products of the form f_if_j and f_if. Since no special assumptions on the integral were used, it is also possible to use a discrete version of this equation. Assume that the function f has been defined at m points and let the symbol $\sum f_if_j$ stand for $\sum_{k=1}^{m} f_i\ (x_k)f_j(x_k)$. In this case the above expression transforms to

$$\begin{pmatrix} \sum f_1f_1 & \sum f_1f_2 & \cdots & \sum f_1f_n \\ \sum f_2f_1 & \sum f_2f_2 & & \sum f_2f_n \\ \vdots & & \ddots & \vdots \\ \sum f_nf_1 & \sum f_nf_2 & \cdots & \sum f_nf_n \end{pmatrix} = \begin{pmatrix} a_1 \\ a_2 \\ \vdots \\ a_n \end{pmatrix} = \begin{pmatrix} \sum f_1f \\ \sum f_2f \\ \vdots \\ \sum f_nf \end{pmatrix}$$

The general formula for the polynomial approximation of m data points $(x_1, y_1), . . . , (x_m, y_m)$ using the primitive functions $x^0, x^1, x^2, . . . , x^{n-1}$ translates directly into

$$\begin{pmatrix} m & \sum x_i & \cdots & \sum x_i^{n-1} \\ \sum x_i & \sum x_i^2 & \ddots & \sum x_i^n \\ \vdots & & & \vdots \\ \sum x_i^{n-1} & \sum x_i^n & \cdots & \sum x_i^{2n-2} \end{pmatrix} \begin{pmatrix} a_1 \\ a_2 \\ \vdots \\ a_n \end{pmatrix} = \begin{pmatrix} \sum y_i \\ \sum x_iy_i \\ \vdots \\ \sum x_i^{n-1}\, y_i \end{pmatrix}$$

In the case of base functions that are mutually orthogonal, the integrals $\int f_kf_j$ vanish when $k \neq j$. In this case the n × n matrix is diagonal and the above equation becomes very simple. Assume, as in the case of the Fourier transform, that the functions are sines and cosines of the form $\sin(k_ix)$ and $\cos(k_jx)$. Assume that no two sine functions have the same integer wave number k_i and no two cosine functions the same integer wave number k_j. In this case the integral $\int_0^{2\pi} \sin\,(k_ix)\sin\,(k_jx)$ is equal to π, whenever $i = j$, otherwise it vanishes. The same is true for the cosine functions. The expression transforms then to

$$\pi \begin{pmatrix} a_1 \\ a_2 \\ \vdots \\ a_n \end{pmatrix} = \begin{pmatrix} \int f_1 f \\ \int f_2 f \\ \vdots \\ \int f_n f \end{pmatrix}$$

which is just another way of computing the coefficients for the Fourier approximation of the function f.

4.8.2 The Hadamard–Walsh Transform

In our case we are interested in expressing Boolean functions in terms of a set of primitive functions. We adopt bipolar coding, so that now the logical value false is represented by −1 and the logical value true by 1. In the case of n logical variables $x_1, \ldots, x_n$ and the logical functions defined on them, we can use the following set of 2n primitive functions:

- The constant function $(x_1, \ldots, x_n) \to 1$
- The $\binom{n}{k}$ monomials $(x_1, \ldots, x_n) \to xl_1 xl_2 \cdots xl_k$, where $k = 1, \ldots, n$ and $l_1, l_2, \ldots, l_k$ is a set of k different indices in $\{1, 2, \ldots, n\}$

All these functions are mutually orthogonal. In the case of two input variables the transformation becomes:

$$4 \begin{pmatrix} a_1 \\ a_2 \\ a_3 \\ a_n \end{pmatrix} = \begin{pmatrix} 1 & 1 & 1 & 1 \\ -1 & 1 & -1 & 1 \\ -1 & -1 & 1 & 1 \\ 1 & -1 & -1 & 1 \end{pmatrix} \begin{pmatrix} f1 \\ f2 \\ f3 \\ f4 \end{pmatrix}$$

In the general case we compute 2^n coefficients using the formula

$$\begin{pmatrix} a_1 \\ a_2 \\ \vdots \\ a_{2^n} \end{pmatrix} \begin{pmatrix} a_1 \\ a_2 \\ \vdots \\ a_{2^n} \end{pmatrix} = H_n \begin{pmatrix} f1 \\ f2 \\ \vdots \\ f2^n \end{pmatrix} \begin{pmatrix} f1 \\ f2 \\ \vdots \\ f2^n \end{pmatrix}$$

where the matrix *Hn* is defined recursively as

$$H_n = \begin{pmatrix} H_{n-1} & H_{n-1} \\ -H_{n-1} & -H_{n-1} \end{pmatrix}$$

Where by

$$H_1 = \frac{1}{2} \begin{pmatrix} 1 & 1 \\ -1 & 1 \end{pmatrix}$$

The AND function can be expressed using this simple prescription as

$$x_1 \wedge x_2 = \frac{1}{4} (-2 + 2x_1 + 2x_2 + 2x_1 x_2)$$

The coefficients are the result of the following computation:

$$\begin{pmatrix} a_1 \\ a_2 \\ a_3 \\ a_4 \end{pmatrix} = \frac{1}{4} \begin{pmatrix} 1 & 1 & 1 & 1 \\ -1 & 1 & -1 & 1 \\ -1 & -1 & 1 & 1 \\ 1 & -1 & -1 & 1 \end{pmatrix} \begin{pmatrix} f1 \\ f2 \\ f3 \\ f4 \end{pmatrix}$$

The expressions obtained for the logical functions can be wired as networks using weighted edges and only two operations, addition and binary multiplication. The Hadamard–Walsh transform is consequently a method for the synthesis of Boolean functions. The next step, that is, the optimisation of the number of components, demands additional techniques which have been extensively studied in the field of combinatorics.

4.8.3 Applications of Threshold Logic

Threshold units can be used in any application in which we want to reduce the execution time of a logic operation to possibly just two layers of computational delay without employing a huge number of computing elements. It has been shown that the parity and majority functions, for example, cannot be implemented in a fixed number of layers of computation without using an exponentially growing number of conventional logic gates, even when unbounded fan-in is used. The majority function k out of n is a threshold function implementable with just a single McCulloch–Pitts unit. Although circuits built from n threshold units can be built using a polynomial number *P(n)* of conventional gates the main difference is that conventional circuits cannot guarantee a constant delay.

With threshold elements we can build multiplication or division circuits that guarantee a constant delay for 32 or 64-bit operands. Any symmetric Boolean function of *n* bits can in fact be built from two layers of computing units using *n+ 1 gate*. Some authors have developed circuits of threshold networks for fast multiplication and division, which are capable of operating with constant delay for a variable number of data bits. Threshold logic offers thus the possibility of harnessing parallelism at the level of the basic arithmetic operations.

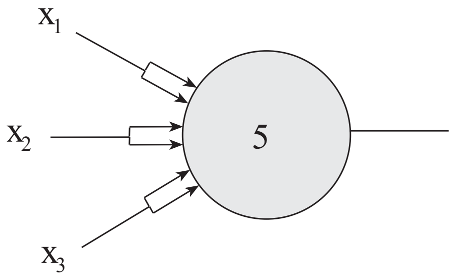


Fig. 4.30 Fault-tolerant gate

Threshold logic also offers a simpler way to achieve fault-tolerance. Figure above shows an example of a unit that can be used to compute the conjunction of three inputs with inherent fault tolerance. Assume that three inputs x_1, x_2, x_3 can be transmitted, each with probability p of error. The probability of a false result when x_1, x_2 and x_3 are equal, and we are computing the conjunction of the three inputs, is *3p*, since we assume that all three values are transmitted independently of each other. But assume that we transmit each value using two independent lines.

The gate of above figure has a threshold of 5, that is, it will produce the correct result even in the case where an input value is transmitted with an error. The probability that exactly two ones arrive as zeros is p^2 and, since there are 15 combinations of two out of six lines, the probability of getting the wrong answer is $15p^2$ in this case. If p is small enough then $15p^2 < 3p$ and the performance of the gate is improved for this combination of input values. Other combinations can be analyzed in a similar way. If threshold units are more reliable than the communication channels, redundancy can be exploited to increase the reliability of any computing system.

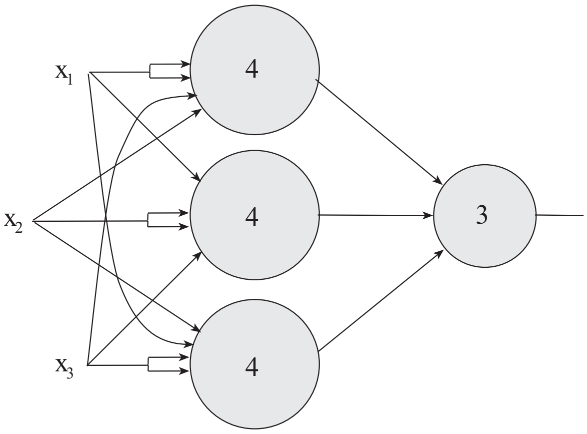


Fig. 4.31 A fault-tolerant AND built of noisy components

When the computing units are unreliable, fault tolerance is achieved using redundant networks. Figure above is an example of a network built using four units. Assume that the first three units connected directly to the three bits of input x_1, x_2, x_3 all fire with probability 1 when the total excitation is greater than or equal to the threshold but also with probability p when it is − 1.

The duplicated connections add redundancy to the transmitted bit, but in such a way that all three units fire with probability one when the three bits are 1. Each unit also fires with probability p if two out of three inputs are 1. However each unit reacts to a different combination. The last unit, finally, is also noisy and fires any time the three units in the first level fire and also with probability p when two of them fire. Since, in the first level, at most one unit fires when just two inputs are set to 1, the third unit will only fire when all three inputs are 1. This makes the logical circuit, the AND function of three inputs, built out of unreliable components error-proof. The network can be simplified using the approach illustrated in figure Fault-tolerant gate.

Summary

- Nerve cells are very complex self-organising systems which have evolved in the course of millions of years.
- In the case of artificial neural networks, the primitive functions are located in the nodes of the network and the composition rules are contained implicitly in the interconnection pattern of the nodes.
- The transmitted information is integrated at the neuron (usually just by adding the different signals) and the primitive function is then evaluated.
- Different models of artificial neural networks differ mainly in the assumptions about the primitive functions used, the interconnection pattern, and the timing of the transmission of information.
- An old problem in approximation theory is to reproduce a given function F: IR→IR either exactly or approximately by evaluating a given set of primitive functions.
- A classical example is the approximation of one-dimensional functions using polynomials or Fourier series.
- The main difference between Taylor or Fourier series and artificial neural networks is, however, that the function *F* to be approximated is given not explicitly but implicitly through a set of input-output examples.
- Ram´on y Cajal perfected Golgi's staining method and published the best diagrams of neurons of his time.
- The word neuron (Greek for nerve) was proposed by the Berlin Professor Wilhelm Waldeger after he saw the preparations of Ram´on y Cajal.
- The Hodgkin–Huxley model was in some ways one of the first artificial neural models, because the postulated dynamics of the nerve impulses could be simulated with simple electric networks.
- The mathematical properties of artificial neural networks were being studied by researchers like Warren McCulloch, Walter Pitts, and John von Neumann.
- The characteristics and structure of biological neural networks provides us with the initial motivation for a deeper inquiry into the properties of networks of abstract neurons.
- Uninhibited threshold logic elements of the McCulloch–Pitts type can only implement monotonic logical functions.
- Every logical function of n variables can be written in tabular form.
- Threshold units can be used in any application in which we want to reduce the execution time of a logic operation to possibly just two layers of computational delay without employing a huge number of computing elements.
- An interesting alternative for the automatic synthesis of logic functions and for a quantification of their implementation complexity is to do an analysis of the distribution of its non-zero values using the tools of harmonic analysis.

References

- Yegnanarayana, B., 2004. *Artificial Neural Networks*, PHI Learning Pvt. Ltd.
- Karayiannis, B. N. &Venetsanopoulos, N. A., 1992. *Artificial Neural Networks: Learning Algorithms, Performance Evaluation, and Applications*, Springer.
- Abraham, A., *Artificial Neural Networks*, [Online] Available at: <http://www.softcomputing.net/ann_chapter.pdf> [Accessed 18 July 2012].
- Cheung, V., Cannons, K., *An Introduction to Neural Networks*, [Online] Available at: <http://www2.econ.iastate.edu/tesfatsi/NeuralNetworks.CheungCannonNotes.pdf> [Accessed 18 July 2012].
- GoogleTechTalks, 4th Dec 2007. *The Next Generation of Neural Networks*, [Video Online] Available at: <http://www.youtube.com/watch?v=AyzOUbkUf3M&feature=results_main&playnext=1&list=PLA47195C498F5DD59> [Accessed 18 July 2012].
- Prof. Hamprecht, F., 29th Sep 2011. *Lecture 05, part 1 | Pattern Recognition*, [Video Online] Available at: <http://www.youtube.com/watch?v=ZeiJmSHx6qI&feature=endscreen&NR=1> [Accessed 18 July 2012].

Recommended Reading

- Priddy, L. K. & Keller, E. P., 2005. *Artificial Neural Networks: An Introduction*, SPIE Press.
- Schalkoff, *Artificial Neural Networks*, Tata McGraw-Hill Education.
- Mehrotra, K., Mohan, K. C. &Ranka, S., 1997. *Elements of Artificial Neural Networks*, 2nd ed. MIT Press.

Self Assessment

1. The chemical transmission of information at the synapses was studied from _____ to_______.
   a. 1920 to 1940
   b. 1921 to 1946
   c. 1930 to 1940
   d. 1931 to 1946

2. _________ and ________ explained the mechanism by which depolarisation waves are produced in the cell membrane.
   a. John von Neumann, Huxley
   b. Hodgkin, Huxley
   c. Ram, Cajal
   d. McCulloch, Walter Pitts

3. A neural network is capable of modeling a function _________.
   a. F: IR → IR
   b. F: $I^{n} \rightarrow R^{m}$
   c. $IR^{n} \rightarrow IR^{m}$
   d. F: $IR^{n} \rightarrow IR^{m}$

4. ___________ networks use solely binary signals that are ones or zeros.
   a. John von Neumann
   b. Huxley
   c. McCulloch–Pitts
   d. Hodgkin

5. John von Neumann showed that through a redundant coding of the inputs ______ and ______ units alone can constitute a logical basis.
   a. NAND, OR
   b. AND, OR
   c. XOR, AND
   d. NAND, NOR

6. Networks of ___________ units are equivalent to net- works with relative inhibition.
   a. Huxley
   b. Hodgkin
   c. John von Neumann
   d. McCulloch–Pitts

7. If the threshold of a unit is the integer m and if n excitatory edges impinge on it then the maximum possible total excitation for this unit are ___________.
   a. $n + m$
   b. n / m
   c. $n – m$
   d. $n * m$

8. The binary nature of information transmission in the nervous system seems to be an efficient way to transport __________.
   a. signals
   b. blood
   c. information
   d. data

9. Which of the following network is not capable of keeping track of previous results?
   a. Recurrent
   b. Feed-forward
   c. Neural
   d. Topology

10. Cycles in the topology of the network make _________ and ________ of signals possible for a certain amount of time after they are produced.
    a. storage, reuse
    b. update, delete
    c. temporary storage, temporary reuse
    d. divert, reuse

Chapter V

Weighted Networks

Aim

The aim of this chapter is to:

- introduce weighted networks
- elucidate computational limits of the perceptron model
- explain pyramidal architecture for image processing

Objectives

The objectives of this chapter are to:

- explain pyramidal networks and the neocognitron
- explicate weighted threshold elements
- elucidate implementation of logical functions

Learning outcome

At the end of this chapter, you will be able to:

- understand the structure of the retina
- create general decision curves
- describe the error function in weight space

5.1 Introduction

In the previous chapter we arrived at the conclusion that McCulloch–Pitts units can be used to build networks capable of computing any logical function and of simulating any finite automaton. From the biological point of view, however, the types of network that can be built are not very relevant. The computing units are too similar to conventional logic gates and the network must be completely specified before it can be used. There are no free parameters which could be adjusted to suit different problems.

Learning can only be implemented by modifying the connection pattern of the network and the thresholds of the units, but this is necessarily more complex than just adjusting numerical parameters. For that reason, we turn our attention to weighted networks and consider their most relevant properties. In the last section of this chapter we show that simple weighted networks can provide a computational model for regular neuronal structures in the nervous system.

5.2 Perceptrons as Weighted Threshold Elements

In 1958 Frank Rosenblatt, an American psychologist, proposed the perceptron, a more general computational model than McCulloch–Pitts units. The essential innovation was the introduction of numerical weights and a special interconnection pattern. In the original Rosenblatt model the computing units are threshold elements and the connectivity is determined stochastically. Learning takes place by adapting the weights of the network with a numerical algorithm. Rosenblatt's model was refined and perfected in the 1960s and its computational properties were carefully analyzed by Minsky and Papert. In the following, Rosenblatt's model will be called the classical perceptron and the model analyzed by Minsky and Papert the perceptron.

The classical perceptron is in fact a whole network for the solution of certain pattern recognition problems. In figure below a projection surface called the retina transmits binary values to a layer of computing units in the projection area. The connections from the retina to the projection units are deterministic and non-adaptive. The connections to the second layer of computing elements and from the second to the third are stochastically selected in order to make the model biologically plausible. The idea is to train the system to recognise certain input patterns in the connection region, which in turn leads to the appropriate path through the connections to the reaction layer. The learning algorithm must derive suitable weights for the connections.

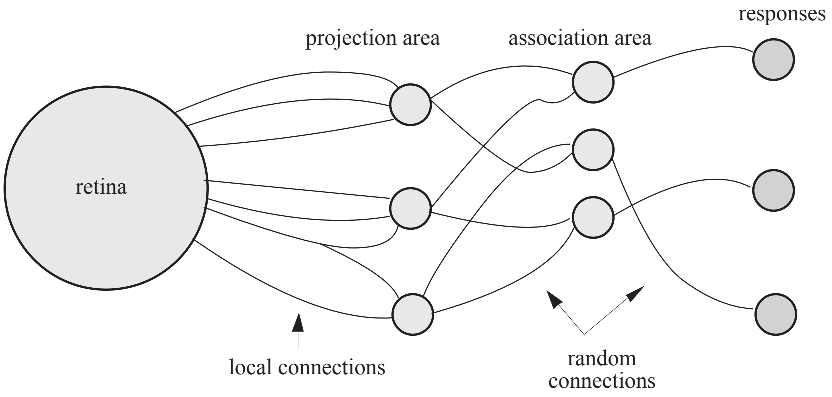


Fig. 5.1 The classical perceptron

Rosenblatt's model can only be understood by first analyzing the elementary computing units. From a formal point of view, the only difference between McCulloch–Pitts elements and perceptrons is the presence of weights in the networks. Rosenblatt also studied models with some other differences, such as putting a limit on the maximum acceptable fan-in of the units. Minsky and Papert distilled the essential features from Rosenblatt's model in order to study the computational capabilities of the perceptron under different assumptions. In the model used by these authors there is also a retina of pixels with binary values on which patterns are projected.

Some pixels from the retina are directly connected to logic elements called predicates which can compute a single bit according to the input. Interestingly, these predicates can be as computationally complex as we like; for example, each predicate could be implemented using a supercomputer. There are some constraints however, such as the number of points in the retina that can be simultaneously examined by each predicate or the distance between those points. The predicates transmit their binary values to a weighted threshold element which is in charge of reaching the final decision in a pattern recognition problem.

The question is then, what kind of patterns can be recognised in this massively parallel manner using a single threshold element at the output of the network? Are there limits to what we can compute in parallel using unlimited processing power for each predicate, when each predicate cannot itself look at the whole retina? The answer to this problem in some ways resembles the speedup problem in parallel processing, in which we ask what percentage of a computational task can be parallelised and what percentage is inherently sequential.

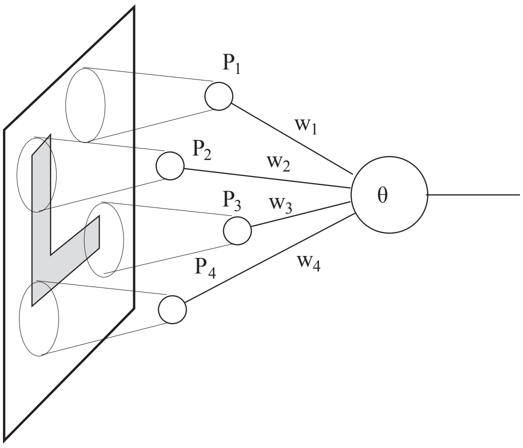


Fig. 5.2 Predicates and weights of a perceptron

Figure above illustrates the model discussed by Minsky and Papert. The predicates P_1 to P_4 deliver information about the points in the projection surface that comprise their receptive fields. The only restriction on the computational capabilities of the predicates is that they produce a binary value and the receptive field cannot cover the whole retina. The threshold element collects the outputs of the predicates through weighted edges and computes the final decision. The system consists in general of *n* predicates P_1, P_2,.......,P_n and the corresponding weights w_1,w_2,.....,w_n. The system fires only when $\sum_{i=1}^{n} w_i \ P_i \geq \theta$ where θ is the threshold of the computing unit at the output.

5.3 Computational Limits of the Perceptron Model

Minsky and Papert used their simplified perceptron model to investigate the computational capabilities of weighted networks. Early experiments with Rosenblatt's model had aroused unrealistic expectations in some quarters, and there was no clear understanding of the class of pattern recognition problems which it could solve efficiently. To explore this matter the number of predicates in the system is fixed, and although they possess unbounded computational power, the final bottleneck is the parallel computation with a single threshold element. This forces each processor to cooperate by producing a partial result pertinent to the global decision. The question now is which problems can be solved in this way and which cannot.

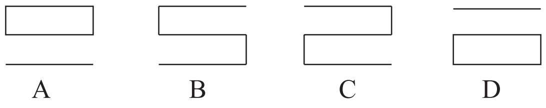
The system considered by Minsky and Papert at first appears to be a strong simplification of parallel decision processes, but it contains some of the most important elements differentiating between sequential and parallel processing. It is known that when some algorithms are parallelised, an irreducible sequential component sometimes limits the maximum achievable speedup. The mathematical relation between speedup and irreducible sequential portion of an algorithm is known as Amdahl's law. In the model considered above the central question is, 'Are there pattern recognition problems in which we are forced to analyze sequentially the output of the predicates associated with each receptive field or not?'

Minsky and Papert showed that problems of this kind do indeed exist which cannot be solved by a single perceptron acting as the last decision unit. The limits imposed on the receptive fields of the predicates are based on realistic assumptions. The predicates are fixed in advance and the pattern recognition problem can be made arbitrarily large (by expanding the retina). According to the number of points and their connections to the predicates, Minsky and Papert differentiated between

- Diameter limited perceptrons: the receptive field of each predicate has a limited diameter.
- Perceptrons of limited order: each receptive field can only contain up to a certain maximum number of points.
- Stochastic perceptrons: each receptive field consists of a number of randomly chosen points

Some patterns are more difficult to identify than others and this structural classification of perceptrons is a first attempt at defining something like complexity classes for pattern recognition. Connectedness is an example of a property that cannot be recognised by constrained systems.

Proposition: No diameter limited perceptron can decide whether a geometric figure is connected or not.



Proof: We proceed by contradiction, assuming that a perceptron can decide whether a figure is connected or not. Consider the four patterns shown above; notice that only the middle two are connected. Since the diameters of the receptive fields are limited, the patterns can be stretched horizontally in such a way that no single receptive field contains points from both the left and the right ends of the patterns. In this case we have three different groups of predicates: the first group consists of those predicates whose receptive fields contain points from the left side of a pattern.

Predicates of the second group are those whose receptive fields cover the right side of a pattern. All other predicates belong to the third group. In the figure below the receptive fields of the predicates are represented by circles.

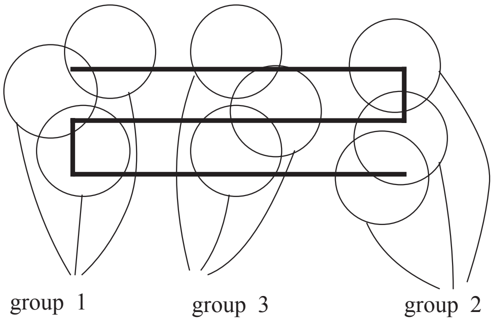


Fig. 5.3 Receptive fields of predicates

All predicates are connected to a threshold element through weighted edges which we denote by the letter w with an index. The threshold element decides whether a figure is connected or not by performing the computation

$$S = \sum_{P4 \in group\ 1} w1iPi + \sum_{P4 \in group\ 2} w2iPi + \sum_{P4 \in group\ 3} w3iPi - \theta \geq 0$$

If S is positive the figure is recognised as connected, as is the case, as given in figure above. If the disconnected pattern *A* is analyzed, then we should have *S < 0*. Pattern *A* can be transformed into pattern *B* without affecting the output of the predicates of group 3, which do not recognise the difference since their receptive fields do not cover the sides of the figures. The predicates of group 2 adjust their outputs by 2S so that now

$$S + \Delta_2 S \geq 0 => \Delta_2 S \geq -S$$

If pattern A is transformed into pattern C, the predicates of group 1 adjust their outputs so that the threshold element receives a net excitation that is,

$$S + \Delta_1 S \geq 0 => \Delta_1 S \geq -S$$

However, if pattern A is transformed into pattern D, the predicates of group 1 cannot distinguish this case from the one for figure C and the predicates of group 2 cannot distinguish this case from the one for figure B. Since the predicates of group 3 do not change their output we have,

$$\Delta S = \Delta_2 S + \Delta_1 S \geq -2S$$

And from this equation we get a equation below:

$$S + \Delta S \geq -S > 0$$

The value of the new sum can only be positive and the whole system classifies figure D as connected. Since this is a contradiction, such a system cannot exist. The above proposition states only that the connectedness of a figure is a global property which cannot be decided locally. If no predicate has access to the whole figure, then the only alternative is to process the outputs of the predicates sequentially. There are some other difficult problems for perceptrons. They cannot decide, for example, whether a set of points contains an even or an odd number of elements when the receptive fields cover only a limited number of points.

5.4 Implementation of Logical Functions

Weighted networks can achieve the same results as in the synthesis of Boolean functions using McCulloch–Pitts networks with fewer threshold gates, but the issue now is which functions can be implemented using a single unit. For this we will see the following points.

5.4.1 Geometric Interpretation

In each of the previous sections a threshold element was associated with a whole set of predicates or a network of computing elements. From now on, we will deal with perceptrons as isolated threshold elements which compute their output without delay.

Definition: A simple perceptron is a computing unit with threshold θ which, when receiving the n real inputs x_1, x_2, . . . , x_n through edges with the associated weights w_1,w_2, . . . ,w_n outputs 1 if the inequality $\sum_{i=1}^{n} w_i \ x_i \geq \theta$ holds and otherwise 0.

The origin of the inputs is not important, whether they come from other perceptrons or another class of computing units. The geometric interpretation of the processing performed by perceptrons is the same as with McCulloch–Pitts elements. A perceptron separates the input space into two half-spaces. For points belonging to one half-space the result of the computation is 0, for points belonging to the other it is 1. Figure below shows this for the case of two variables x_1 and x_2. A perceptron with threshold 1, at which two edges with weights 0.9 and 2.0 impinge, tests the condition

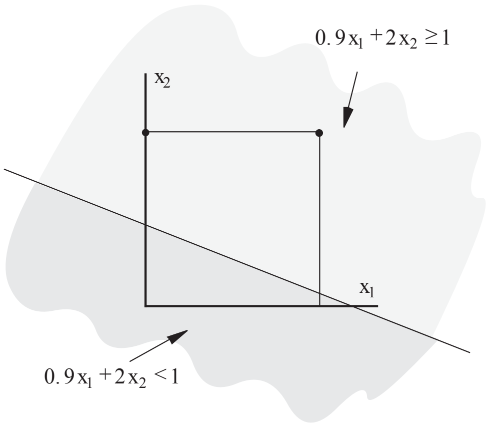


Fig. 5.4 Separation of input space with a perceptron

$$0.9_{x1} + 2_{x2} \geq 1$$

It is possible to generate arbitrary separations of input space by adjusting the parameters of this example. In many cases it is more convenient to deal with perceptrons of threshold zero only. This corresponds to linear separations which are forced to go through the origin of the input space. The two perceptrons in figure below are equivalent. The threshold of the perceptron to the left has been converted into the weight $-\theta$ of an additional input channel connected to the constant. This extra weight connected to a constant is called the bias of the element.

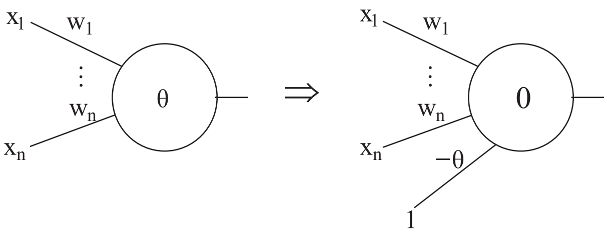


Fig. 5.5 A perceptron with a bias

Most learning algorithms can be stated more concisely by transforming thresholds into biases. The input vector (x_1, x_2,...., x_n) must be extended with an additional 1 and the resulting ($n+1$)-dimensional vector (x_1, x_2,....,x_n, 1) is called the extended input vector. The extended weight vector associate with this perceptron is (w_1,....,w_n, w_{n+1}), whereby $w_{n+1} = -\theta$.

5.4.2 The XOR Problem

We can now deal with the problem of determining which logical functions can be implemented with a single perceptron. A perceptron network is capable of computing any logical function, since perceptrons are even more powerful than un-weighted McCulloch–Pitts elements. If we reduce the network to single element, which functions are still computable? Taking the functions of two variables as an example we can gain some insight into this problem. Table below shows all 16 possible Boolean functions of two variables f_0 to f_{15}. Each column f_1 shows the value of the function for each combination of the two variables x_1 and x_2. The function f_0, for example, is the zero function whereas f_{14} is the OR-function.

x_1	x_2	f_0	f_1	f_2	f_3	f_4	f_5	f_6	f_7	f_8	f_9	f_{10}	f_{11}	f_{12}	f_{13}	f_{14}	f_{15}
0	0	0	1	0	1	0	1	0	1	0	1	0	1	0	1	0	1
1	1	0	0	1	1	0	0	1	1	0	0	1	1	0	0	1	1
1	0	0	0	0	0	1	1	1	1	0	0	0	0	1	1	1	1
1	1	0	0	0	0	0	0	0	0	1	1	1	1	1	1	1	1

Perceptron-computable functions are those for which the points whose function value is 0 can be separated from the points whose function value is 1 using a line. Figure below shows two possible separations to compute the OR and the AND functions.

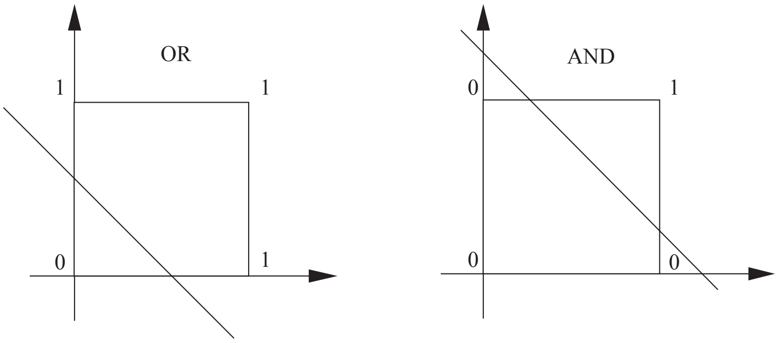


Fig. 5.6 Separations of input space corresponding to OR and AND

It is clear that two of the functions in the table cannot be computed in this way. They are the function XOR and identity (f_6 and f_9). It is intuitively evident that no line can produce the necessary separation of the input space. This can also be shown analytically. Let w_1 and w_2 be the weights of a perceptron with two inputs, and θ its threshold. If the perceptron computes the XOR function the following four inequalities must be fulfilled:

$x_1 = 0\ x_2 = 0\ w_1x_1 + w_2x_2 = 0 \quad \Rightarrow 0 < \theta$
$x_1 = 1\ x_2 = 0\ w_1x_1 + w_2x_2 = w_1 \quad \Rightarrow w_1 \geq \theta$
$x_1 = 0\ x_2 = 1\ w_1x_1 + w_2x_2 = w_2 \quad \Rightarrow w_2 \geq \theta$
$x_1 = 1\ x_2 = 1\ w_1x_1 + w_2x_2 = w_1 + w2 \Rightarrow w_1 + w_2 < \theta$

Since θ is positive, according to the first inequality, w_1 and w_2 are positive too, according to the second and third inequalities. Therefore the inequality $w_1 + w_2 < \theta$ cannot be true. This contradiction implies that no perceptron capable of computing the XOR function exists. An analogous proof holds for the function f_9.

5.5 Linearly Separable Functions

The example of the logical functions of two variables shows that the problem of perceptron computability must be discussed in more detail. In this section we provide the necessary tools to deal more effectively with functions of n arguments.

5.5.1 Linear Separability

We can deduce from our experience with the XOR function that many other logical functions of several arguments must exist which cannot be computed with a threshold element. This fact has to do with the geometry of the n-dimensional hypercube whose vertices represent the combination of logic values of the arguments. Each logical function separates the vertices into two classes. If the points whose function value is 1 cannot be separated with a linear cut from the points whose function value is 0, the function is not perceptron-computable. The following two definitions give this problem a more general setting.

Definition: Two sets of points A and B in an n-dimensional space are called linearly separable if $n + 1$ real numbers $w_1, \ldots, w_{n+1}$ exist, such that every point $(x_1, x_2, \ldots, x_n) \in A$ satisfies $\sum_{i=1}^{n} w_i\ x_i < w_{n+1}$ and every point $(x_1, x_2, \ldots, x_n)$ 2 B satisfies $\sum_{i=1}^{n} w_i\ x_i < w_{n+1}$.

Since a perceptron can only compute linearly separable functions, an interesting question is how many linearly separable functions of n binary arguments there are. When n = 2, 14 out of the 16 possible Boolean functions are linearly separable. When n = 3, 104 out of 256 and when n = 4, 1882 out of 65536 possible functions are linearly separable. Although there has been extensive research on linearly separable functions in recent years, no formula for expressing the number of linearly separable functions as a function of n has yet been found. However we will provide some upper bounds for this number in the following chapters.

5.5.2 Duality of Input Space and Weight Space

The computation performed by a perceptron can be visualised as a linear separation of input space. However, when trying to find the appropriate weights for a perceptron, the search process can be better visualised in weight space. When *m* real weights must be determined, the search space is the whole of IR^m for a perceptron with n input lines finding the appropriate linear separation amounts to finding $n + 1$ free parameters (n weights and the bias).

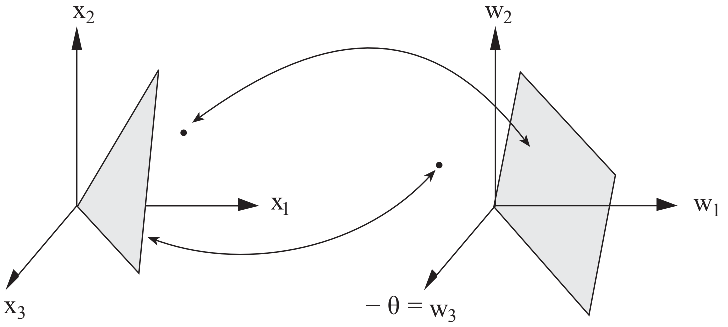


Fig. 5.7 Illustration of the duality of input and weight space

These n+1 parameters represent a point in (n+1)-dimensional weight space. Each time we pick one point in weight space we are choosing one combination of weights and a specific linear separation of input space. This means that every point in (n + 1) dimensional weight space can be associated with a hyper plane in (n + 1)-dimensional extended input space. The above figure shows an example. Each combination of three weights, w_1, w_2, w_3, which represent a point in weight space, defines a separation of input space with the plane,

$$w_1x_1 + w_2x_2 + w_3x_3 = 0.$$

There is the same kind of relation in the inverse direction, from input to weight space. If we want the point x_1, x_2, x_3 to be located in the positive half-space defined by a plane, we need to determine the appropriate weights w_1, w_2 and w_3. The inequality must hold. However this inequality defines a linear separation of weight space, that is, the point (x_1, x_2, x_3) defines a cutting plane in weight space.

$$w_1x_1 + w_2x_2 + w_3x_3 \geq 0$$

Points in one space are mapped to planes in the other and vice versa. This complementary relation is called duality. Input and weight space are dual spaces and we can visualise the computations done by perceptrons and learning algorithms in any one of them.

5.5.3 The Error Function in Weight Space

Given two sets of patterns which must be separated by a perceptron, a learning algorithm should automatically find the weights and threshold necessary for the solution of the problem. The perceptron learning algorithm can accomplish this for threshold units. Although proposed by Rosenblatt it was already known in another context.

Assume that the set *A* of input vectors in n-dimensional space must be separated from the set *B* of input vectors in such a way that a perceptron computes the binary function f_w with $f_w(x) = 1$ for x∈A and $f_w(x) = 0$ for x∈B. The binary function fw depends on the set *w* of weights and threshold. The error function is the number of false classifications obtained using the weight vector *w*. It can be defined as:

$$E\,(w) = \sum_{x\in A} (1 - f_w(x)) + \sum_{x\in B} f_w(x)$$

This is a function defined over all of weight space and the aim of perceptron learning is to minimise it. Since E(*w*) is positive or zero, we want to reach the global minimum where E(*w*) = 0. This will be done by starting with a random weight vector w, and then searching in weight space a better alternative, in an attempt to reduce the error function E(*w*) at each step.

5.5.4 General Decision Curves

A perceptron makes a decision based on a linear separation of the input space. This reduces the kinds of problem solvable with a single perceptron. More general separations of input space can help to deal with other kinds of problem unsolvable with a single threshold unit. Assume that a single computing unit can produce the separation shown in Figure below. Such a separation of the input space into two regions would allow the computation of the XOR function with a single unit. Functions used to discriminate between regions of input space are called decision curves. Some of the decision curves which have been studied are polynomials and splines.

In statistical pattern recognition problems we assume that the patterns to be recognised are grouped in clusters in input space. Using a combination of decision curves we try to isolate one cluster from the others. One alternative is combining several perceptrons to isolate a convex region of space. Other alternatives which have been studied are, for example, so-called Sigma-Pi units which, for a given input $x_1, x_2, \ldots, x_n$, compute the sum of all or some partial products of the form $x_i x_j$.

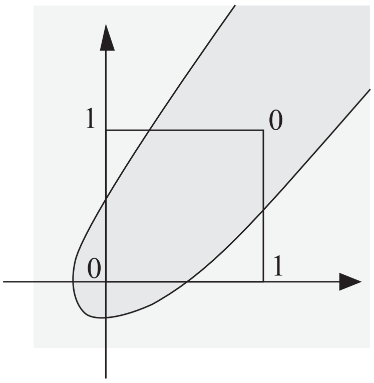


Fig. 5.8 Non-linear separation of input space

In the general case we want to distinguish between regions of space. A neural network must learn to identify these regions and to associate them with the correct response. The main problem is determining whether the free parameters of these decision regions can be found using a learning algorithm.

5.6 Applications and Biological Analogy

The appeal of the perceptron model is grounded on its simplicity and the wide range of applications that it has found. As we show in this section, weighted threshold elements can play an important role in image processing and computer vision.

5.6.1 Edge Detection with Perceptrons

A good example of the pattern recognition capabilities of perceptrons is edge detection. Assume that a method of extracting the edges of a figure darker than the background or the converse is needed. Each pixel in the figure is compared to its immediate neighbours and in the case where the pixel is black and one of its neighbours white, it will be classified as part of an edge. This can be programmed sequentially in a computer, but since the decision about each point uses only local information, it is straightforward to implement the strategy in parallel.

Assume that the figures to be processed are projected on a screen in which each pixel is connected to a perceptron, which also receives inputs from its immediate neighbours. Figure Edge detection operator shows the shape of the receptive field (a so-called Moore neighbourhood) and the weights of the connections to the perceptron. The central point is weighted with 8 and the rest with −1. In the field of image processing this is called a convolution operator, because it is used by centering it at each pixel of the image to produce a certain output value for each pixel.

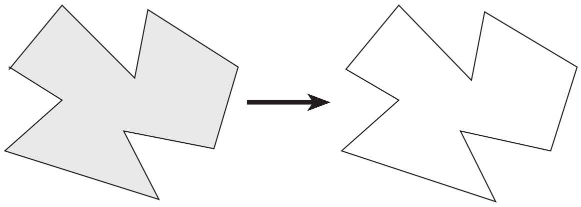


Fig. 5.9 Example of edge detection

The operator shown has a maximum at the center of the receptive field and local minima at the periphery.

-1	-1	-1
-1	8	-1
-1	-1	-1

Fig. 5.10 Edge detection operator

Figure below shows the kind of interconnection we have in mind. A perceptron is needed for each pixel. The interconnection pattern repeats for each pixel in the projection lattice, taking care to treat the borders of the screen differently. The weights are those given by the edge detection operator.

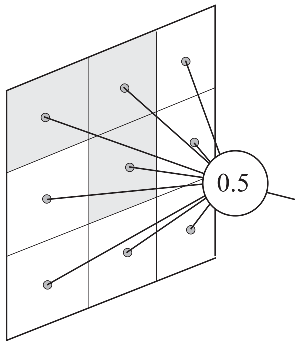


Fig. 5.11 Connection of a perceptron to the projection grid

For each pattern projected onto the screen, the weighted input is compared to the threshold 0.5. When all points in the neighbourhood are black or all white, the total excitation is 0. In the situation shown below the total excitation is 5 and the point in the middle belongs to an edge. There are many other operators for different uses, such as detecting horizontal or vertical lines or blurring or making a picture sharper. The size of the neighbourhood can be adjusted to the specific application. For example, the operator below can be used to detect the vertical edges between a white surface to the left and a dark one to the right.

$$\begin{pmatrix} -1 & 0 & 1 \\ -1 & 0 & 1 \\ -1 & 0 & 1 \end{pmatrix}$$

5.6.2 The Structure of the Retina

The visual pathway is the part of the human brain which is best understood. The retina can be conceived as a continuation of this organ, since it consists of neural cells organised in layers and capable of providing in situ some of the information processing necessary for vision. In frogs and other small vertebrates some neurons have been found directly in the retina which actually fire in the presence of a small blob in the visual field. These are bug detectors which tell these animals when a small insect has been sighted.

Researchers have found that the cells in the retina are interconnected in such a way that each nerve going from the eyes to the brain encodes a summary of the information detected by several photoreceptors in the retina. As in the case of the convolution operators discussed previously, each nerve transmits a signal which depends on the relative luminosity of a point in relation to its immediate neighbourhood.

Study the figure below it shows the interconnection pattern found in the retina. The cones and rods are the photoreceptors, which react to photons by depolarising. Horizontal cells compute the average luminosity in a region by connecting to the cones or rods in this region. Bipolar and ganglion cells fire only when the difference in the luminosity of a point is significantly higher than the average light intensity.

Although not all details of the retinal circuits have been reverse-engineered, there is a recurrent feature: each receptor cell in the retina is part of a roughly circular receptive field. The receptive fields of neighbouring cells overlap. Their function is similar to the edge processing operators, because the neighbourhood inhibits a neuron whereas a photoreceptor excites it, or conversely. This kind of weighting of the input has a strong resemblance to the so-called Mexican hat function.

David Marr tried to summarise what we know about the visual pathway in humans and proposed his idea of a process in three stages, in which the brain first decomposes an image into features (edges, blobs, etc.), which are then used to build an interpretation of surfaces, depth relations and groupings of tokens and which in turn leads to a full interpretation of the objects present in the visual field (the primal sketch). He tried to explain the structure of the retina from the point of view of the computational machinery needed for vision. He proposed that at a certain stage of the computation the retina blurs an image and then extracts from it contrast information.

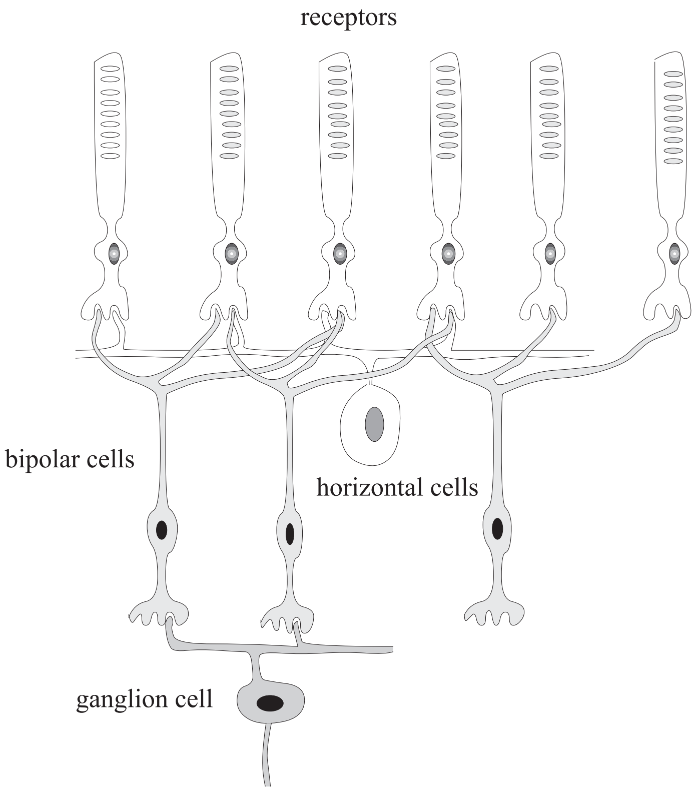
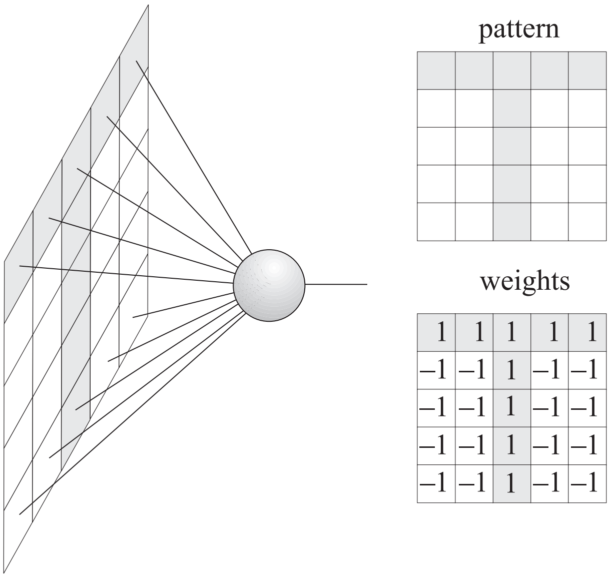


Fig. 5.12 The interconnection pattern in the retina



1	1	1	1	1
–1	–1	1	–1	–1
–1	–1	1	–1	–1
–1	–1	1	–1	–1
–1	–1	1	–1	–1

Fig. 5.13 Feature detector for the pattern T

Blurring an image can be done by averaging at each pixel the values of this pixel and its neighbours. A Gaussian distribution of weights can be used for this purpose. Information about changes in darkness levels can be extracted using the sum of the second derivatives of the illumination function, the so-called Laplace operator $\nabla^2 = \frac{\partial^2}{\partial x^2} + \frac{\partial^2}{\partial y^2}$. The composition of the Laplace operator and a Gaussian blurring corresponds to the Mexican hat function. Processing of the image is done by computing the convolution of the discrete version of the operator with the image. The 3 × 3 discrete version of this operator is the edge detection operator which we used before.

Different levels of blurring, and thus feature extraction at several different resolution levels, can be controlled by adjusting the size of the receptive fields of the computing units. It seems that the human visual pathway also exploits feature detection at several resolution levels, which has led in turn to the idea of using several resolution layers for the computational analysis of images.

5.6.3 Pyramidal Networks and the Neocognitron

Single perceptrons can be thought of as feature detectors. Take the case of above figure in which a perceptron is defined with weights adequate for recognising the letter 'T' in which t pixels are black. If another 'T' is presented, in which one black pixel is missing, the excitation of the perceptron is t − 1. The same happens if one white pixel is transformed into a black one due to noise, since the weights of the connections going from points that should be white are −1. If the threshold of the perceptron is set to t − 1, then this perceptron will be capable of correctly classifying patterns with one noisy pixel.

By adjusting the threshold of the unit, 2, 3 or more noisy pixels can be tolerated. Perceptrons thus compute the similarity of a pattern to the ideal pattern they have been designed to identify, and the threshold is the minimal similarity that we require from the pattern. Note that since the weights of the perceptron are correlated with the pattern it recognises, a simple way to visualise the connections of a perceptron is to draw the pattern it identifies in its receptive field. This technique will be used below.

The problem with this pattern recognition scheme is that it only works if the patterns have been normalised in some way, that is, if they have been centered in the window to which the perceptron connects and their size does not differ appreciably from the ideal pattern. Also, any kind of translational shift will lead to ideal patterns no longer being recognised. The same happens in the case of rotations.

An alternative way of handling this problem is to try to detect patterns not in a single step, but in several stages. If we are trying, for example, to recognise handwritten digits, then we could attempt to find some small distinctive features such as lines in certain orientations and then combine our knowledge about the presence or absence of these features in a final logical decision. We should try to recognise these small features, regardless of their position on the projection screen.

The cognitron and neocognitron were designed by Fukushima and his colleagues as an attempt to deal with this problem and in some way to try to mimic the structure of the human vision pathway. The main idea of the neocognitron is to transform the contents of the screen into other screens in which some features have been enhanced, and then again into other screens, and so on, until a final decision is made.

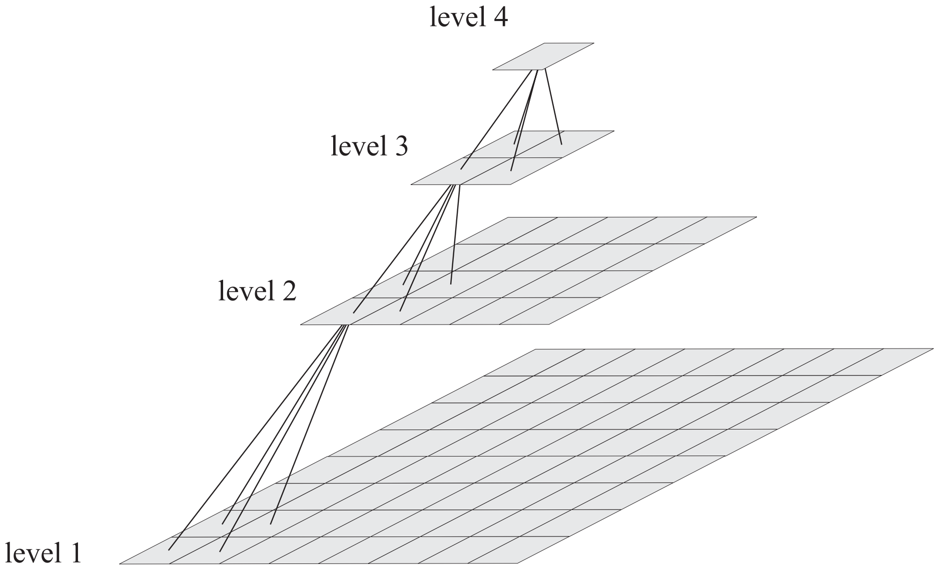


Fig. 5.14 Pyramidal architecture for image processing

The resolution of the screen can be changed from transformation to transformation or more screens can be introduced, but the objective is to reduce the representation to make a final classification in the last stage based on just a few points of input. The general structure of the neural system proposed by Fukushima is a kind of variant of what is known in the image processing community as a pyramidal architecture, in which the resolution of the image is reduced by a certain factor from plane to plane.

Figure above shows an example of a quad-pyramid, that is, a system in which the resolution is reduced by a factor of four from plane to plane. Each pixel in one of the upper planes connects to four pixels in the plane immediately below and deals with them as elements of its receptive field. The computation to determine the value of the upper pixel can be arbitrary, but in our case we are interested in threshold computations. Note that in this case receptive fields do not overlap. Such architectures have been studied intensively to determine their capabilities as data structures for parallel algorithms. The neocognitron has a more complex architecture. The image is transformed from the original plane to other planes to look for specific features.

Figure below shows the general strategy adopted in the neocognitron. The projection screen is transformed, deciding for each pixel if it should be kept white or black. This can be done by identifying the patterns shown for each of the three transformations by looking at each pixel and its eight neighbours. In the case of the first transformed screen only horizontal lines are kept; in the second screen only vertical lines and in the third screen only diagonal lines. The convolution operators needed have the same distribution of positive weights, as shown for each screen.

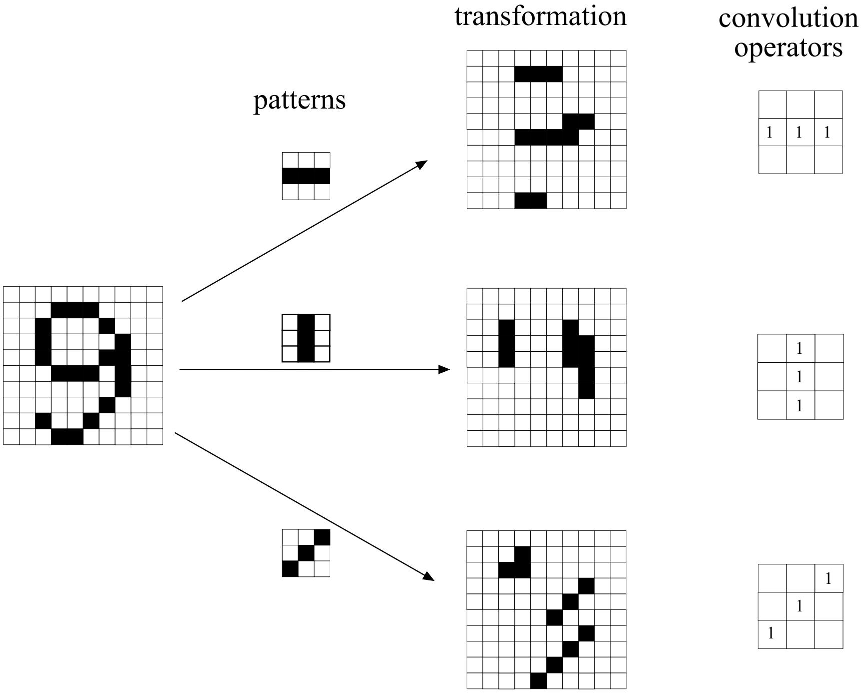


Fig. 5.15 Feature extraction in the neocognitron

The rest of the weights is 0. Note that these special weights work better if the pattern has been previously ‘skeletonized’. Strictly speaking Fukushima’s neocognitron uses linear computing units and not perceptrons. The units compute their total weighted input and this is interpreted as a kind of correlation index with the patterns that each unit can identify. This means that black and white patterns are transformed into patterns with shadings of gray, according to the output of each mapping unit.

Study the figure below, it shows the general structure of the neocognitron network. The input layer of the network is called UC^0. This input layer is processed and converted into twelve different images numbered US_0^1 0 to US_{11}^1 with the same resolution. The super index in front of a name is the layer number and the sub-index the number of the plane in this layer. The operators used to transform from UC^0 to each of the US_i^1 planes have a receptive field of 3×3 pixels and one operator is associated with each pixel in the US_i^1. In each plane only one kind of feature is recognised. The first plane US_1^1, for example, could contain all the vertical edges found in UC^0, the second plane U S_2^1, only diagonal edges, and so forth. The next level of processing is represented by the UC_j^1 planes.

Each pixel in one of these planes connects to a receptive field in one or two of the underlying US_i^1, planes. The weights are purely excitatory and the effect of this layer is to overlap the activations of the selected US_i^1, images, blurring them at the same time that is, making the patterns wider. This is achieved by transforming each pixel's value in the weighted average of its own and its neighbour's values.

In the next level of processing each pixel in a US_i^2 plane connects to a receptive field at the same position in all of the U$C_j^1C_j^1$ images. At this level the resolution of the US_i^2 planes can be reduced, as in standard pyramidal architectures.

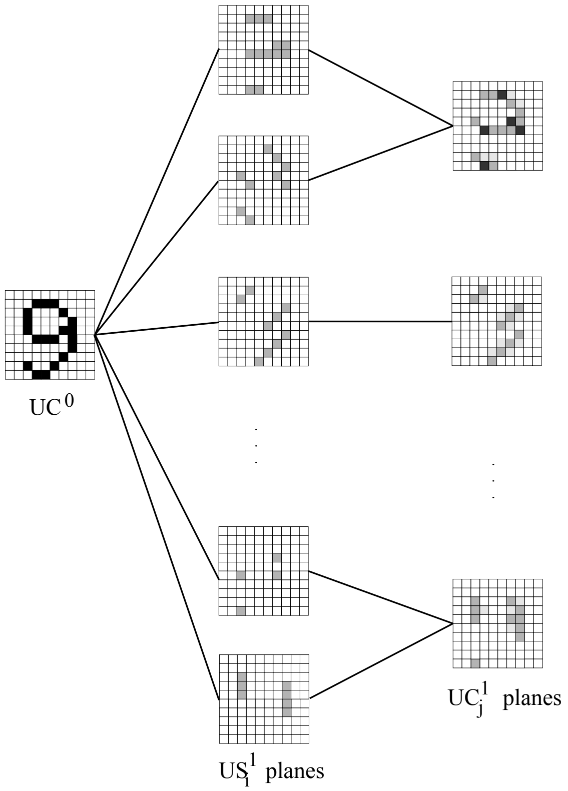


Fig. 5.16 The architecture of the neocognitron

Figure above shows the sizes of the planes used by Fukushima for handwritten digit recognition. Several layers of alternating US and UC planes are arranged in this way until at the plane UC4 a classification of the handwritten digit in one of the classes 0, . . . , 9 is made. Finding the appropriate weights for the classification task is something we discuss in the next chapter. Fukushima has proposed several improvements of the original model over the years. The main advantage of the neocognitron as a pattern recognition device should be its tolerance to shifts and distortions. Since the UC layers blur the image and the US layers look for specific features, a certain amount of displacement or rotation of lines should be tolerated.

This can happen, but the system is highly sensitive to the training method used and does not outperform other simpler neural networks. Other authors have examined variations of the neocognitron which are more similar to pyramidal networks. The neocognitron is just an example of a class of network which relies extensively on convolution operators and pattern recognition in small receptive fields. For an extensive discussion of the neocognitron consult.

5.6.4 The Silicon Retina

Carver Mead's group at Caltech has been active for several years in the field of neuromorphic engineering, that is, the production of chips capable of emulating the sensory response of some human organs. Their silicon retina, in particular, is able to simulate some of the features of the human retina.

Mead and his collaborators modelled the first three layers of the retina: the photoreceptors, the horizontal, and the bipolar cells. The horizontal cells are simulated in the silicon retina as a grid of resistors. Each photoreceptor (the dark points in figure below) is connected to each of its six neighbours and produces a potential difference proportional to the logarithm of the luminosity measured. The grid of resistors reaches electric equilibrium when an average potential difference has settled in. The individual neurons of the silicon retina fire only when the difference between the average and their own potential reaches a certain threshold.

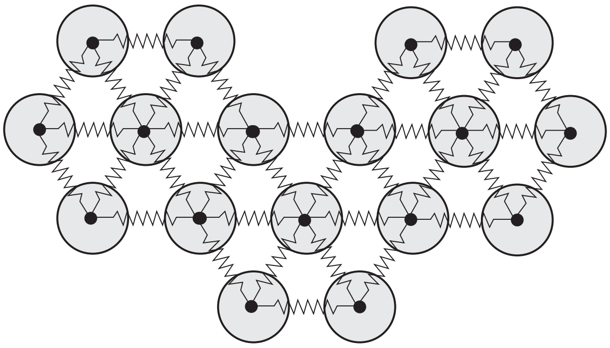


Fig. 5.17 Diagram of a portion of the silicon retina

The average potential S of n potentials Si can be computed by letting each potential Si be proportional to the logarithm of the measured light intensity Hi, that is,

$$S = \frac{1}{n}\sum_{i=1}^{n} Si = \frac{1}{n}\sum_{i=1}^{n} log Hi$$

This expression can be transformed into,

$$S = \frac{1}{n}\log(H_1, H_2 \ldots H_n) = \log(H_1, H_2 \ldots H_n)^{1/n}$$

The equation tells us that the average potential is the logarithm of the geometric mean of the light intensities. A unit only fires when the measured intensity Si minus the average intensity lies above a certain threshold, that is,

$$\log(H_i) - \log(H_1H_2 \ldots H_n)^{1/n} \geq \gamma$$

And this is valid only when,

$$\log \frac{Hi}{(H1H2 \ldots Hn)^{1/n}} \geq \gamma$$

The units in the silicon retina fire when the relative luminosity of a point with respect to the background is significantly higher, such as in a human retina. We know from optical measurements that when outside on a sunny day, the black letters in a book reflect more photons on our eyes than white paper does in a room. Our eyes adjust automatically to compensate for the luminosity of the background so that we can recognise patterns and read books inside and outside.

Summary

- Learning can only be implemented by modifying the connection pattern of the network and the thresholds of the units, but this is necessarily more complex than just adjusting numerical parameters.
- In 1958 Frank Rosenblatt, an American psychologist, proposed the perceptron, a more general computational model than McCulloch–Pitts units.
- The classical perceptron is in fact a whole network for the solution of certain pattern recognition problems.
- Rosenblatt's model can only be understood by first analyzing the elementary computing units.
- Some pixels from the retina are directly connected to logic elements called predicates which can compute a single bit according to the input.
- Minsky and Papert used their simplified perceptron model to investigate the computational capabilities of weighted networks.
- No diameter limited perceptron can decide whether a geometric figure is connected or not.
- Weighted networks can achieve the same results as in the synthesis of Boolean functions using McCulloch–Pitts networks with fewer threshold gates.
- A simple perceptron is a computing unit with threshold θ which, when receiving the n real inputs $x_1, x_2, \ldots,$ x_n through edges with the associated weights $w_1, w_2, \ldots, w_n$ outputs 1 if the inequality $\sum_{i=1}^{n} w_i \ x_i \geq \theta$ holds and otherwise 0.
- A perceptron network is capable of computing any logical function, since perceptrons are even more powerful than un-weighted McCulloch–Pitts elements.
- The computation performed by a perceptron can be visualised as a linear separation of input space.
- A perceptron makes a decision based on a linear separation of the input space.
- The appeal of the perceptron model is grounded on its simplicity and the wide range of applications that it has found.

References

- Horvath, S., 2011. *Weighted Network Analysis: Applications in Genomics and Systems Biology*, Springer.
- Bollobás, B., Kozma, R. & Miklós, D., 2009. *Handbook of Large-Scale Random Networks*, Springer.
- *Weighted Networks – The Perceptron*, [Online] Available at: <http://page.mi.fu-berlin.de/rojas/neural/chapter/K3.pdf> [Accessed 19 July 2012].
- Newman, J. E. M., *Analysis of Weighted Networks*, [Online] Available at: <http://arxiv.org/pdf/condmat/0407503.pdf> [Accessed 19 July 2012].
- 2012. *Lecture 10 - Neural Networks (May 3, 2012)*, [Video Online] Available at: <http://www.youtube.com/watch?v=Ih5Mr93E-2c> [Accessed 19 July 2012].
- 2012. *Neural Network Tutorial: The back-propagation Algorithm (Part 2)* [Video Online] Available at: <http://www.youtube.com/watch?v=zpykfC4VnpM&feature=related> [Accessed 19 July 2012].

Recommended Reading

- Neruda, R. & Koutník, J., 2008. *Artificial Neural Networks - ICANN 2008: 18th International Conference, Prague, Czech Republic, September 3-6, Proceedings*, Springer.
- Tosh, C. & Ruxton, D. G., 2010. *Modelling Perception with Artificial Neural Networks*, Cambridge University Press.
- Webb, R. A., Copsey, D. K. & Cawley, G., 2011. *Statistical Pattern Recognition*, 3rd ed. John Wiley & Sons.

Self Assessment

1. Which of the following is the pattern recognition capabilities of perceptrons?
    a. Edge detection
    b. Biological analogy
    c. Centering
    d. Image processing

2. _________ explain the structure of the retina from the point of view of the computational machinery needed for vision.
    a. McCulloch
    b. Frank Rosenblatt
    c. Papert
    d. David Marr

3. Laplace operator and a Gaussian blurring corresponds to the Mexican ____ function.
    a. logical
    b. hat
    c. Boolean
    d. OR

4. __________ operator shows the shape of the receptive field and the weights of the connections to the perceptron.
    a. Edge Processing
    b. Edge detection
    c. Convolution
    d. Laplace

5. The cognitron and neocognitron were designed by________ .
    a. Fukushima
    b. Frank Rosenblatt
    c. Papert
    d. David Marr

6. A simple perceptron is a computing unit with threshold ______.
    a. θ
    b. Φ
    c. α
    d. µ

7. The error function is the number of false classifications obtained using the weight vector ______.
    a. x
    b. µ
    c. α
    d. w

8. Functions used to discriminate between regions of input space are called ________.
    a. edge detection
    b. decision curves
    c. perceptron
    d. Gaussian distribution

9. The horizontal cells are simulated in the ________ retina as a grid of resistors.
    a. bipolar
    b. neuron
    c. silicon
    d. photoreceptors

10. The units in the ________ fire when the relative luminosity of a point with respect to the background is significantly higher.
    a. silicon retina
    b. bipolar retina
    c. neuron retina
    d. photoreceptors

Chapter VI

Adaptive Resonance Theory

Aim

The aim of this chapter is to:

- introduce adaptive resonance theory
- elucidate phases of ART1
- explain direct access of learned patterns

Objectives

The objectives of this chapter are to:

- explain basic properties of adaptive resonance theory
- explicate elements in the recognition module of the ART1 system
- elucidate art module

Learning outcome

At the end of this chapter, you will be able to:

- understand direct access after learning stabilises
- distinguish between model implementation
- describe adaptive resonance theorems

6.1 Introduction

The adaptive resonance theory (ART) has been developed to avoid the stability-plasticity dilemma in competitive networks learning. The stability-plasticity dilemma addresses how a learning system can preserve its previously learned knowledge while keeping its ability to learn new patterns. ART architecture models can self-organise in real time producing stable recognition while getting input patterns beyond those originally stored. ART is a family of different neural architectures. The first and most basic architecture is ART1. ART1 can learn and recognise binary patterns. ART2 is a class of architectures categorising arbitrary sequences of analog input patterns. ART is used in modelling such as invariant visual pattern recognition where biological equivalence is discussed.

An ART system consists of two subsystems, an attentional subsystem and an orienting subsystem. The stabilisation of learning and activation occurs in the attentional subsystem by matching bottom-up input activation and top-down expectation. The orienting subsystem controls the attentional subsystem when a mismatch occurs in the attentional subsystem. In other words, the orienting subsystem works like a novelty detector.

An ART system has four basic properties:

- The first is the self-scaling computational units. The attentional subsystem is based on competitive learning enhancing pattern features but suppressing noise.
- The second is self-adjusting memory search. The system can search memory in parallel and adaptively change its search order.
- Third, already learned patterns directly access their corresponding category.
- Finally, the system can adaptively modulate attentional vigilance using the environment as a teacher. If the environment disapproves the current recognition of the system, it changes this parameter to be more vigilant.

There are two models of ART1, a slow-learning and a fast-learning one. The slow learning model is described by in terms of differential equations while the fast learning model uses the results of convergence in the slow learning model. In this chapter we will not show a full implementation on ART1, instead an implementation of the fast learning model will be more efficient and sufficient to show the ART1 architecture behaviour.

6.2 Model Description

ART1 is the simplest ART learning model specifically designed for recognising binary patterns. The ART1 system consists of an attentional subsystem and an orienting subsystem as shown in figure below.

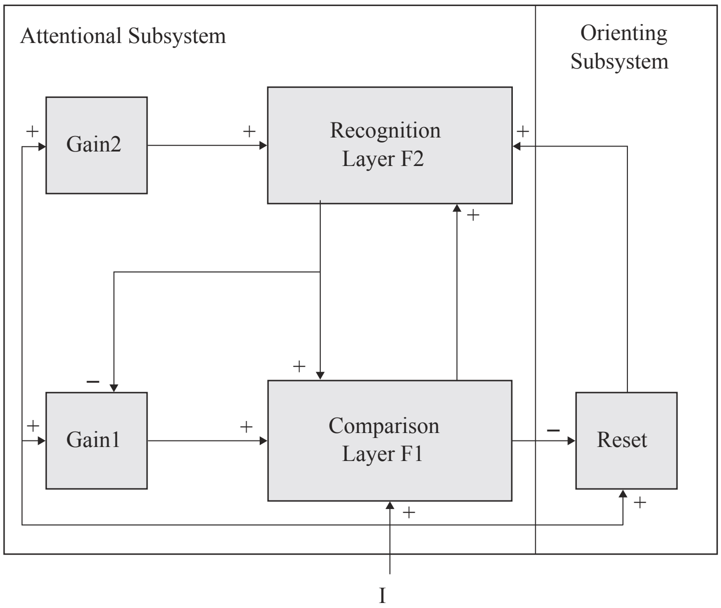


Fig. 6.1 ART1 model

ART1 consists of an attentional subsystem and an orienting subsystem. The attentional subsystem has two short term memory (STM) stages, *F1* and *F2*. Long term memory (LTM) traces between *F1* and *F2* multiply the signal in these pathways. Gain control signals enable *F1* and *F2* to distinguish current stages of a running cycle. STM reset wave inhibits active *F2* cells when mismatches between bottom-up and top-down signals occur at *F1*.

The attentional subsystem consists of two competitive networks, the comparison layer *F1* and the recognition layer *F2*, and two control gains, Gain 1 and Gain 2. The orienting subsystem contains the reset layer for controlling the attentional subsystem overall dynamics.

The comparison layer receives the binary external input passing it to the recognition layer responsible for matching it to a classification category. This result is passed back to the comparison layer to find out if the category matches that of the input vector. If there is a match a new input vector is read and the cycle starts again. If there is a mismatch the orienting system is in charge of inhibiting the previous category in order to get a new category match in the recognition layer. The two gains control the activity of the recognition and comparison layer, respectively. A processing element x_{1i} in layer F_1 is shown in figure below.

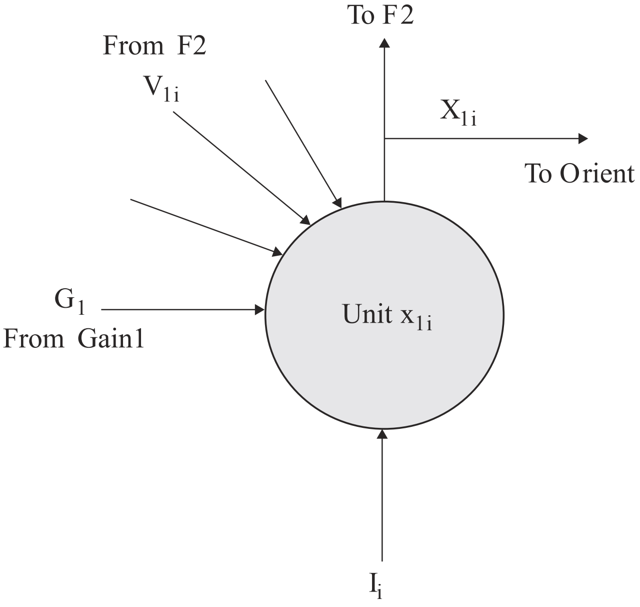


Fig. 6.2 A processing unit x_{1i} in *F1* receives input from: pattern I_i, gain control signal *G1* and V_{1i} equivalent to output X_{2j} from *F2* multiplied by interconnection weight E_{21ij}. The local activity serving also as unit output is X_{1i}.

The excitatory input to x_{1i} in layer *F1* comes from three sources:

- the external input vector I_i
- the control gain *G1*
- the internal network input *V1i* made of the output from *F2* multiplied appropriate connections weights

There is no inhibitory input to the neuron. The output of the neuron is fed to the *F2* layer as well as the orient subsystem. A processing element x_{2j} in layer *F2* is shown in figure below.

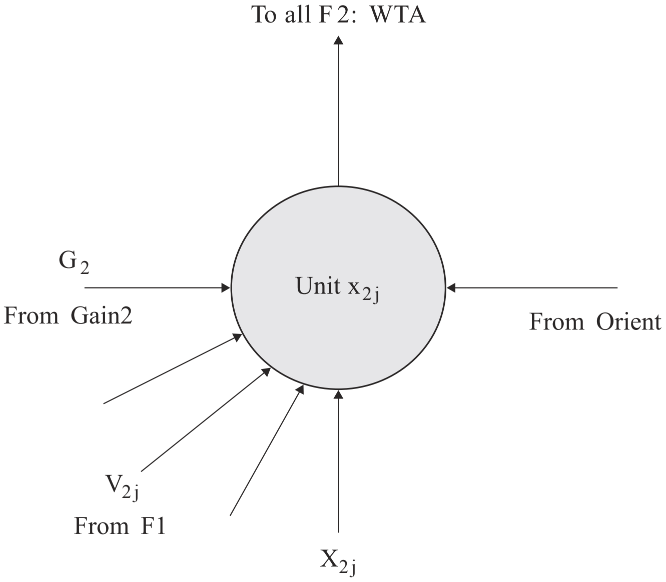


Fig. 6.3 A processing element x2j in F2 receives input from: gain control signal G2 and V2j equivalent to output X1i from F1 multiplied by interconnection weight W12ji. The local activity is also the unit output X2j.

The excitatory input to x_{2j} in *F2* comes from three sources:

- The orient subsystem
- The control gain *g2*
- The internal network input V_{2j} made of the output from *F1* multiplied appropriate connections weights

There is no inhibitory input to the neuron. The output of the neuron is fed to the F1 layer as well as the Gain 1 control. The original dynamic equations handle both binary and analog computations. We shall concentrate here on the binary model. Processing in ART1 can be divided into four phases:

- Recognition
- Comparison
- Search
- Learning

6.2.1 Recognition

Initially, in the recognition or bottom-up activation, no input vector I is applied disabling all recognition in *F2* and making the two control gains, *G1* and *G2*, equal to zero. This causes all *F2* elements to be set to zero, giving them an equal chance to win the subsequent recognition competition. When an input vector is applied one or more of its components must be set to one thereby making both *G1* and *G2* equal to one.

Thus, the control gain *G1* depends on both the input vector *I* and the output *X2* from *F2,*

$$G_1 = \begin{cases} 1 \ if\ I \neq 0\ and\ X_2 = 0 \\ 0 \qquad otherwise \end{cases}$$

In other words, if there is an input vector *I* and *F2* is not actively producing output, then *G1* = 1. Any other combination of activity on *I* and *F2* would inhibit the gain control from exciting units on *F1*. On the other hand, the output *G2* of the gain control module depends only on the input vector I,

$$G_1 = \begin{cases} 1 \quad if\ I \neq 0 \\ 0 \ otherwise \end{cases}$$

In other words, if there exists an input vector then $G2 = 1$ and recognition in *F2* is allowed. Each node in *F1* receiving a nonzero input value generates an STM pattern activity greater than zero and the node's output is an exact duplicate of input vector. Since both X_{1i} and I_i are binary, their values would be either 1 or 0, $X1 = I$, if $G1 = 1$. Each node in *F1* whose activity is beyond the threshold sends excitatory outputs to the *F2* nodes. The *F1* output pattern *X1* is multiplied by the LTM traces *W12* connecting from *F1* to *F2*. Each node in *F2* sums up all its LTM gated signals

$$V_{2j} = \sum_i X_{1i} W_{12ji}$$

These connections represent the input pattern classification categories, where each weight stores one category. The output X_{2j} is defined so that the element that receives the largest input should be clearly enhanced. As such, the competitive network *F2* works as a winner-take-all network described by.

$$X_{2j} = \begin{cases} 1 \; if \; G_2 = 1 \cap V_{2j} = \overset{max}{k} \{V_{2k}\} \forall k \\ 0 \qquad\qquad otherwise \end{cases}$$

The *F2* unit receiving the largest *F1* output is the one that best matches the input vector category, thus winning the competition. The *F2* winner node fires, having its value set to one, inhibiting all other nodes in the layer resulting in all other nodes being set to zero.

6.2.2 Comparison

In the comparison or top-down template matching, the STM activation pattern *X2* on *F2* generates a top-down template on *F1*. This pattern is multiplied by the LTM traces *W12* connecting from *F2* to *F1*. Each node in *F1* sums up all its LTM gated signals,

$$V_{1i} = \sum_j X_{2j} W_{21ij}$$

The most active recognition unit from *F2* passes a one back to the comparison layer *F1*. Since the recognition layer is now active, *G1* is inhibited and its output is set to zero. In accordance with the "2/3" rule, stating that from three different input sources at least two are required to be active in order to generate an excitatory output, the only comparison units that will fire are those that receive simultaneous ones from the input vector and the recognition layer. Units not receiving a top down signal from *F2* must be inactive even if they receive input from below. This is summarised as follows:

$$X_{1i} = \begin{cases} 1 \; I_i \; \cap \mathrm{V}_{1\mathrm{i}} = 1 \\ 0 \;\; otherwise \end{cases}$$

If there is a good match between the top-down template and the input vector, the system becomes stable and learning may occur. If there is a mismatch between the input vector and the activity coming from the recognition layer, this indicates that the pattern being returned is not the one desired and the recognition layer should be inhibited.

6.2.3 Search

The reset layer in the orienting subsystem measures the similarity between the input vector and the recognition layer output pattern. If a mismatch between them, the reset layer inhibits the *F2* layer activity. The orienting systems compares the input vector to the *F1* layer output and causes a reset signal if their degree of similarity is less than the vigilance level, where ρ is the vigilance parameter set as $0 < \rho \leq 1$. The input pattern mismatch occurs if the following inequality is true,

$$\rho < \frac{|X_1|}{|I|}$$

If the two patterns differ by more than the vigilance parameter, a reset signal is sent to disable the firing unit in the recognition layer *F2*. The effect of the reset is to force the output of the recognition layer back to zero, disabling it for the duration of the current classification in order to search for a better match. The parameter ρ determines how large a mismatch is tolerated.

A large vigilance parameter makes the system to search for new categories in response to small difference between *I* and *X2* learning to classify input patterns into a large number of finer categories. Having a small vigilance parameter allows for larger differences and more input patterns are classified into the same category.

When a mismatch occurs, the total inhibitory signal from *F1* to the orienting subsystem is increased. If the inhibition is sufficient, the orienting subsystem fires and sends a reset signal. The activated signal affects the *F2* nodes in a state-dependent fashion. If an *F2* node is active, the signal through a mechanism known as gated dipole field causes a long-lasting inhibition.

When the active *F2* node is suppressed, the top-down output pattern *X2* and the top-down template *V1* are removed and the former *F1* activation pattern *X1* is generated again. The newly generated pattern *X1* causes the orienting subsystem to cancel the reset signal and bottom-up activation starts again. Since *F2* nodes having fired receive the long lasting inhibition, a different *F2* unit will win in the recognition layer and a different stored pattern is fed back to the comparison layer. If the pattern once again does not match the input, the whole process gets repeated.

If no reset signal is generated this time, the match is adequate and the classification is finished. The above three stages, that is, recognition, comparison, and search, are repeated until the input pattern matches a top-down template *X1*. Otherwise a *F2* node that has not learned any patterns yet is activated. In the latter case, the chosen *F2* node becomes a learned new input pattern recognition category.

6.2.4 Learning

The above three stages take place very quickly relative to the time constants of the learning equations of the LTM traces between *F1* and *F2*. Thus, we can assume that the learning occurs only when the STM reset and search process end and all STM patterns on *F1* and *F2* are stable. The LTM traces from *F1* to *F2* follow the equation

$$\tau_1 \frac{dW_{12ij}}{dt} = \begin{cases} (1 - W_{12ij})L - W_{12ij}(|X_1|) & if V_{1i} and\ V_{1j} are\ active \\ -|X_1|W_{12ij} & if\ only\ V_{1j\ is\ active} \\ o & if\ only\ V_{1j}\ is\ inactive \end{cases}$$

where τ_1 is the time constant and *L* is a parameter with a value greater than one. Because time constant τ is sufficiently larger than the STM activation and smaller than the input pattern presentation, the above is a slow learning equation that converges in the fast learning equation

$$W_{12ij} = \begin{cases} \frac{L}{L-1+|X_1|} & if V_{1i} and\ V_{1j} are\ active \\ 0 & if\ only\ V_{1j\ is\ active} \\ no\ change & if\ only\ V_{1j}\ is\ inactive \end{cases}$$

The initial values for W_{12ij} must be randomly chosen while satisfying the inequality

$$0 < W_{12ij} \frac{L}{L-1+|M|}$$

where *M* is the input pattern dimension equal to the number of nodes in *F1*. The LTM traces from F2 to F1 follows the equation,

$$\tau_2 \frac{dW_{21ji}}{dt} = X_{2j}\ (-W_{21ji} + X_{1j})$$

where τ2 is the time constant and the equation is defined to converge during a presentation of an input pattern. Thus, the fast learning equation of the for W_{21ji} is

$$W_{21ji} = \begin{cases} 1\ if V_{1i} and\ V_{1j} are\ active \\ 0\ \ if\ only\ V_{1i}\ is\ inactive \end{cases}$$

The initial value for W_{21ji} must be randomly chosen to satisfy the inequality

$1 \geq W_{21ji}$ (0) > C

where C is decided by the slow learning equation parameters. However, all $W_{21ji}(0)$ may be set 1 in the fast learning case.

6.3 Theorems

The theorems describing ART1 behaviour are described next with proofs given in Carpenter and Grossberg. These theorems hold in the fast learning case with initial LTM traces satisfying constraints (10) and (14). If parameters are properly set, however, the following results also hold in the slow learning case.

(Theorem 1) Direct Access of Learned Patterns

If an F2 node has already learned input pattern I as its template, then input pattern I activates the F2 node at once. The theorem states that a pattern that has been perfectly memorised by an F2 node activates the node immediately.

(Theorem 2) Stable Category Learning

This theorem guarantees that the LTM traces W_{12ij} and W_{21ji} become stable after a finite number of learning trials in response to an arbitrary list of binary input patterns. The V_{1j} template corresponding to the j^{th} *F2* node remains constant after at most *M-1* times. In stable states, the LTM traces W_{12ij} become *L/(L-1+M)* if the i^{th} element of the top-down template corresponding to the j^{th} *F2* node is one. Otherwise, it is zero. The LTM traces W_{21ji} become one if the i^{th} element of the template of corresponding to the *jth F2* node is one. Otherwise, it is zero.

However, theorem 2 doesn't guarantee that a perfectly coded input pattern by an F2 node will be coded by the same F2 node after presentation. The F2 node may forget the input pattern in successive learning, though the template of the F2 node continues to be a subset of the input pattern.

(Theorem 3) Direct Access after Learning Stabilises

After learning has stabilised in response to an arbitrary list of binary input patterns, each input pattern *I* either directly activates the *F2* node which possesses the largest subset template with respect to I, or I cannot activate any *F2* node. In the latter case, *F2* contains no uncommitted nodes.

This theorem guarantees that a memorised pattern activates an *F2* node at once after learning and that all *F2* nodes have been already committed if any input patterns cannot be coded. If an input pattern list contains many different input patterns and *F2* contains fewer nodes, all input patterns cannot be coded with ρ close to 1. However, the theorem doesn't guarantee that an input pattern having activated an *F2* node during learning should have been coded. If there are many input patterns with respect to the number of *F2* nodes, input patterns which have smaller $|X1|$ tend to be coded while input patterns with larger $|X1|$ tend to be coded by their subsets or not coded at all after learning.

6.4 Model Implementation

The complete model incorporates the Attentional and Orient Subsystem into a single Art module, as shown in figure below, together with the ArtModel instantiating the Art module with the appropriate sizes for its layers.

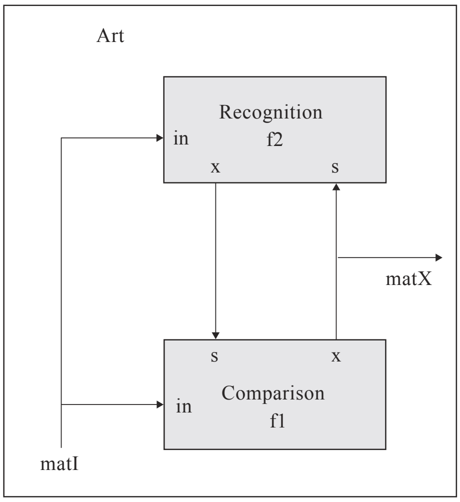


Fig. 6.4 ART module containing the F2 and F2 submodules incorporating the functionality of both the Attentional and Orienting subsystems

6.4.1 Art Module

Due to the limited process complexity of some of the components of the model only two sub-modules *F1* and *F2* are defined within the Art module. These two sub-modules correspond to layers *F1* and *F2* in the Attentional subsystem and include their respective gains. Also considering the simplicity of the orienting subsystem structures, it is incorporated directly into module *F1*.

Every simulation run initialisation, corresponding to the beginning of a new epoch, a new input pattern is sent to the *F1* and *F2* input vector ports in. Since the input ports in is a vector and *mat I* is a matrix we do a corresponding conversion between the two.

```
public void initRun() {
        matrixToVector(matI,in);
}
```

After completing a simulation run the *endRun* method is called, in this case we want to update the matX array in order to display to the user the letter output in a visually appropriate form.

```
public void endRun() {
        vectorToMatrix(x,matX);
}
```

6.4.2 Comparison Module

The Comparison module contains the corresponding data structure for the *F1* layer including gain 1. Input layer s and activity layer *x* are both initialised to 0 while weights are initialised 1.0. This is all done in the *initModule* method. The *initTrain* method resets the active elements. Simulation processing is specified in the *simTrain* method as follows

```
public void simTrain() {
if (resetActive == 1) { // input vector G1 condition, eq (7.1)

resetActive = 0;
active = -1;
x = in;
}
else { // eq (8.7)
if (s.nslMax() > 0)
s.nslMax(active);
v = w*s; // eq (8.6)
// this is a step function: x=nslStep(in+s,1.99)
for (int i = 0;i < in.getSize();i++) {
if (in[i] + v[i] >= 1.99)
x[i] = 1.0;
else
x[i] = 0.0;
}
}
}
```

This module executes the bottom-up activation, the top-down template matching, and the STM reset and search. The activation cycle is repeated until matching is complete. After running a complete simulation for a single pattern the *endTrain* method gets called. The module changes the LTM traces *F12* and *F21* after the system reaches stable responding to an input pattern. This modifies bottom-up and top-down traces *F12* and *F21* by the fast learning equations. The LTM learning module may be turned off when learning is unnecessary.

```
public void endTrain() {
s.nslMax(active); // eq. (8.9)
for (int i = 0;i < w.getRows();i++) {
if (x[i] == 1.0)
w[i][active] = 1.0;
else
w[i][active] = 0.0;
}
}
```

6.4.3 Recognition Module

The Recognition module contains the corresponding data structures for the *F2* layer. Simulation variables are initialised in the *initModule* method as follows:

```
public void initModule() {
// initialisation of all LTM weights // eq (8.10)
float max_value = l.getData()/(l.getData() - 1.0 +
in.getSize());
for (int xi = 0; xi < w.getRows(); xi++) {
for (int yi = 0; yi < w.getCols(); yi++) {
w[xi][yi] = uniformRandom(float(0.0),max_value);
}
}
}
```

The initTrain method resets the active elements. Simulation processing is specified in the simTrain method where LTM traces are multiplied to the input from *F1* and *F* activation *x* is computed. The *F2* unit that receives the biggest input from *F1* that has not been reset is activated while the other units are deactivated.

```
public void simTrain() {
if (s.nslSum() / in.nslSum() < rho.getData()) { // eq (8.8)
resetY[active] = -1.0;
active = -1;
}
if (active >= 0) {
nslPrintln(“Matching is passed”);
system.breakCycle();
return;
}
v = w*s; // eq (8.6)
num_type maxvalue;
int i;
active = -1;
x = 0.0;
float BIG_MINUS = -1.0; // the smallest value in this
program
// To exclude units which have been already reset
for (i = 0;i < resetY.getSize();i++) {
if (resetY[i] == -1.0) {
v[i] = BIG_MINUS;
}
}
// search for the unit which receives maximum input
maxvalue = v.nslMax();
// In the case that there is no available unit
if (maxvalue == BIG_MINUS) {
active = -1;
nslPrint(“An error has occured”);
system.breakCycle();
return;
}

// To find the maximum input // eq (8.5)
for (i = 0;i < v.getSize();i++) {
if (v[i] == maxvalue) {
```

```
x[i] = 1.0;
active = i;
break;
}
}
// For the error
if (i >= v.getSize()) {
nslPrintln(“An error has occured”);
system.breakCycle();
return;
}
if (active < 0) {
nslPrintln(“There are no available units”);
system.breakCycle();
return;
}
}
```

After running a complete simulation for a single pattern the *endTrain* method gets called.

```
public void endTrain() {
nslPrintln(“Top-Down Template Unit:” ,active);
if (active < 0) {
nslPrintln(“There are no units for this input”);
system.breakCycle();
return;
}
float val = l.getData() / (l.getData() - 1.0 + s.sum()); //
eq (8.11)
for (int i = 0; i < w.getCols(); i++) {
if (s[i] == 1.0)
w[active][i] = val;
else
w[active][i] = 0.0;
}
}
```

6.5 Simulation and Results

The ART1 model simulation will be illustrated with character recognition example. The NSLS command file ART1.nsls contains NSL command to set parameters and prepare graphics. The parameters to be set are only the vigilance parameter and the weight initialisation parameter besides the usual simulation steps specification.

```
nsl set art.f2.rho 0.7
nsl set art.f2.l 2.0
```

The system may run without learning by setting the epoch steps to 0. A window frame with two windows inside corresponding to the input vector and *F1* activation pattern *X*, both shown as a square pattern, are opened in the simulation. A second frame with a single window shows the *F2* activation pattern *X*. The latter layer is shown as a vector representing a group of classified categories.

Execution

A typical ART1 simulation session is as follows:

- Loading ArtModel.nsl: "nsl source artModel.nsl."
- Initialisation: Execute the NSL command "nsl init." This initialises LTM traces and variables.
- Setting character: Characters may be interactively fed by the user or read from a script file. For example read the "nsl source patI1.nsl" file for a single letter.
- Activation and Learning: Type "nsl train" to train a single cycle of the Art model. After either the maximum numbers of simulation steps are executed or X2 stabilises, *endTrain* is executed. Learning may be disabled, only by setting the epoch step number to 1.

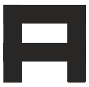
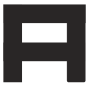
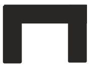
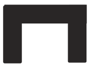
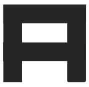
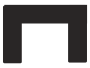
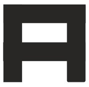
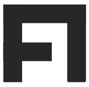
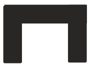
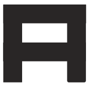
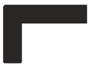
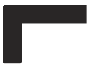
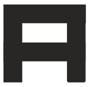
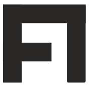
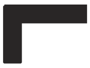
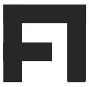
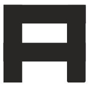
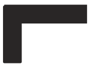
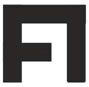
Input	I_1	I_2	I_3	I_4	Active	V_1	V_2
1					V_1		
2					V_1		
3					V_2		
4					V_1		
5					V_1		
6					V_2		
7					V_2		

Fig. 6.5 Four two-dimensional 5 by 5 (I_1, I_2, I_3 *and* I_4) patterns are presented to the ART1 system. The correct output is specified by the active V element

Output

We give a simple simulation example in this section. Four input patterns are presented to the model for a total of seven times. The input patterns, the *F2* nodes activated by them, and top-down template of the activated *F2* nodes are shown in figure above.

- An input pattern I_1 is given in the first presentation. Because no patterns have been memorised yet, the input pattern is completely learned by an *F2* node n_1 and the top-down template of n_1 is I_1 after learning.
- An input pattern I_2 is then given. Because *I2* is a subset of I_1, I_2 directly activates the same *F2* node n_1, and I_2 becomes a new template of n_1.
- The input pattern I_1 is presented again in the third trial. The F_2 node n_1 is activated at first, but it is reset because its template pattern I_2 and the input pattern I_1 are very different. Thus, another F_2 node n_2 is activated and I_1 becomes its template.
- An input pattern I_3 is given in the fourth presentation. Though I_3 looks closer to I_1 than I_2, I_3 directly accesses n_1 and the activated pattern on F_1 is I_2. The top-down template of n1 doesn't change and it is still I_2.

- The next input pattern I_4 activates n_1 because I_4 is a subset of the current template I_2 of n_1. Then, the template of n_1 becomes I_4 instead of I_2.
- Next, the input pattern I_3 is given again. It activates n_1 at first, but it is reset because its current template I_4 and I_3 are very different. Thus, I_3 activates the *F2* node n_2 at the second search, and it becomes the template of the node.
- Finally, the input pattern I_1 is given again. It directly activates the *F2* node n_2 and the activated pattern on F_1 is I_3.

The NSL simulation displays for the V elements are shown in figure below.

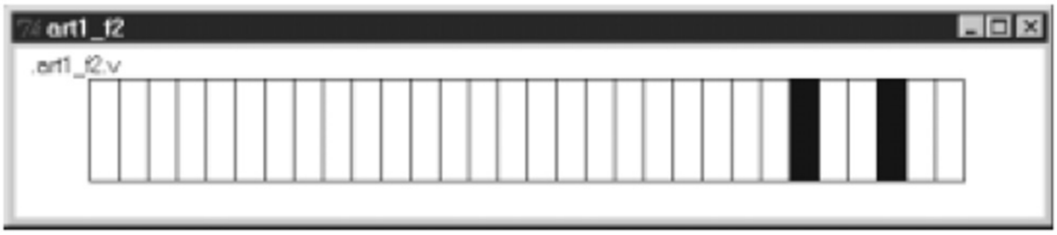


Fig. 6.6 V elements in the recognition module of the ART1 system

The NSL simulation displays comparing the letter input to the corresponding output is shown in figure below. The above example illustrates some of the features of the model:

- An *F2* node that memorises an input pattern will not necessarily keep memorising it. Though the *F2* node n1 first memorises the input pattern I_1 in the above simulation, for example, the node doesn't respond to I_1 in the final presentation. This means that the final stable state of the model may be largely different from early stages.
- Simpler patterns which have smaller |I|'s tend to be learned. Thus, when the number of the *F2* nodes are limited, complex patterns may not be learned. Skilled adjustment of a vigilance parameter is indispensable for balanced learning.
- The criterion to classify input patterns is not intuitive. For example, the input pattern I_3 is judged closer to I_2 than I_1.
- The previous top-down template n presented as an input pattern is not necessarily the final activation pattern on *F1*. This means that the model cannot restore pixels erased by noise though it can remove pixels added by noise.
- These features may be flaws of the model, but they can be taken also as good points.

Though we chose a simplified way to simulate ART1 on NSL, some interesting features of ART1 have been made clear. Different extensions can be made to the NSL implementation of ART1:

- The first extension would be a full implementation of ART1 original dynamic equations, in particular the inclusion of membrane potential equations of *F1* and *F2* nodes and the slow learning equations.
- The second extension would be to improve ART1. Some features present in our simulation are not desirable for many applications. We believe some improvements of the learning equations and matching rules would extend to further applications while keeping the basic structure of ART.

The third extension would be the implementation of other ART models. ART is a theory applying to many models, such as ART2, FUZZY-ART, besides various practical applications. A good exercise here would be to use the Maximum Selector model instead of the simple WTA used in ART.

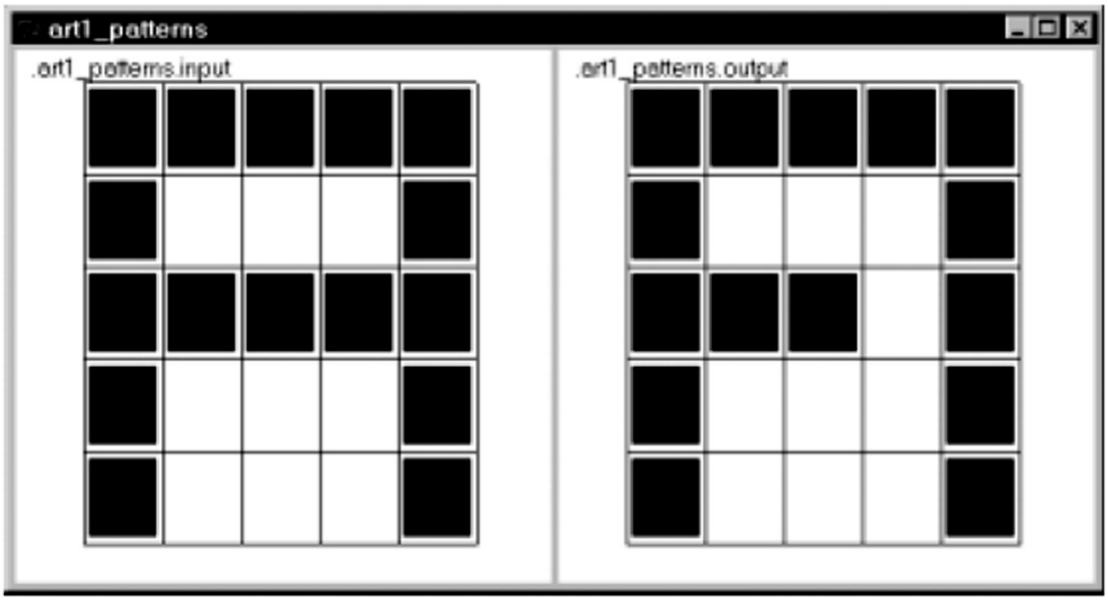


Fig. 6.7 Sample letter input and output in the ART1 system

Summary

- The adaptive resonance theory (ART) has been developed to avoid the stability-plasticity dilemma in competitive networks learning.
- The stability-plasticity dilemma addresses how a learning system can preserve its previously learned knowledge while keeping its ability to learn new patterns.
- An ART system consists of two subsystems, an attentional subsystem and an orienting subsystem.
- There are two models of ART1, a slow-learning and a fast-learning one.
- ART1 is the simplest ART learning model specifically designed for recognising binary patterns.
- If there is a good match between the top-down template and the input vector, the system becomes stable and learning may occur.
- The reset layer in the orienting subsystem measures the similarity between the input vector and the recognition layer output pattern.
- ART architecture models can self-organise in real time producing stable recognition while getting input patterns beyond those originally stored.
- ART is a family of different neural architectures. The first and most basic architecture is ART1.
- ART2 is a class of architectures categorising arbitrary sequences of analog input patterns.
- The attentional subsystem is based on competitive learning enhancing pattern features but suppressing noise.
- The system can search memory in parallel and adaptively change its search order.
- The attentional subsystem has two short term memory (STM) stages.
- The attentional subsystem consists of two competitive networks, the comparison layer *F1* and the recognition layer *F2*, and two control gains, Gain 1 and Gain 2.
- The parameter ρ determines how large a mismatch is tolerated.

References

- Serrano-Gotarredona, T., Linares-Barranco, B. & Andreou, G. A., 1998. *Adaptive Resonance Theory Microchips*, Springer.
- Braspenning, J. P., Thuijsman F. & Weijters, A. J. M. M., 1995. *Artificial Neural Networks: An Introduction To Ann Theory And Practice*, Springer.
- Tanaka, T. & Weitzenfeld1, A., 8 *Adaptive Resonance Theory*, [Online] Available at: <http://www.ica.luz.ve/dfinol/NeuroCienciaCognitiva/NeuralNetworkModels/ART%20INTRO%20-%20157_170.CH08.pdf> [Accessed 18 July 2012].
- Zacharie, M., *Adaptive Resonance Theory 1 (ART1) Neural Network Based Horizontal and Vertical Classification of 0-9 Digits Recognition*, [Online] Available at: <citeseerx.ist.psu.edu/viewdoc/download?doi=10.1.1.153...> [Accessed 18 July 2012].
- 2011. *Lecture 05, part 1 | Pattern Recognition*, [Video Online] Available at: <http://www.youtube.com/watch?v=ZeiJmSHx6qI> [Accessed 18 July 2012].
- Prof. Sarkar, S. & Prof. Basu, A., 2008. *Lecture - 30 Fuzzy Reasoning – I*, [Video Online] Available at: <http://www.youtube.com/watch?v=p-9G7mEF9D4> [Accessed 18 July 2012].

Recommended Reading

- Carpenter, A. G.& Grossberg, S., *Adaptive Resonance Theory*, Boston University.
- Saddington, P., 2002. *Adaptive Resonance Theory: Theory and Application to Synthetic Aperture Radar*, University of Surrey.
- Kussul, E., Baidyk, T. & Wunsch, C. D., 2009. *Neural Networks and Micromechanics*, Springer.

Self Assessment

1. ______ is the simplest ART learning model specifically designed for recognising binary patterns.
   a. ART1
   b. ART2
   c. ART3
   d. ART4

2. The ART1 system consists of an _________ subsystem and an _________ subsystem.
   a. model, descriptive
   b. ART1, ART2
   c. attentional, orienting
   d. F1, F2

3. Which of the following statements is false?
   a. The adaptive resonance theory (ART) has been developed to avoid the stability-plasticity dilemma in competitive networks learning.
   b. ART is a family of different neural architectures.
   c. ART1 cannot learn and recognise binary patterns.
   d. ART2 is a class of architectures categorising arbitrary sequences of analog input patterns.

4. ART2 is a class of architectures categorising arbitrary sequences of ______ input patterns.
   a. digital
   b. analog
   c. catalog
   d. electronic

5. The stabilisation of learning and activation occurs in the _________ subsystem.
   a. orienting
   b. attentional
   c. electronic
   d. digital

6. Which of the following statements is false?
   a. An ART system has four basic properties.
   b. There are two models of ART1, a slow-learning and a fast-learning one.
   c. The ART1 system consists of an attentional subsystem and an orienting subsystem.
   d. The attentional subsystem has four short term memory stages.

7. Which of the following is not an ART1 processing phase?
   a. attention
   b. comparison
   c. search
   d. learning

8. The reset layer in the ______ subsystem measures the similarity between the input vector and the recognition layer output pattern.
   a. orienting
   b. attentional
   c. ART1
   d. ART2

9. The _____ method resets the active elements.
   a. alert
   b. init
   c. reset
   d. update

10. The LTM learning module may be turned off when learning is ________ .
    a. necessary
    b. at peak point
    c. producing output
    d. unnecessary

Chapter VII

Fuzzy Sets and Fuzzy Logic

Aim

The aim of this chapter is to:

- introduce fuzzy sets and fuzzy logic
- elucidate imprecise data and imprecise rules
- explain function approximation with fuzzy methods

Objectives

The objectives of this chapter are to:

- explain logic operators and geometry
- explicate geometric representation of fuzzy sets
- elucidate the eye as a fuzzy system

Learning outcome

At the end of this chapter, you will be able to:

- understand the fuzzy set concept
- identify the function approximation with fuzzy methods
- describe control with fuzzy logic

7.1 Introduction

We know that the learning problem is NP-complete for a broad class of neural networks. Learning algorithms may require an exponential number of iterations with respect to the number of weights until a solution to a learning task is found. A second important point is that in back propagation networks, the individual units perform computations more general than simple threshold logic. Since the output of the units is not limited to the values 0 and 1, giving an interpretation of the computation performed by the network is not so easy. The network acts like a black box by computing a statistically sound approximation to a function known only from a training set. In many applications an interpretation of the output is necessary or desirable. In all such cases the methods of fuzzy logic can be used.

7.2 Imprecise Data and Imprecise Rules

Fuzzy logic can be conceptualised as a generalisation of classical logic. Modern fuzzy logic was developed by Lotfi Zadeh in the mid-1960s to model those problems in which imprecise data must be used or in which the rules of inference are formulated in a very general way making use of diffuse categories. In fuzzy logic, which is also sometimes called diffuse logic, there are not just two alternatives but a whole continuum of truth values for logical propositions.

A proposition A can have the truth value 0.4 and its complement Ac the truth value 0.5. According to the type of negation operator that is used, the two truth values must not be necessarily add up to 1. Fuzzy logic has a weak connection to probability theory. Probabilistic methods that deal with imprecise knowledge are formulated in the Bayesian framework, but fuzzy logic does not need to be justified using a probabilistic approach. The common route is to generalise the findings of multi-valued logic in such a way as to preserve part of the algebraic structure.

In this chapter we will show that there is a strong link between set theory, logic, and geometry. A fuzzy set theory corresponds to fuzzy logic and the semantic of fuzzy operators can be understood using a geometric model. The geometric visualisation of fuzzy logic will give us a hint as to the possible connection with neural networks. Fuzzy logic can be used as an interpretation model for the properties of neural networks, as well as for giving a more precise description of their performance.

We will show that fuzzy operators can be conceived as generalised output functions of computing units. Fuzzy logic can also be used to specify networks directly without having to apply a learning algorithm. An expert in a certain field can sometimes produce a simple set of control rules for a dynamical system with less effort than the work involved in training a neural network. A classical example proposed by Zadeh to the neural network community is developing a system to park a car. It is straightforward to formulate a set of fuzzy rules for this task, but it is not immediately obvious how to build a network to do the same n or how to train it.

Fuzzy logic is now being used in many products of industrial and consumer electronics for which a good control system is sufficient and where the question of optimal control does not necessarily arise.

7.3 The Fuzzy Set Concept

The difference between crisp that is classical and fuzzy sets is established by introducing a membership function. Consider a finite set $X = \{x_1, x_2, \dots, x_n\}$ which will be considered the universal set in what follows. The subset A of X consisting of the single element x_1 can be described by the n-dimensional membership vector $Z(A) = (1, 0, 0, \dots, 0)$, where the convention has been adopted that a 1 at the i-th position indicates that *xi* belongs to A. The set B composed of the elements x_1 and x_n is described by the vector Z(B) = (1, 0, 0, ..., 1). Any other crisp subset of X can be represented in the same way by an n-dimensional binary vector. But what happens if we lift the restriction to binary vectors? In that case we can define the fuzzy set C with the following vector description: $Z(C) = (0.5, 0, 0, ..., 0)$

In classical set theory such a set cannot be defined. An element belongs to a subset or it does not. In the theory of fuzzy sets we make a generalisation and allow descriptions of this type. In our example the element x_1 belongs to the set C only to some extent. The degree of membership is expressed by a real number in the interval [0, 1], in this case 0.5. This interpretation of the degree of membership is similar to the meaning we assign to statements such as “person x_1 is an adult”. Obviously, it is not possible to define a definite age which represents the absolute threshold to enter into adulthood. The act of becoming mature can be interpreted as a continuous process in which the membership of a person to the set of adults goes slowly from 0 to 1.

There are many other examples of such diffuse statements. The concepts “old” and “young” or the adjectives “fast” and “slow” are imprecise but easy to interpret in a given context. In some applications, such as expert systems, for example, it is necessary to introduce formal methods capable of dealing with such expressions so that a computer using rigid Boolean logic can still process them. This is what the theory of fuzzy sets and fuzzy logic tries to accomplish.

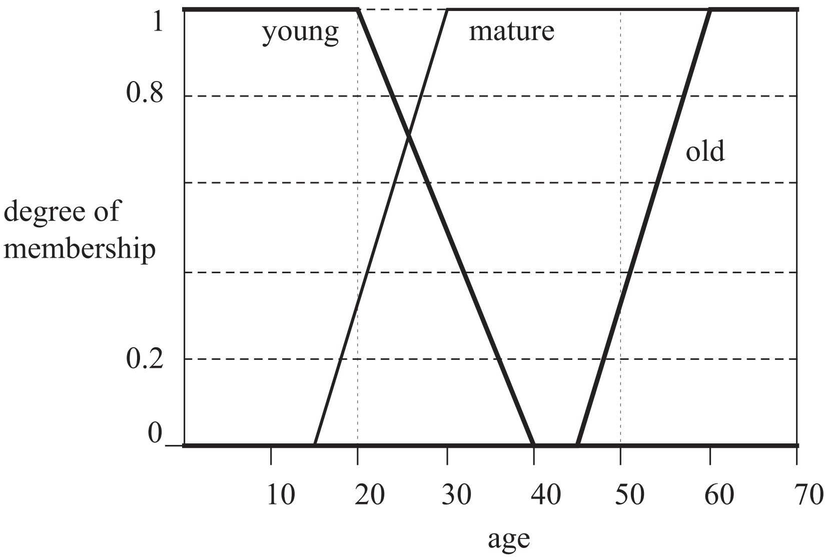


Fig. 7.1 Membership functions for the concepts young, mature and old

The above figure shows three examples of a membership function in the interval 0 to 70 years. The three functions define the degree of membership of any given age in the sets of young, adult, and old ages. If someone is 20 years old, for example, his degree of membership in the set of young persons is 1.0, in the set of adults 0.35, and in the set of old persons 0.0. If someone is 50 years old the degrees of membership are 0.0, 1.0, 0.3 in the respective sets.

Definition: Let X be a classical universal set. A real function $\mu A : X \rightarrow [0, 1]$ is called the membership function of A and defines the fuzzy set A of X. This is the set of all pairs $(x, \mu A(x))$ with $x \in X$.

A fuzzy set is completely determined by its membership function. Note that the above definition also covers the case in which X is not a finite set. The set of support of a fuzzy set A is the set of all elements x of X for which $(x, \mu_A(x)) \in A$ and $\mu_A(x) > 0$ holds. A fuzzy set A with the finite set of support $\{a_1, a_2, \ldots, a_m\}$ can be described in the following way $A = \mu_1/a_1 + \mu_2/a_2 + \cdots + \mu_m/a_m$, where $\mu_i = \mu_A(a_i)$ for $i = 1, \ldots, m$. The symbols “/” and “+” are used only as syntactical constructors.

Crisp sets are a special case of fuzzy sets, since the range of the function is restricted to the values 0 and 1. Operations defined over crisp sets, such as union or intersection, can be generalised to cover also fuzzy sets. Assume as an example that $X = \{x_1, x_2, x_3\}$. The classical subsets $A = \{x_1, x_2\}$ and $B = \{x_2, x_3\}$ can be represented as $A = 1/x_1 + 1/x_2 + 0/x_3$ $B = 0/x_1 + 1/x_2 + 1/x_3$.

The union of A and B is computed by taking for each element xi the maximum of its membership in both sets, that is: $A \cup B = 1/x_1 + 1/x_2 + 1/x_3$ The fuzzy union of two fuzzy sets can be computed in the same way. The union of the two fuzzy sets C = $0.5/x1 + 0.6/x_2 + 0.3/x_3$ $D = 0.7/x_1 + 0.2/x_2 + 0.8/x_3$ is given by,

$$\mathrm{C} \cup \mathrm{D} = 0.7/x_1 + 0.6/x_2 + 0.8/x_3$$

The fuzzy intersection of two sets *A* and *B* can be defined in a similar way, but instead of taking the maximum we compute the minimum of the membership of each element xi to *A* and *B*. The maximum or minimum of the membership values are just one pair of possible definitions of the union and intersection operations for fuzzy sets. As we show later on, there are other alternative definitions.

7.4 Geometric Representation of Fuzzy Sets

Bart Kosko introduced a very useful graphical representation of fuzzy sets. Figure below shows an example in which the universal set consists only of the two elements x_1 and x_2. Each point in the interior of the unit square represents a subset of *X*. The convention is that the coordinates of the representation correspond to the membership values of the elements in the fuzzy set. The point (1, 1), for example, represents the universal set *X*, with membership function $\mu_A(x_1) = 1$ and $\mu_A(x_2) = 1$. The point (1, 0) represents the set $\{x_1\}$ and the point (0, 1) the set $\{x_2\}$.

The crisp subsets of X are located at the vertices of the unit square. The geometric visualisation can be extended to an n-dimensional hypercube. Kosko calls the inner region of a unit hypercube in an n-dimensional space the fuzzy region. We find here all combinations of membership values that a fuzzy set could assume. The point M in figure below corresponds to the fuzzy set $M = 0.5/x_1+0.3/x_2$. The center of the square represents the most diffuse of all possible fuzzy sets of *X*, that is the set $Y = 0.5/x_1 + 0.5/x_2$. The degree of fuzziness of a fuzzy set can be measured by its entropy. In the geometric visualisation, this corresponds inversely to the distance between the representation of the set and the center of the unit square.

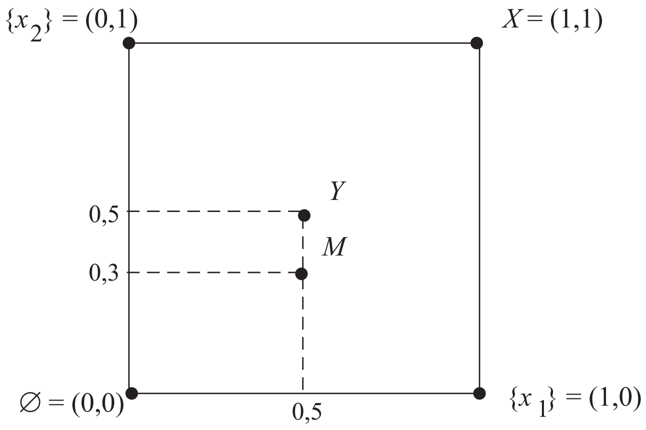


Fig. 7.2 Geometric visualisation of fuzzy sets

The set Y in figure below has the maximum possible entropy. The vertices represent the crisp sets and have the lowest entropy, that is, zero. Note that the fuzzy concept of entropy is mathematically different from the entropy concept in physics or information theory. Some authors prefer to use terms like index of fuzziness or also crispness, certitude, ambiguity, etc. With this caveat we adopt a preliminary definition of the entropy of a fuzzy set M as the quotient of the distance d1 (according to some metric) of the corner which is nearest to the representation of M to the distance d2 from the corner which is farthest away. Figure below shows the two relevant segments. The entropy E (M) of M is therefore

$$E\,(M) = \frac{d_1}{d_2}$$

According to this definition the entropy is bounded by 0 and 1. The maximum entropy is reached at the center of the square. The union or intersection of sets can be also visualised using this representation. The membership function for the union of two sets A and B can be defined as

$$\mu A \cup B(x) = \max(\mu A(x), \mu B(x)) \; \forall x \in X \qquad (7.1)$$

and corresponds to the maximum of the corresponding coordinates in the geometric visualisation. The membership function for the intersection of two sets A and B is given by

$\mu A \cap B(x) = \min(\mu A(x), \mu B(x)) \; \forall x \in X$ (7.2)

Together with the points representing the sets A and B. Figure Intersection and union of two fuzzy sets shows the points which represent their union and intersection.

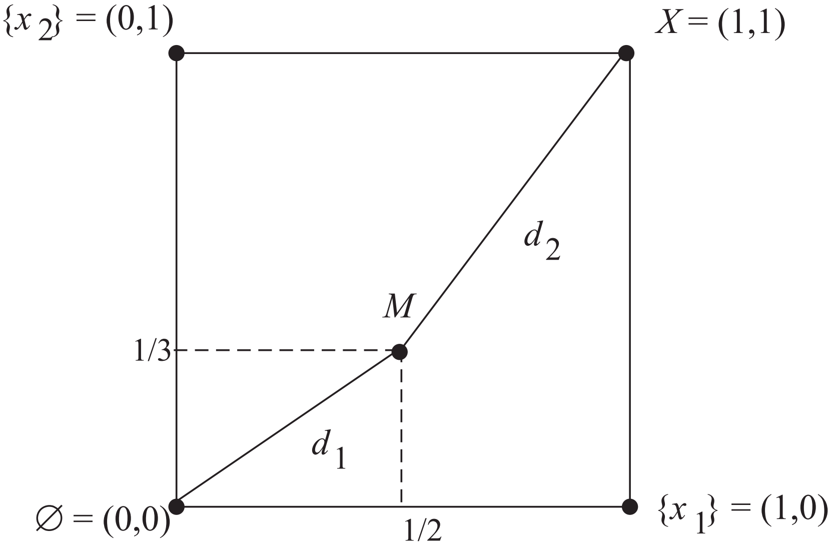


Fig. 7.3 Distance of the set M to the universal and to the void set

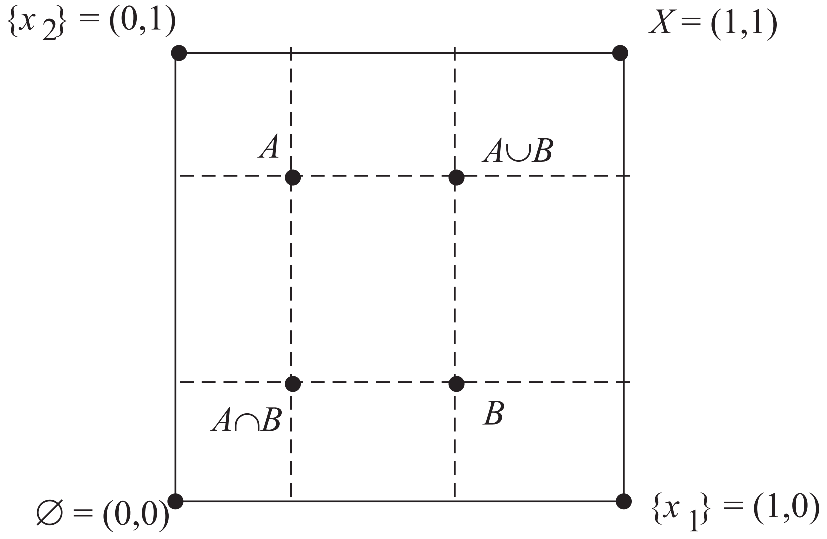


Fig. 7.4 Intersection and union of two fuzzy sets

The union or intersection of two fuzzy sets is in general a fuzzy, not a crisp set. The complement A^c of a fuzzy set A can be defined with the help of the membership function μA^c given by

$\mu_A{}^c(x) = 1 - \mu_A(x) \; \forall x \in X$ (7.3)

Figure below shows that the representation of *A* must be transformed into another point at the same distance from the center of the unit square. The line joining the representation of *A* and *Ac* goes through the center of the square. Figure below also shows how to obtain the representations for $A \cup A^c$ and $A \cap A^c$ using the union and intersection operators defined before.

For fuzzy sets, it holds in general that $A \cup A^c = X$ and $A \cap A^c = \emptyset$ which is not true in classical set theory. This means that the principle of excluded middle and absence of contradiction do not necessarily hold in fuzzy logic. The consequences will be discussed later.

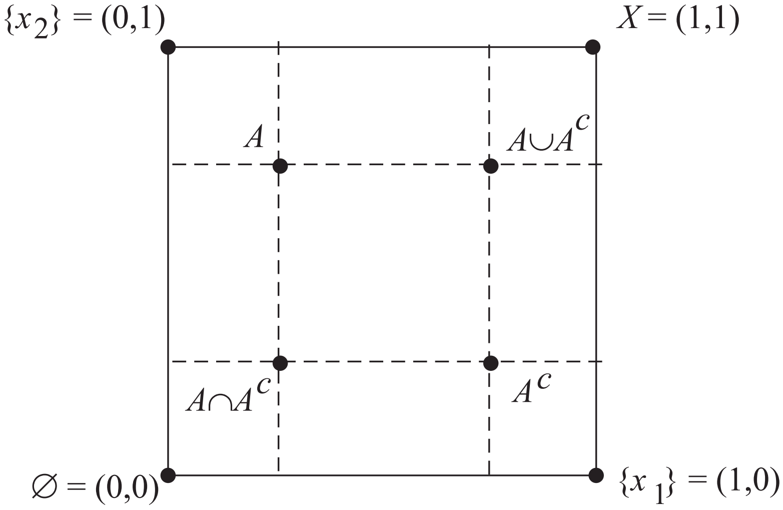


Fig. 7.5 Complement Ac of a fuzzy set

Kosko establishes a direct relationship between the entropy of fuzzy sets and the geometric visualisation of the union and intersection operations. To compute the entropy of a set, we need to determine the distance between the origin and the coordinates of the set. This distance is called the cardinality of the fuzzy set.

Definition: Let A be a subset of a universal set X. The cardinality $|A|$ of A is the sum of the membership values of all elements of X with respect to A, i.e.,

$$|A| = \sum_{x \in X} \mu A(x)$$

This definition of cardinality corresponds to the distance of the representation of A from the origin using a Manhattan metric. Figure below shows how to define the entropy of a set A using the cardinality of the sets $A \cap A^c$ and $A \cup A^c$. The Manhattan distances d_1 and d_2, introduced before to measure the fuzzy entropy, correspond to the cardinalities of the sets $A \cap A^c$ and $A \cup A^c$. The entropy concept introduced previously in an informal manner can then be formalised with our next definition.

Definition: The real value

$$E(A) = \frac{|A \cap A^c|}{|A \cup A^c|}$$

is called the entropy of the fuzzy set A.

The entropy of a crisp set is always zero, since for a crisp set $A \cap A^c = \emptyset$. In fuzzy set theory $E(A)$ is a value in the interval [0, 1], since $A \cap A^c$ can be non-void. Some authors take the geometric definition of entropy as given and derive. Here we take the definition as given, since the geometric interpretation of fuzzy union and intersection depends on the exact definition of the fuzzy operators.

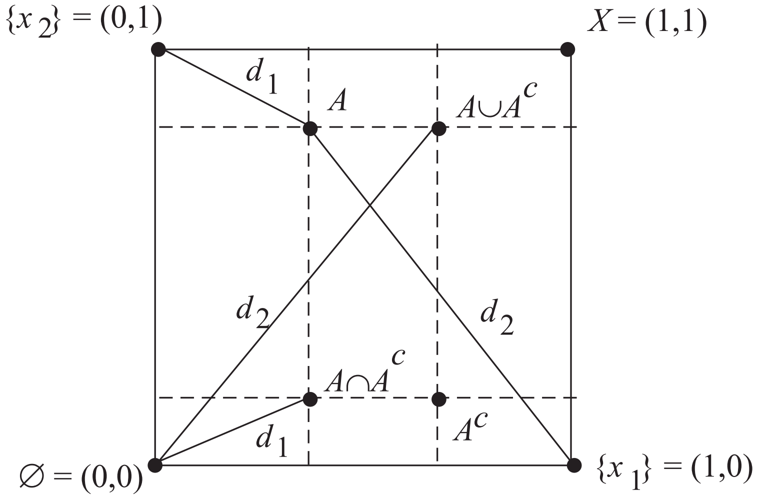


Fig. 7.6 Geometric representation of the entropy of a fuzzy set

7.5 Fuzzy Set Theory, Logic Operators, and Geometry

It is known in mathematics that an isomorphism exists between set theory and classic propositional logic. In set theory, the three operators union, intersection, and complement ($\cup$, $\cap$, c) allow the construction of new sets from other sets. In propositional logic, the operators OR, AND and NOT ($\vee$, $\wedge$, $\neg$) are used to build new propositions.

The union operator of classical set theory can be constructed using the OR operator. Let A and B be two crisp sets, that is, $\mu_A, \mu_B : X \rightarrow \{0, 1\}$. The membership function $\mu_a \cup B$ of the union set $A \cup B$ is

$$\mu A \cup B(x) = \mu A(x) \vee \mu B(x)\ \forall x \in X \qquad (7.4)$$

where the value 0 is interpreted as the logic value false and 1 as true. In a similar way it holds that for the intersection of the sets A and B

$$\mu A \cap B(x) = \mu A(x) \wedge \mu B(x)\ \forall x \in X \qquad (7.5)$$

For the complement A^c of the set A it holds that

$$\mu A_c(x) = \neg \mu A(x) \qquad (7.6)$$

This correspondence between the operators of classical set theory and classical logic can be extended to the validity of some equations in both systems. The laws of de Morgan, for example, are valid in classical set theory as well as in classical logic:

$(A \cup B)^c \equiv A^c \cap B^c$ corresponds to $\neg(A \vee B) \equiv \neg A \wedge \neg B$
$(A \cap B)^c \equiv A^c \cup B^c$ corresponds to $\neg(A \wedge B) \equiv \neg A \vee \neg B$

A fuzzy logic can be also derived from fuzzy set theory by respecting the isomorphism mentioned above. The fuzzy AND, OR, and NOT operators must be defined in such a way that the same general kinds of relation exist between them and their equivalents in classical set theory and logic.

The straightforward approach is therefore to identify the OR operation $(\tilde{V})$ with the maximum function, AND $(\tilde{V})$ with the minimum, and complementation $(\tilde{\neg})$ with the function $x \rightarrow 1 - x$. Equations (7.1), (7.2), and (7.3) can be written as

$$\mu_{A \cup B}(x) = \mu A(x)\ \tilde{V}\ \mu B(x)\ \forall_x \in X \qquad (7.7)$$

$\mu_{A\cup B}(x) = \mu A(x) \, \tilde{\vee} \, \mu B(x) \; \forall x \in X$ (7.8)

$\mu_{A}c(x) = \tilde{\neg}\, \mu_{A}(x) \; \forall x \in X$ (7.9)

In this way an isomorphism between fuzzy set theory and fuzzy logic is constructed which preserves the properties of the isomorphism present in the classical theories. Many rules of classical logic are still valid in the world of fuzzy operators. For example, the functions min and max are commutative and associative. However, the principle of no contradiction has been abolished. For a proposition A with truth value 0.4 we get

$$\mathrm{A} \,\tilde{\wedge}\, \tilde{\neg}\,\mathrm{A} = \min(0.4,\ 1 - 0.4) \neq 0$$

The principle of excluded middle is not valid for A either:

$$\mathrm{A} \,\tilde{\vee}\, \tilde{\neg}\,\mathrm{A} = \max(0.4,\ 1 - 0.4) \neq 1$$

7.6 Families of Fuzzy Operators

Up to this point we have worked with fuzzy operators defined in a rather informal way, since there are whole families of these operators that can be defined. Now we will give an axiomatic definition using the properties we would like the operators to exhibit. Consider the fuzzy OR operator. In the fuzzy logic literature such an operator is required to fulfil the following axioms:

Axiom U1: Boundary conditions:
$0 \,\tilde{\vee}\, 0 = 0$
$1 \,\tilde{\vee}\, 0 = 1$
$0 \,\tilde{\vee}\, 1 = 1$
$1 \,\tilde{\vee}\, 1 = 1$

Axiom U2: Commutativity:
$a \,\tilde{\vee}\, b = b \,\tilde{\vee}\, a$

Axiom U3: Monotonicity:
If $a \leq a'$ and $b \leq b'$, then $a \,\tilde{\vee}\, b \leq a' \,\tilde{\vee}\, b'$

Axiom U4: Associativity:
$a \,\tilde{\vee}\, (b \,\tilde{\vee}\, c) = (a \,\tilde{\vee}\, b) \,\tilde{\vee}\, c$

It is easy to show that the maximum function fulfils these four conditions. There are many other functions for which the four axioms are valid, for example

$B(a, b) = \min(1, a + b)$

Which is called the bounded sum of a and b. An alternative fuzzy OR operator can be defined using this function. However, the bounded sum is not idempotent, that is, in general $B(a, a) \neq a$. We can therefore introduce a fifth axiom to exclude such operators:

Axiom U5: Idempotence
$a \,\tilde{\vee}\, a = a$

Depending on the axioms selected, different fuzzy operators and different logic systems can be defined. Consequently, the term fuzzy logic refers to a family of different theories and not to a unique system of logic. In the case of the fuzzy operator $\tilde{\wedge}$ axioms are also formulated in such a way that fuzzy AND is monotonic, commutative, and associative. The boundary conditions are:

$0 \tilde{\wedge} 0 = 0$
$1 \tilde{\wedge} 0 = 0$
$0 \tilde{\wedge} 1 = 0$
$1 \tilde{\wedge} 1 = 1$

Idempotence can be demanded and can be enforced using a fifth axiom. For the fuzzy negation we use the following three axioms:

Axiom N1. Boundary conditions:
$\tilde{\neg} 1 = 0$
$\tilde{\neg} 0 = 1$

Axiom N2: Monotonicity:
If $a \leq b$ then $\tilde{\neg} b \leq \tilde{\neg} a$

Axiom N3: Involution:
$\tilde{\neg} \tilde{\neg} a = a$

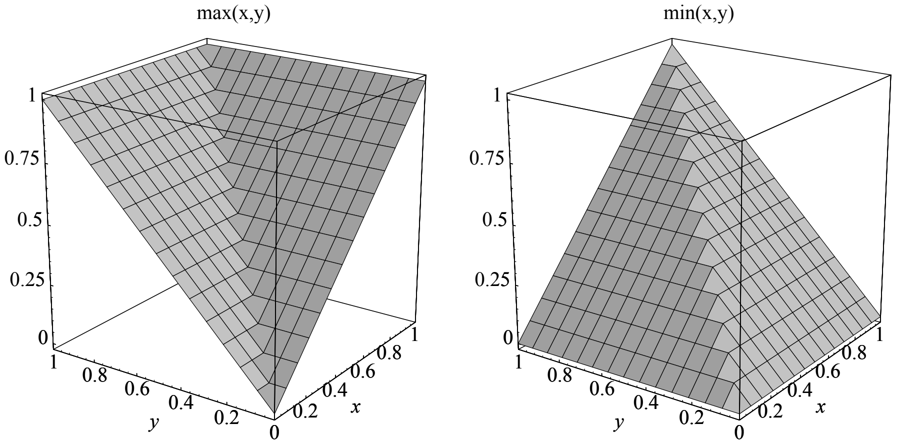


Fig. 7.7 The fuzzy operators

The difference between these fuzzy operators can be better visualised by looking at their graphs. Figure above shows the graphs of the functions max and min, that is, a possible fuzzy AND and fuzzy OR combination. They could also be used as activation functions in neural networks. Using output functions derived from fuzzy logic can have the added benefit of providing a logical interpretation of the neural output.

The graphs of the functions bounded sum and bounded difference are shown in Figure below. Both fulfil the conditions imposed by the first four axioms for fuzzy OR and fuzzy AND operators, but are not idempotent.

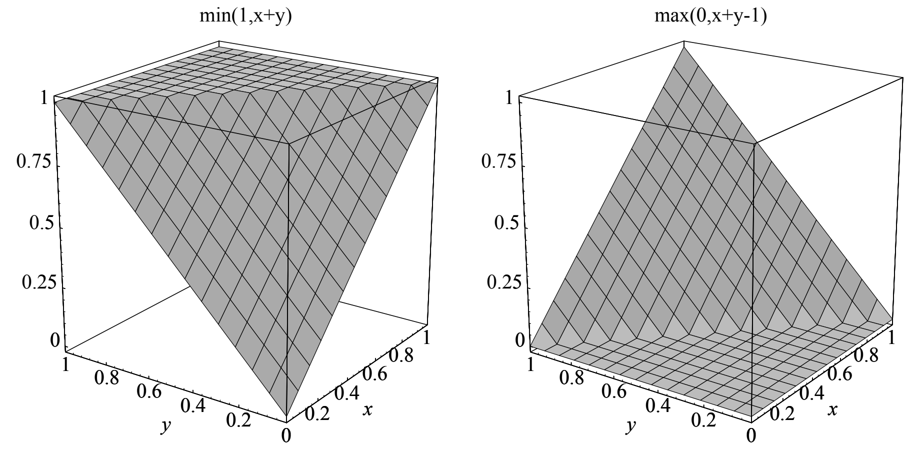


Fig. 7.8 The fuzzy operators bounded sum and bounded difference

It is also possible to define a parameterised family of fuzzy operators. Figure below illustrates this approach for the case of the so-called Yager union function which is given by

$$Y_p(a, b) = \min(1, (a^p + b^p)^{1/p}) \text{ for } p \geq 1$$

where a and b are real numbers in the interval [0, 1]. The formula describes a family of operators. For $p = 2$, the function is an approximation of the bounded sum operator. For p » 1, Y_p is an approximation of the max function. Adjusting the parameter p we can select the desired variant of fuzzy logic.

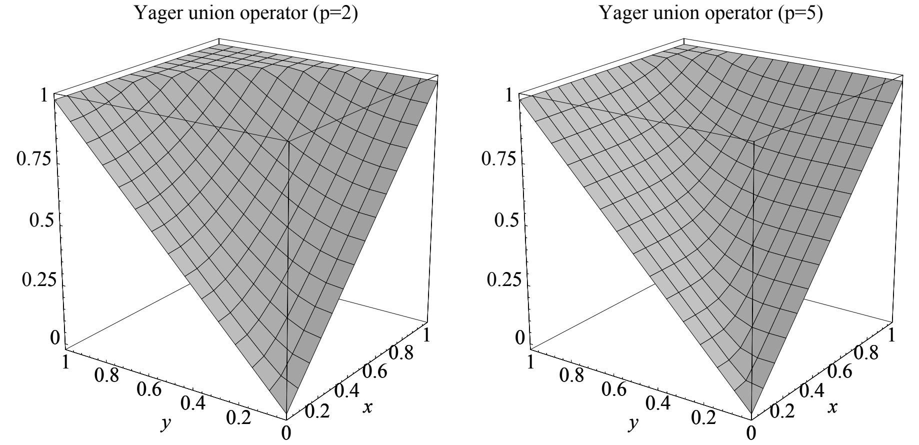


Fig. 7.9 Two variants of the Yager union operator

Figure above shows that the Yager union operator is not idempotent. If it were, the diagonal from the point (0, 0) to the point (1, 1) would belong to the graph of the function. This is the case for the functions min and max. It can be shown that these functions are the only ones in which the five axioms for fuzzy OR and fuzzy AND fully apply. The geometric properties of fuzzy operators can be derived from the axioms for fuzzy OR, fuzzy AND, and fuzzy negation operators. The boundary conditions determine four values of the function. The commutativity of the operators forces the graph of the functions to be symmetrical with respect to the plane normal to the xy plane and which cuts it at the 45-degree diagonal.

Monotonicity of the operators allows only those function graphs that do not fold. Associativity is more difficult to visualise but it roughly indicates that the function does not grow abruptly in some regions and stagnate in others. If all or some of these symmetry properties hold for a binary function, then this function fulfils the operator axioms and can be used as a fuzzy operator. The symmetry properties, in turn, lead to useful algebraic properties of the operators, and the connection between set theory, logic, and geometry is readily perceived.

7.7 Fuzzy Inferences

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

7.7.1 Inferences from Imprecise Data

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

$$\text{R1: If } (A \,\tilde{\wedge}\, B) \text{ then } C$$
$$\text{R2: If } (A \,\tilde{\vee}\, B) \text{ then } D$$

The difference in conventional inference rules is the semantics of the fuzzy operators. In this section we identify the fuzzy operators $\tilde{\wedge}$ and $\tilde{\vee}$ with the functions min and max respectively. Let the truth values of A and B be 0.4 and 0.7 respectively. In this case

$$A \,\tilde{\wedge}\, B = \min(0.4, 0.7) = 0.4$$
$$A \,\tilde{\vee}\, B = \max(0.4, 0.7) = 0.7$$

This is interpreted by the fuzzy inference mechanism as meaning that the rules *R1* and *R2* can only be partially applied that is rule *R1* is applied to 40% and rule R2 to 70%. The result of the inference is a combination of the propositions *C* and *D*. Let us consider another example. Assume that a temperature controller must regulate an electrical heater. We can formulate three rules for such a system:

R1: If (temperature = cold) then heat
R2: If (temperature = normal) then maintain
R3: If (temperature=warm) then reduce power

Assume that a temperature of 12 degrees Celsius has a membership degree of 0.5 in relation to the set of cold temperatures and a membership degree of 0.3 in relation to the temperatures classified as normal. The temperature of 12 degrees is converted first of all into a fuzzy category which is the list of membership values of an element x of X in relation to previously selected fuzzy sets of the universal set X. The fuzzy category T can be expressed using a similar notation as for fuzzy sets. In our example:

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

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

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

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

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

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

The advantage of formulating fuzzy inference rules is their low granularity. In many cases apparently complex control problems can be modelled using just a few rules. If we tried to express all actions as consequences of exact numerical rules, we would have to write more of them or make each one much more complex.

7.7.2 Fuzzy Numbers and Inverse Operation

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

There are several alternative ways to transform a measurement into fuzzy categories. A frequent approach is to use triangular or trapezium-shaped membership functions. Fig.7.10 shows how a measurement interval can be subdivided using triangular-shaped membership functions and Fig.7.11 shows the same kind of subdivision but with trapezium-shaped membership functions.

The transformation of a measurement x into a fuzzy category is given by the membership values $\alpha_1, \alpha_2, \alpha_3$ derived from the membership functions (as shown in Figure 11.10). An important problem is how to transform the membership values $\alpha_1, \alpha_2, \alpha_3$ back into the measurement x, that is, how to implement the inverse operation to the fuzzifying of the crisp number. A popular approach is the centroid method. Fig. 7.12 shows the value x and its transformation into $\alpha_1, \alpha_2, \alpha_3$. From these three values we can reconstruct the original x. To do this, the surfaces of the triangular regions limited by the heights α_1, α_2 and α_3 are computed.

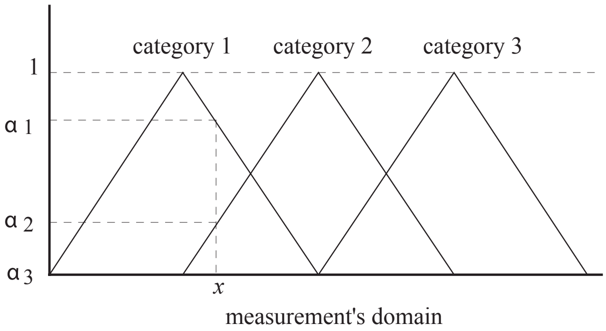


Fig. 7.10 Categories with triangular membership functions

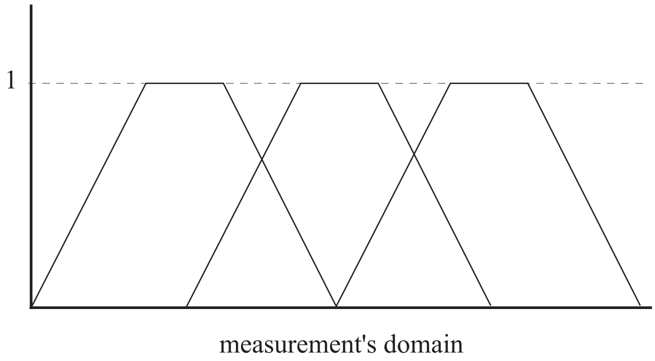


Fig. 7.11 Categories with trapezium-shaped membership functions

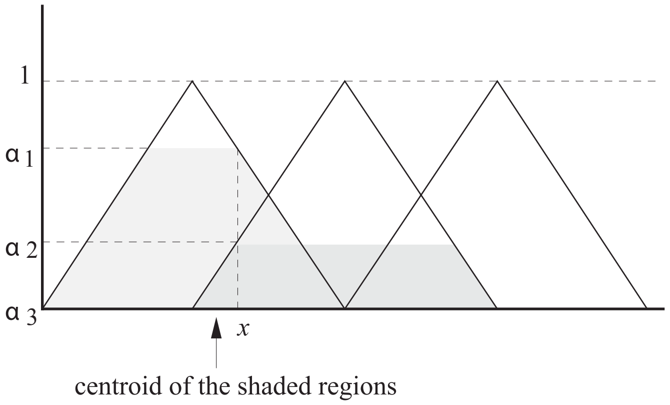


Fig. 7.12 The centroid method

The horizontal component of the centroid of the total surface is the approximation to x we are looking for. For all x values for which at least two of the three numbers α_1, α_2, α_3 are different from zero, we can compute a good approximation using the centroid method. Figure below shows the difference between x and its approximation when the basis of the triangular regions is of length 2, their height is 1 and the arrangement is the one shown in figure above. The value of x has been chosen in the interval [1, 2]. Figure below shows that the relative difference from the correct value of x is never greater than 10%.

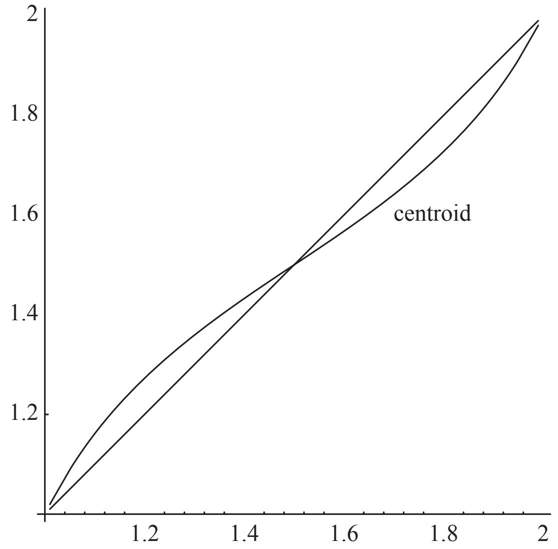


Fig. 7.13 Reconstruction error of the centroid method

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

7.8 Control with Fuzzy Logic

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

7.8.1 Fuzzy Controllers

The example with the electrical heater will be completed in this section. We must determine the domain of definition of the variables used in the problem. Assume that the room temperature is a number between 0 and 40 degrees Celsius. The controller can vary the electrical power consumed between 0 and 100 (in some suitable units), whereby 50 is the normal stand-by value.

Figure below shows the membership functions for the temperature categories “cold”, “normal”, and “warm” and the control categories “reduce”, “maintain”, and “heat”.

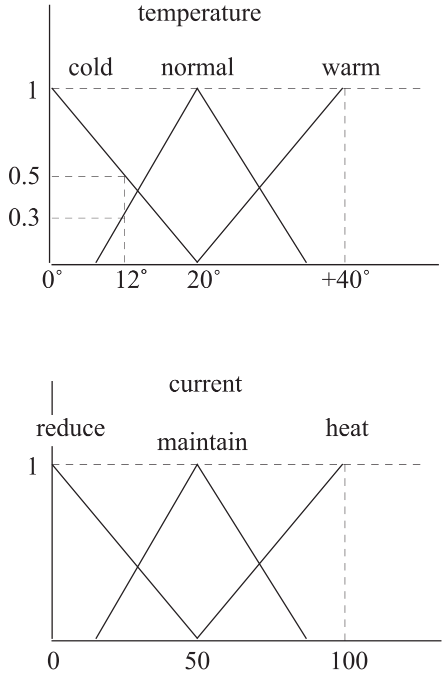


Fig. 7.14 Membership functions for temperature and electric current categories

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

It is of course possible to formulate more complex rules involving more than two variables. In all cases, though, we have to evaluate all rules simultaneously. Kosko shows some examples of dynamical systems with three or more control variables [259].

7.8.2 Fuzzy Networks

Fuzzy systems can be represented as networks. The computing units must implement fuzzy operators. Figure “Example of a fuzzy network**”** shows a network with four hidden units. Each one of them receives the inputs x1, x2 and x3 which correspond to the fuzzy categorisation of a specific number. The fuzzy operators are evaluated in parallel in the hidden layer of the network, which corresponds to the set of inference rules. The last unit in the network is the de-fuzzifier, which transforms the fuzzy inferences into a specific control variable.

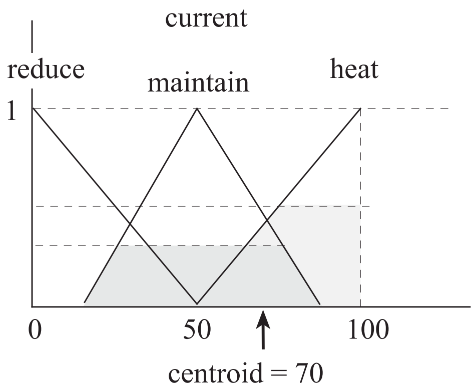


Fig. 7.15 Centroid computation

The importance of each fuzzy inference rule is weighted by the numbers α1, α2, and α3 as in a weighted centroid computation.

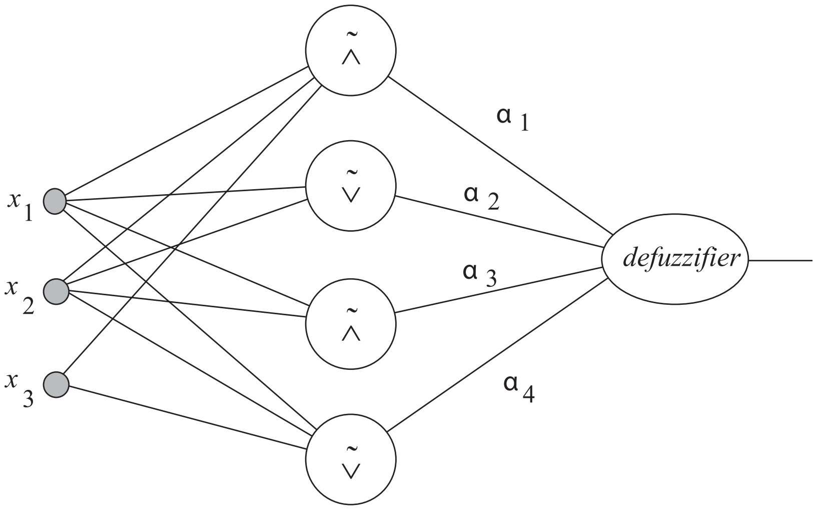


Fig. 7.16 Example of a fuzzy network

More complex rules can be implemented and this can lead to networks with several layers. However, fuzzy systems do not usually lead to very deep networks. Since at each fuzzy inference step the precision of the conclusion is reduced, it is not advisable to build too long an inference chain. Fuzzy operators cannot be computed exactly by sigmoidal units, but for some of them a relatively good approximation is possible, for example, for bounded sum or bounded difference. A fuzzy inference chain using these operators can be approximated by a neural network.

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

7.8.3 Function Approximation with Fuzzy Methods

A fuzzy controller is just a system for the rapid computation of an approximation of a coarsely defined control surface, like the one shown in Figure 7.17. The fuzzy controller computes a control variable according to the values of the variables x and y. Both variables are transformed into fuzzy categories. Assume that each variable is transformed into a combination of three categories. There are nine different combinations of the categories for x and y. For each of these nine combinations the value of the control variable is defined. This fixes nine points of the control surface.

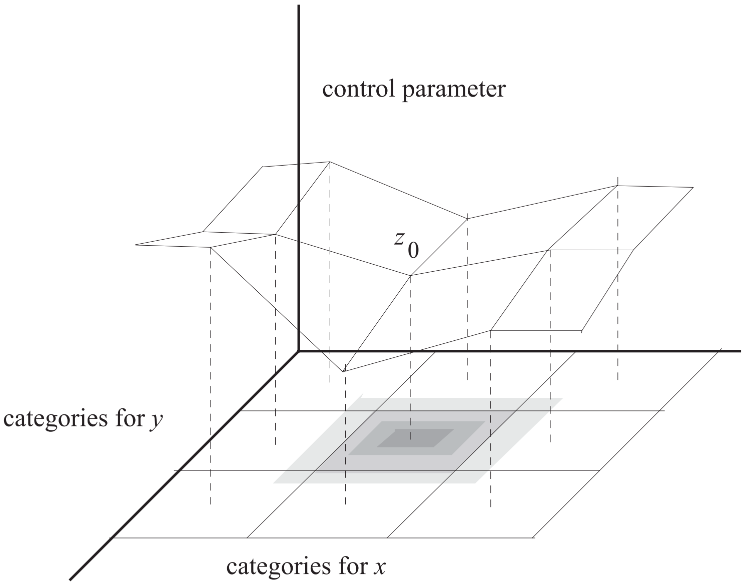


Fig. 7.17 Approximation of a control surface

Arbitrary values of x and y belong, to different degrees, to the nine combined categories. This means that for arbitrary combinations of x and y an interpolation of the known function values of the control variable is needed. A fuzzy controller performs this computation according to the degree of membership of (x, y) in each combined category. In Figure 7.17 the different shadings of the quadratic regions in the *xy* plane represent the membership of the input in the category for which the control variable assumes the value z0. Other values, which correspond to the lighter shaded region, receive a value for the control variable which is an interpolation of the neighbouring z-values.

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

7.8.4 The Eye as a Fuzzy System – Colour Vision

It is interesting to note that the eye makes use of a similar strategy to fuzzy controllers with regard to colour vision. Photons of different wavelengths, corresponding to different colours in the visible spectrum, impinge on the retina. The eye contains receptors for only three types of colours. We can find in the cochlea different receptors for many of the frequencies present in sounds that we can hear, but in the eyes we find only receptors that are maximally excited with light from the spectral regions corresponding to the colours blue, green, and red. That is why the receptors have received the name of the colours they detect. Colour vision must accommodate sensors to process a two-dimensional image at every pixel and this can only be done by reducing the number of detector types available.

The visible spectrum for humans extends from 400 up to 650 nano-meters wavelength. A monochromatic colour, that is, a colour with a definite and crisp wavelength, excites all three receptor types in the retina. The output of each receptor class, however, is not identical but depends on the wavelength of the light. It has been shown in many colour vision experiments, and later through direct measurements, that the output functions of the three receptor classes correspond to those shown in Figure 7.18. Blue receptors, for example, reach their maximal excitation for wavelengths around 430 nm. Green receptors respond maximally at 530 nm and red receptors at 560 nm.

When monochromatic light excites the receptors on the retina its wavelength is transformed into three different excitation levels, that is, into a relative excitation of the three receptor types. The wavelength is transformed into a fuzzy category, just as in the case of fuzzy controllers. The three excitation levels measure the degree of membership in each of the three colour categories blue, green, and red. The subsequent processing of the colour information is performed based on this and additional coding steps (for example, comparing complementary colours). This is why a mixture of two colours is perceived by the brain as a third colour.

Some simple physiological considerations show that good colour discrimination requires at least three types of receptors. Coding of the wavelength using three excitation values reduces the number of rules needed in subsequent processing. The sparseness of rules in fuzzy controllers finds its equivalent here in the sparseness of the biological organs.

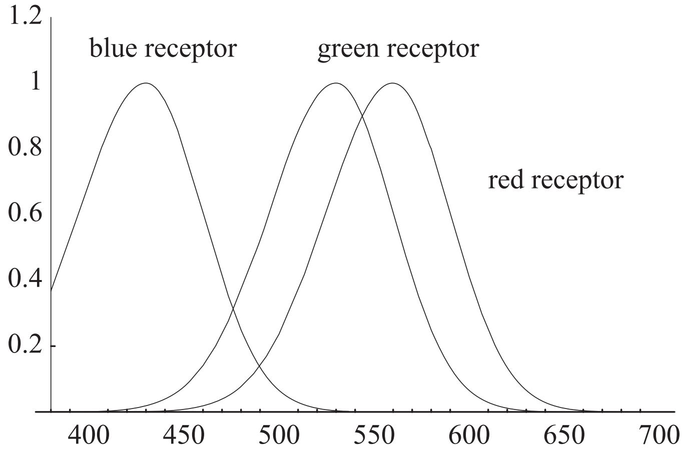


Fig. 7.18 Response function of the three receptors in the retina

Summary

- Learning algorithms may require an exponential number of iterations with respect to the number of weights until a solution to a learning task is found.
- In back propagation networks, the individual units perform computations more general than simple threshold logic.
- Fuzzy logic can be conceptualised as a generalisation of classical logic.
- In fuzzy logic, which is also sometimes called diffuse logic, there are not just two alternatives but a whole continuum of truth values for logical propositions.
- Fuzzy logic is now being used in many products of industrial and consumer electronics for which a good control system is sufficient and where the question of optimal control does not necessarily arise.
- The difference between crisp that is classical and fuzzy sets is established by introducing a membership function.
- Let X be a classical universal set. A real function $\mu A : X \rightarrow [0, 1]$ is called the membership function of A and defines the fuzzy set A of X. This is the set of all pairs $(x, \mu A(x))$ with $x \in X$.
- Bart Kosko introduced a very useful graphical representation of fuzzy sets.
- The union or intersection of two fuzzy sets is in general a fuzzy, not a crisp set.
- This definition of cardinality corresponds to the distance of the representation of A from the origin using a Manhattan metric.
- The entropy of a crisp set is always zero, since for a crisp set $A \cap A^c = \emptyset$.
- It is known in mathematics that an isomorphism exists between set theory and classic propositional logic.
- A fuzzy logic can be also derived from fuzzy set theory by respecting the isomorphism.
- Depending on the axioms selected, different fuzzy operators and different logic systems can be defined.
- Associativity is more difficult to visualise but it roughly indicates that the function does not grow abruptly in some regions and stagnate in others.
- Fuzzy logic operators can be used as the basis for inference systems.

References

- Klir, J. G., Yuan, B., 1995. *Fuzzy sets and fuzzy logic: Theory and Applications*, Prentice Hall PTR.
- Ganesh, M., 2006. *Introduction To Fuzzy Sets and Fuzzy Logic*, PHI Learning Pvt. Ltd.
- *Fuzzy Logic*, [Online] Available at: <http://page.mi.fu-berlin.de/rojas/neural/chapter/K11.pdf> [Accessed 19 July 2012].
- Watkins, T., Valley, S., & Alley, T., *Fuzzy Logic: The Logic of Fuzzy Sets*, [Online] Available at: <http://www.sjsu.edu/faculty/watkins/fuzzysets.htm> [Accessed 19 July 2012].
- EngineeringVideosNet, 11th May 2010. *Fuzzy Sets Example*, [Video Online] Available at: <http://www.youtube.com/watch?v=5TqyJGfgwbw> [Accessed 19 July 2012].
- ignousocis, 15th Oct 2008. *Fuzzy Logic Why and What 1*, Video Online] Available at: <http://www.youtube.com/watch?v=YM4PVm5HDDY> [Accessed 19 July 2012].

Recommended Reading

- Zadeh, A. L., Klir, J. G.& Yuan, B., 1996. *Fuzzy Sets, Fuzzy Logic, and Fuzzy Systems: Selected Papers*, World Scientific.
- Buckley, J. J.& Eslami, E., 2002. *An Introduction to Fuzzy Logic and Fuzzy Sets*, Springer.
- Bojadziev, G.& Bojadziev, M., 1995. *Fuzzy Sets, Fuzzy Logic, Applications*, World Scientific.

Self Assessment

1. Modern fuzzy logic was developed by _______ in the mid-1960s.
   a. Lotfi Zadeh
   b. Bart Kosko
   c. Yager
   d. Pitts

2. Which of the following statements is false?
   a. Fuzzy logic can be used as an interpretation model for the properties of neural networks.
   b. A fuzzy set theory do not corresponds to fuzzy logic.
   c. Fuzzy logic can be used to specify networks directly without having to apply a learning algorithm.
   d. A classical example proposed by Zadeh to the neural network community is developing a system to park a car.

3. The difference between crisp that is classical and fuzzy sets is established by introducing a _________ function.
   a. membership
   b. tree
   c. real
   d. point

4. A _________ is completely determined by its membership function.
   a. fuzzy logic
   b. crisp set
   c. fuzzy intersection
   d. fuzzy set

5. The union or intersection of two fuzzy sets is in general a _______.
   a. crisp set
   b. fuzzy
   c. classical
   d. In

6. A fuzzy logic can be also derived from fuzzy set theory by respecting the _________ .
   a. polymorphism
   b. dimorphism
   c. isomorphism
   d. no- isomorphism

7. Monotonicity of the operators allows only those function graphs that do not ______.
   a. fold
   b. change
   c. crisp
   d. delete

8. _______ roughly indicates that the function does not grow abruptly in some regions and stagnate in others.
   a. Monotonicity
   b. Commutativity
   c. Associativity
   d. Centroid

9. The ________ method produces better or worse inverse transformations depending on the placement of the triangular categories.
   a. centroid
   b. controller
   c. fuzzified
   d. defuzzifier

10. _______ that can only be formulated in a fuzzy, imprecise manner can be captured in rules that can be processed by a computer.
   a. Data
   b. Information
   c. Signal
   d. Knowledge

Chapter VIII

Genetic Algorithms

Aim

The aim of this chapter is to:

- introduce genetic algorithms
- elucidate brief history of evolutionary computation
- explain elements of genetic algorithms

Objectives

The objectives of this chapter are to:

- explain working of genetic algorithms
- explicate fitness function
- elucidate sorting networks

Learning Outcome

At the end of this chapter, you will be able to:

- understand search spaces and fitness landscapes
- elucidate genetic algorithm and examples
- understand traditional search methods

8.1 Introduction

Science arises from the very human desire to understand and control the world. Over the course of history, we humans have gradually built up a grand edifice of knowledge that enables us to predict, to varying extents, the weather, the motions of the planets, solar and lunar eclipses, the courses of diseases, the rise and fall of economic growth, the stages of language development in children, and a vast panorama of other natural, social, and cultural phenomena. More recently we have even come to understand some fundamental limits to our abilities to predict. Over the eons we have developed increasingly complex means to control many aspects of our lives and our interactions with nature, and we have learned, often the hard way, the extent to which other aspects are uncontrollable.

The advent of electronic computers has arguably been the most revolutionary development in the history of science and technology. This ongoing revolution is profoundly increasing our ability to predict and control nature in ways that were barely conceived of even half a century ago. For many, the crowning achievements of this revolution will be the creation in the form of computer programs of new species of intelligent beings, and even of new forms of life.

The goals of creating artificial intelligence and artificial life can be traced back to the very beginnings of the computer age. The earliest computer scientists-Alan Turing, John von Neumann, Norbert Wiener, and others were motivated in large part by visions of imbuing computer programs with intelligence, with the life-like ability to self replicate, and with the adaptive capability to learn and to control their environments. These early pioneers of computer science were as much interested in biology and psychology as in electronics, and they looked to natural systems as guiding metaphors for how to achieve their visions. It should be no surprise, then, that from the earliest days computers were applied not only to calculating missile trajectories and deciphering military codes but also to modelling the brain, mimicking human learning, and simulating biological evolution.

These biologically motivated computing activities have waxed and waned over the years, but since the early 1980s they have all undergone resurgence in the computation research community. The first has grown into the field of neural networks, the second into machine learning, and the third into what is now called "evolutionary computation," of which genetic algorithms are the most prominent example.

8.2 A Brief History of Evolutionary Computation

In the 1950s and the 1960s several computer scientists independently studied evolutionary systems with the idea that evolution could be used as an optimisation tool for engineering problems. The idea in all these systems was to evolve a population of candidate solutions to a given problem, using operators inspired by natural genetic variation and natural selection.

In the 1960s, Rechenberg introduced "evolution strategies" (Evolutions strategies in the original German), a method he used to optimise real−valued parameters for devices such as airfoils. This idea was further developed by Schwefel. The field of evolution strategies has remained an active area of research, mostly developing independently from the field of genetic algorithms. Fogel, Owens, and Walsh developed "evolutionary programming" a technique in which candidate solutions to given tasks were represented as finite−state machines, which were evolved by randomly mutating their state−transition diagrams and selecting the fittest.

A somewhat broader formulation of evolutionary programming also remains an area of active research. Together, evolution strategies, evolutionary programming, and genetic algorithms form the backbone of the field of evolutionary computation. Several other people working in the 1950s and the 1960s developed evolution−inspired algorithms for optimisation and machine learning. Box, Friedman, Bledsoe, Bremermann, and Reed, Toombs, and Baricelli all worked in this area, though their work has been given little or none of the kind of attention or follow-up that evolution strategies, evolutionary programming, and genetic algorithms have seen.

8.3 The Appeal of Evolution

Why use evolution as an inspiration for solving computational problems? To evolutionary computation researchers, the mechanisms of evolution seem well suited for some of the most pressing computational problems in many fields. Many computational problems require searching through a huge number of possibilities for solutions. One example is the problem of computational protein engineering, in which an algorithm is sought that will search among the vast number of possible amino acid sequences for a protein with specified properties.

Another example is searching for a set of rules or equations that will predict the ups and downs of a financial market, such as that for foreign currency. Such search problems can often benefit from an effective use of parallelism, in which many different possibilities are explored simultaneously in an efficient way. For example, in searching for proteins with specified properties, rather than evaluate one amino acid sequence at a time it would be much faster to evaluate many simultaneously. What is needed is both computational parallelism that is, many processors evaluating sequences at the same time and an intelligent strategy for choosing the next set of sequences to evaluate.

Many computational problems require a computer program to be adaptive to continue to perform well in a changing environment. This is typified by problems in robot control in which a robot has to perform a task in a variable environment, and by computer interfaces that must adapt to the idiosyncrasies of different users. Other problems require computer programs to be innovative to construct something truly new and original, such as a new algorithm for accomplishing a computational task or even a new scientific discovery.

Finally, many computational problems require complex solutions that are difficult to program by hand. A striking example is the problem of creating artificial intelligence. Early on, AI practitioners believed that it would be straightforward to encode the rules that would confer intelligence on a program; expert systems were one result of this early optimism. Nowadays, many AI researchers believe that the "rules" underlying intelligence are too complex for scientists to encode by hand in a "top-down" fashion. Instead they believe that the best route to artificial intelligence is through a "bottom−up" paradigm in which humans write only very simple rules, and complex behaviours such as intelligence emerge from the massively parallel application and interaction of these simple rules.

Connectionism is the study of computer programs inspired by neural systems. It is one example of this philosophy; evolutionary computation is another. In connectionism the rules are typically simple "neural" thresholding, activation spreading, and strengthening or weakening of connections; the hoped−for emergent behaviour is sophisticated pattern recognition and learning.

In evolutionary computation the rules are typically "natural selection" with variation due to crossover and/or mutation; the hoped−for emergent behaviour is the design of high−quality solutions to difficult problems and the ability to adapt these solutions in the face of a changing environment. Biological evolution is an appealing source of inspiration for addressing these problems. Evolution is, in effect, a method of searching among an enormous number of possibilities for "solutions." In biology the enormous set of possibilities is the set of possible genetic sequences, and the desired "solutions" are highly fit organisms well able to survive and reproduce in their environments.

Evolution can also be seen as a method for designing innovative solutions to complex problems. For example, the mammalian immune system is a marvellous evolved solution to the problem of germs invading the body. Seen in this light, the mechanisms of evolution can inspire computational search methods. Of course the fitness of a biological organism depends on many factors for example, how well it can weather the physical characteristics of its environment and how well it can compete with or cooperate with the other organisms around it. The fitness criteria continually change as creatures evolve, so evolution is searching a constantly changing set of possibilities.

Searching for solutions in the face of changing conditions is precisely what is required for adaptive computer programs. Furthermore, evolution is a massively parallel search method: rather than work on one species at a time, evolution tests and changes millions of species in parallel. Finally, viewed from a high level the "rules" of evolution are remarkably simple: species evolve by means of random variation (via mutation, recombination, and other operators), followed by natural selection in which the fittest tend to survive and reproduce, thus propagating their genetic material to future generations.

8.4 Search Spaces and Fitness Landscapes

The idea of searching among a collection of candidate solutions for a desired solution is so common in computer science that it has been given its own name: searching in a "search space." Here the term "search space" refers to some collection of candidate solutions to a problem and some notion of "distance" between candidate solutions. For an example, let us take one of the most important problems in computational bioengineering: the aforementioned problem of computational protein design.

Suppose you want use a computer to search for a protein a sequence of amino acids that folds up to a particular three dimensional shape so it can be used, say, to fight a specific virus. The search space is the collection of all possible protein sequences an infinite set of possibilities. To constrain it, let us restrict the search to all possible sequences of length 100 or less—still a huge search space, since there are 20 possible amino acids at each position in the sequence. If we represent the 20 amino acids by letters of the alphabet, candidate solutions will look like this:

A G G M C G B L….

We will define the distance between two sequences as the number of positions in which the letters at corresponding positions differ. For example, the distance between A G G M C G B L and MG G M C G B L is 1, and the distance between A G G M C G B L and L B M P A F G A is 8. An algorithm for searching this space is a method for choosing which candidate solutions to test at each stage of the search. In most cases the next candidate solution(s) to be tested will depend on the results of testing previous sequences; most useful algorithms assume that there will be some correlation between the qualities of "neighbouring" candidate solutions those close in the space. Genetic algorithms assume that high quality "parent" candidate solutions from different regions in the space can be combined via crossover to, on occasion, produce high quality "offspring" candidate solutions.

Another important concept is that of "fitness landscape." Originally defined by the biologist Sewell Wright in the context of population genetics, a fitness landscape is a representation of the space of all possible genotypes along with their fitness's. Suppose, for the sake of simplicity, that each genotype is a bit string of length l, and that the distance between two genotypes is their "Hamming distance" the number of locations at which corresponding bits differs. Also suppose that each genotype can be assigned a real−valued fitness. A fitness landscape can be pictured as an (l + 1) dimensional plot in which each genotype is a point in l dimensions and its fitness is plotted along the (l + 1)st axis.

A simple landscape for l = 2 is shown in figure below Such plots are called landscapes because the plot of fitness values can form "hills," "peaks," "valleys," and other features analogous to those of physical landscapes. Under Wright's formulation, evolution causes populations to move along landscapes in particular ways, and "adaptation" can be seen as the movement toward local peaks. (A "local peak," or "local optimum," is not necessarily the highest point in the landscape, but any small movement away from it goes downward in fitness.) Likewise, in GAs the operators of crossover and mutation can be seen as ways of moving a population around on the landscape defined by the fitness function.

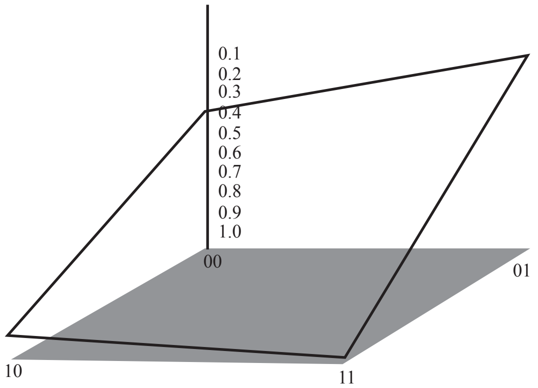


Fig. 8.1 A simple fitness landscape for l = 2. Here f(00) = 0.7, f(01) = 1.0, f(10) = 0.1, and f(11) = 0.0

The idea of evolution moving populations around in unchanging landscapes is biologically unrealistic for several reasons. For example, an organism cannot be assigned a fitness value independent of the other organisms in its environment; thus, as the population changes, the fitness's of particular genotypes will change as well. In other words, in the real world the "landscape" cannot be separated from the organisms that inhabit it. In spite of such caveats, the notion of fitness landscape has become central to the study of genetic algorithms, and it will come up in various guises throughout this book.

8.5 Elements of Genetic Algorithms

It turns out that there is no rigorous definition of "genetic algorithm" accepted by all in the evolutionary computation community that differentiates GAs from other evolutionary computation methods. However, it can be said that most methods called "GAs" have at least the following elements in common: populations of chromosomes, selection according to fitness, crossover to produce new offspring, and random mutation of new offspring. Inversion Holland's fourth element of GAs is rarely used in today's implementations, and its advantages, if any, are not well established.

The chromosomes in a GA population typically take the form of bit strings. Each locus in the chromosome has two possible alleles: 0 and 1. Each chromosome can be thought of as a point in the search space of candidate solutions. The GA processes populations of chromosomes, successively replacing one such population with another. The GA most often requires a fitness function that assigns a score (fitness) to each chromosome in the current population. The fitness of a chromosome depends on how well that chromosome solves the problem at hand.

8.6 Examples of Fitness Functions

One common application of GAs is function optimisation, where the goal is to find a set of parameter values that maximise, say, a complex multi-parameter function. As a simple example, one might want to maximise the real−valued one−dimensional function.

$$f(y) = y + |\sin(32y)|,\ 0 \leq y < \pi$$

Here the candidate solutions are values of y, which can be encoded as bit strings representing real numbers. The fitness calculation translates a given bit string x into a real number y and then evaluates the function at that value. The fitness of a string is the function value at that point. As a non−numerical example, consider the problem of finding a sequence of 50 amino acids that will fold to a desired three−dimensional protein structure. A GA could be applied to this problem by searching a population of candidate solutions, each encoded as a 50 letter string such as,

IHCCVASASDMIKPVFTVASYLKNWTKAKGPNFEICISGRTPYWDNFPGI

Where each letter represents one of 20 possible amino acids; one way to define the fitness of a candidate sequence is as the negative of the potential energy of the sequence with respect to the desired structure. The potential energy is a measure of how much physical resistance the sequence would put up if forced to be folded into the desired structure the lower the potential energy, the higher the fitness. Of course one would not want to physically force every sequence in the population into the desired structure and measure its resistance this would be very difficult, if not impossible.

Instead, given a sequence and a desired structure (and knowing some of the relevant biophysics), one can estimate the potential energy by calculating some of the forces acting on each amino acid, so the whole fitness calculation can be done computationally. These examples show two different contexts in which candidate solutions to a problem are encoded as abstract chromosomes encoded as strings of symbols, with fitness functions defined on the resulting space of strings. A genetic algorithm is a method for searching such fitness landscapes for highly fit strings.

8.7 GA Operators

The simplest form of genetic algorithm involves three types of operators: selection, crossover (single point), and mutation.

- Selection: This operator selects chromosomes in the population for reproduction. The fitter the chromosome, the more times it is likely to be selected to reproduce.
- Crossover: This operator randomly chooses a locus and exchanges the sub-sequences before and after that locus between two chromosomes to create two offspring. For example, the strings 10000100 and 11111111 could be crossed over after the third locus in each to produce the two offspring 10011111 and 11100100. The crossover operator roughly mimics biological recombination between two single−chromosome (haploid) organisms.
- Mutation: This operator randomly flips some of the bits in a chromosome. For example, the string 00000100 might be mutated in its second position to yield 01000100. Mutation can occur at each bit position in a string with some probability, usually very small for example, 0.001.

8.8. Simple Genetic Algorithm

Given a clearly defined problem to be solved and a bit string representation for candidate solutions, a simple GA works as follows:

- Start with a randomly generated population of n l−bit chromosomes (candidate solutions to a problem).
- Calculate the fitness $f(x)$ of each chromosome x in the population.
- Repeat the following steps until n offspring have been created:
    - Select a pair of parent chromosomes from the current population, the probability of selection being an increasing function of fitness. Selection is done “with replacement,” meaning that the same chromosome can be selected more than once to become a parent.
    - With probability pc (the “crossover probability” or “crossover rate”), cross over the pair at a randomly chosen point (chosen with uniform probability) to form two offspring. If no crossover takes place, form two offspring that are exact copies of their respective parents.
    - Mutate the two offspring at each locus with probability pm (the mutation probability or mutation rate), and place the resulting chromosomes in the new population. If n is odd, one new population member can be discarded at random.
- Replace the current population with the new population.
- Go to step 2.

Each iteration of this process is called a generation. A GA is typically iterated for anywhere from 50 to 500 or more generations. The entire set of generations is called a run. At the end of a run there are often one or more highly fit chromosomes in the population. Since randomness plays a large role in each run, two runs with different random number seeds will generally produce different detailed behaviours. GA researchers often report statistics (such as the best fitness found in a run and the generation at which the individual with that best fitness was discovered) averaged over many different runs of the GA on the same problem.

The simple procedure just described is the basis for most applications of GAs. There are a number of details to fill in, such as the size of the population and the probabilities of crossover and mutation, and the success of the algorithm often depends greatly on these details. There are also more complicated versions of GAs (example, GAs that work on representations other than strings or GAs that have different types of crossover and mutation operators). Many examples will be given in later chapters.

As a more detailed example of a simple GA, suppose that l (string length) is 8, that $f(x)$ is equal to the number of ones in bit string x (an extremely simple fitness function, used here only for illustrative purposes), that n(the population size)is 4, that pc = 0.7, and that pm = 0.001. (Like the fitness function, these values of l and n were chosen for simplicity. More typical values of l and n are in the range 50–1000. The values given for pc and pm are fairly typical.) The initial (randomly generated) population might look like this:

Chromosome Label	Chromosome String	Fitness
A	00000110	2
B	11101110	6
C	00100000	1
D	00110100	3

Table 8.1 Randomly generated populations

A common selection method in GAs is fitness proportionate selection, in which the number of times an individual is expected to reproduce is equal to its fitness divided by the average of fitness's in the population.

A simple method of implementing fitness proportionate selection is "roulette−wheel sampling", which is conceptually equivalent to giving each individual a slice of a circular roulette wheel equal in area to the individual's fitness. The roulette wheel is spun, the ball comes to rest on one wedge−shaped slice, and the corresponding individual is selected. In the n = 4 example above, the roulette wheel would be spun four times; the first two spins might choose chromosomes B and D to be parents, and the second two spins might choose chromosomes B and C to be parents. The fact that A might not be selected is just the luck of the draw. If the roulette wheel were spun many times, the average results would be closer to the expected values.

Once a pair of parents is selected, with probability pc they cross over to form two offspring. If they do not cross over, then the offspring are exact copies of each parent. Suppose, in the example above, that parents B and D cross over after the first bit position to form offspring E = 10110100 and F = 01101110, and parents B and C do not cross over, instead forming offspring that are exact copies of B and C. Next, each offspring is subject to mutation at each locus with probability pm. For example, suppose offspring E is mutated at the sixth locus to form E' = 10110000, offspring F and C are not mutated at all, and offspring B is mutated at the first locus to form B' = 01101110. The new population will be the following:

Chromosome Label	Chromosome String	Fitness
E'	10110000	3
F	01101110	5
C	00100000	1
B'	01101110	5

Note that, in the new population, although the best string (the one with fitness 6) was lost, the average fitness rose from 12/4 to 14/4. Iterating this procedure will eventually result in a string with all ones.

8.9 Genetic Algorithms and Traditional Search Methods

In the preceding sections I used the word "search" to describe what GAs do. It is important at this point to contrast this meaning of "search" with its other meanings in computer science. There are at least three (overlapping) meanings of "search": Search for stored data here the problem is to efficiently retrieve information stored in computer memory. Suppose you have a large database of names and addresses stored in some ordered way. What is the best way to search for the record corresponding to a given last name? "Binary search" is one method for efficiently finding the desired record. Knuth (1973) describes and analyzes many such search methods.

Search for paths to goals Here the problem is to efficiently find a set of actions that will move from a given initial state to a given goal. This form of search is central to many approaches in artificial intelligence. A simple example—all too familiar to anyone who has taken a course in AI—is the “8−puzzle,” illustrated in figure 8.2. A set of tiles numbered 1 to 8 are placed in a square, leaving one space empty. Sliding one of the adjacent tiles into the blank space is termed a “move.” Figure 8.2a illustrates the problem of finding a set of moves from the initial state to the state in which all the tiles are in order.

A partial search tree corresponding to this problem is illustrated in figure 8.2b The “root” node represents the initial state, the nodes branching out from it represent all possible results of one move from that state, and so on down the tree. The search algorithms discussed in most AI contexts are methods for efficiently finding the best (here, the shortest) path in the tree from the initial state to the goal state. Typical algorithms are “depth−first search,” “branch and bound,” and “A*.”

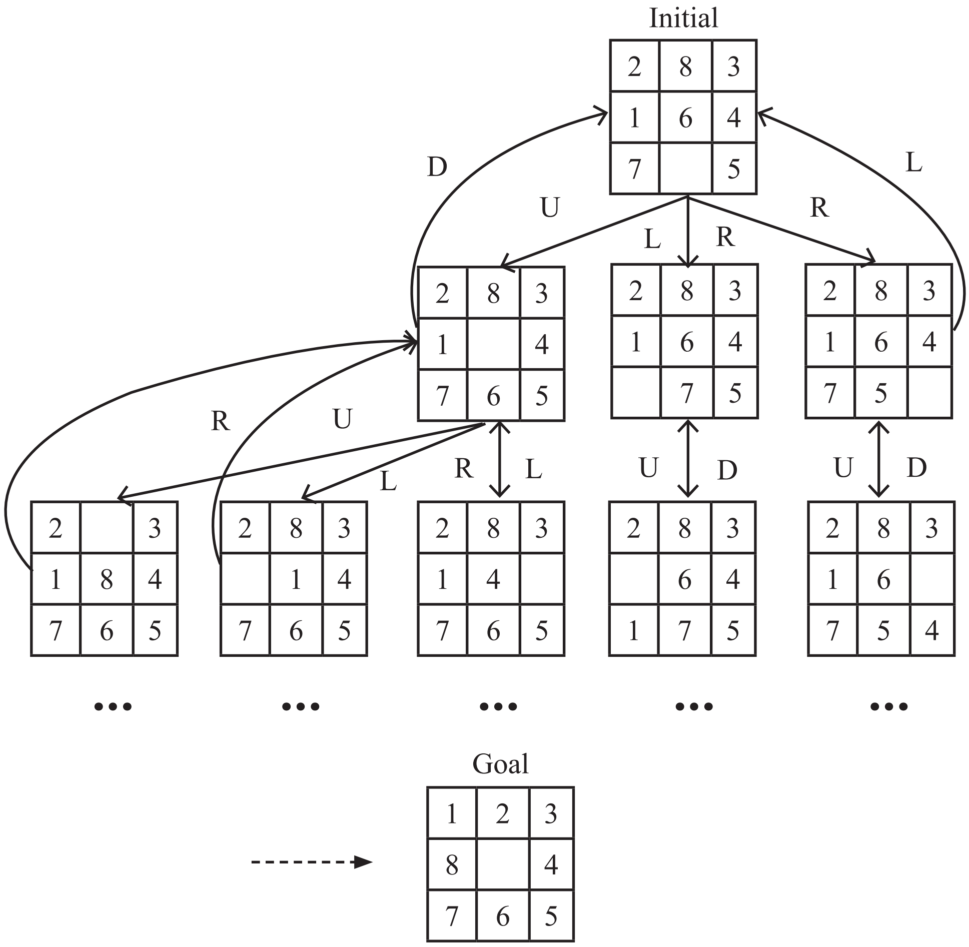


Fig. 8.2 The 8−puzzle. (a) The problem is to find a sequence of moves that will go from the initial state to the state with the tiles in the correct order (the goal state). (b) A partial search tree for the 8−puzzle.

Search for solutions

This is a more general class of search than “search for paths to goals.” The idea is to efficiently find a solution to a problem in a large space of candidate solutions. These are the kinds of search problems for which genetic algorithms are used. There is clearly a big difference between the first kind of search and the second two. The first concerns problems in which one needs to find a piece of information (e.g., a telephone number) in a collection of explicitly stored information. In the second two, the information to be searched is not explicitly stored; rather, candidate solutions are created as the search process proceeds.

For example, the AI search methods for solving the 8 puzzle do not begin with a complete search tree in which all the nodes are already stored in memory; for most problems of interest there are too many possible nodes in the tree to store them all. Rather, the search tree is elaborated step by step in a way that depends on the particular algorithm, and the goal is to find an optimal or high−quality solution by examining only a small portion of the tree. Likewise, when searching a space of candidate solutions with a GA, not all possible candidate solutions are created first and then evaluated; rather, the GA is a method for finding optimal or good solutions by examining only a small fraction of the possible candidates.

"Search for solutions" subsumes "search for paths to goals," since a path through a search tree can be encoded as a candidate solution. For the 8−puzzle, the candidate solutions could be lists of moves from the initial state to some other state (correct only if the final state is the goal state). However, many "search for paths to goals" problems are better solved by the AI tree−search techniques (in which partial solutions can be evaluated) than by GA or GA−like techniques (in which full candidate solutions must typically be generated before they can be evaluated). However, the standard AI tree−search (or, more generally, graph−search) methods do not always apply.

Not all problems require finding a path from an initial state to a goal. For example, predicting the three dimensional structure of a protein from its amino acid sequence does not necessarily require knowing the sequence of physical moves by which a protein folds up into a 3D structure; it requires only that the final 3D configuration be predicted. Also, for many problems, including the protein−prediction problem, the configuration of the goal state is not known ahead of time.

The GA is a general method for solving "search for solutions" problems (as are the other evolution−inspired techniques, such as evolution strategies and evolutionary programming). Hill climbing, simulated annealing, and tabu search are examples of other general methods. Some of these are similar to "search for paths to goals" methods such as branch−and−bound and A*. For descriptions of these and other search methods. "Steepest−ascent" hill climbing, for example, works as follows:

- Choose a candidate solution (e.g., encoded as a bit string) at random. Call this string current−string.
- Systematically mutate each bit in the string from left to right, one at a time, recording the fitness's of the resulting one−bit mutants.
- If any of the resulting one−bit mutants give a fitness increase, then set current−string to the one−bit mutant giving the highest fitness increase (the "steepest ascent").
- If there is no fitness increase, then save current−string (a "hilltop") and go to step 1. Otherwise, go to step 2 with the new current−string.
- When a set number of fitness−function evaluations has been performed, return the highest hilltop that was found.

In AI such general methods (methods that can work on a large variety of problems) are called "weak methods," to differentiate them from "strong methods" specially designed to work on particular problems. All the "search for solutions" methods:

- Initially generate a set of candidate solutions (in the GA this is the initial population; in steepest−ascent hill climbing this is the initial string and all the one−bit mutants of it)
- Evaluate the candidate solutions according to some fitness criteria
- Decide on the basis of this evaluation which candidates will be kept and which will be discarded
- Produce further variants by using some kind of operators on the surviving candidates

The particular combination of elements in genetic algorithms—parallel population−based search with stochastic selection of many individuals, stochastic crossover and mutation—distinguishes them from other search methods. Many other search methods have some of these elements, but not this particular combination.

8.10 Two Brief Examples

As warm ups to more extensive discussions of GA applications, here are brief examples of GAs in action on two particularly interesting projects. Using GAs to Evolve Strategies for the Prisoner's Dilemma The Prisoner's Dilemma, a simple two−person game invented by Merrill Flood and Melvin Dresher in the 1950s, has been studied extensively in game theory, economics, and political science because it can be seen as an idealised model for real−world phenomena such as arms races. It can be formulated as follows: Two individuals (call them Alice and Bob) are arrested for committing a crime together and are held in separate cells, with no communication possible between them.

Alice is offered the following deal: If she confesses and agrees to testify against Bob, she will receive a suspended sentence with probation, and Bob will be put away for 5 years. However, if at the same time Bob confesses and agrees to testify against Alice, her testimony will be discredited, and each will receive 4 years for pleading guilty. Alice is told that Bob is being offered precisely the same deal. Both Alice and Bob know that if neither testify against the other they can be convicted only on a lesser charge for which they will each get 2 years in jail.

Should Alice "defect" against Bob and hope for the suspended sentence, risking a 4−year sentence if Bob defects? Or should she "cooperate" with Bob (even though they cannot communicate), in the hope that he will also cooperate so each will get only 2 years, thereby risking a defection by Bob that will send her away for 5 years?

The game can be described more abstractly. Each player independently decides which move to make—i.e. whether to cooperate or defect. A "game" consists of each player's making a decision (a "move"). The possible results of a single game are summarised in a payoff matrix like the one shown in figure below. Here the goal is to get as many points (as opposed to as few years in prison) as possible. (In figure below, the payoff in each case can be interpreted as 5 minus the number of years in prison.) If both players cooperate, each gets 3 points. If player A defects and player B cooperates, then player A gets 5 points and player B gets 0 points, and vice versa if the situation is reversed.

If both players defect, each gets 1 point. What is the best strategy to use in order to maximise one's own payoff? If you suspect that your opponent is going to cooperate, then you should surely defect. If you suspect that your opponent is going to defect, then you should defect too. No matter what the other player does, it is always better to defect. The dilemma is that if both players defect each gets a worse score than if they cooperate. If the game is iterated (that is, if the two players play several games in a row), both players' always defecting will lead to a much lower total payoff than the players would get if they.

		Player B	
		Cooperate	Defect
Player A	Cooperate	3,3	0,5
	Defect	5,0	1,1

Fig. 8.3 The payoff matrix for the Prisoner's Dilemma

The two numbers given in each box are the payoffs for players A and B in the given situation, with player A's payoff listed first in each pair cooperated. How can reciprocal cooperation be induced? This question takes on special significance when the notions of cooperating and defecting correspond to actions in, say, a real−world arms race (e.g., reducing or increasing one's arsenal).

Robert Axelrod of the University of Michigan has studied the Prisoner's Dilemma and related games extensively. His interest in determining what makes for a good strategy led him to organise two Prisoner's Dilemma tournaments (described in Axelrod 1984). He solicited strategies from researchers in a number of disciplines. Each participant submitted a computer program that implemented a particular strategy, and the various programs played iterated games with each other. During each game, each program remembered what move (i.e., cooperate or defect) both it and its opponent had made in each of the three previous games that they had played with each other, and its strategy was based on this memory.

The programs were paired in a round–robin tournament in which each played with all the other programs over a number of games. The first tournament consisted of 14 different programs; the second consisted of 63 programs (including one that made random moves). Some of the strategies submitted were rather complicated, using techniques such as Markov processes and Bayesian inference to model the other players in order to determine the best move. However, in both tournaments the winner (the strategy with the highest average score) was the simplest of the submitted strategies: TIT FOR TAT. This strategy, submitted by Anatol Rapoport, cooperates in the first game and then, in subsequent games, does whatever the other player did in its move in the previous game with TIT FOR TAT. That is, it offers cooperation and reciprocates it. But if the other player defects, TIT FOR TAT punishes that defection with a defection of its own, and continues the punishment until the other player begins cooperating again.

After the two tournaments, Axelrod (1987) decided to see if a GA could evolve strategies to play this game successfully. The first issue was figuring out how to encode a strategy as a string. Here is how Axelrod's encoding worked. Suppose the memory of each player is one previous game. There are four possibilities for the previous game:

- CC (case 1)
- CD (case 2)
- DC (case 3)
- DD (case 4)

where C denotes "cooperate" and D denotes "defect." Case 1 is when both players cooperated in the previous game; case 2 is when player A cooperated and player B defected, and so on. A strategy is simply a rule that specifies an action in each of these cases. For example, TIT FOR TAT as played by player A is as follows:

- If CC (case 1), then C.
- If CD (case 2), then D.
- If DC (case 3), then C.
- If DD (case 4), then D.

If the cases are ordered in this canonical way, this strategy can be expressed compactly as the string CDCD. To use the string as a strategy, the player records the moves made in the previous game (e.g., CD), finds the case number *i* by looking up that case in a table of ordered cases like that given above (for CD, *i* = 2), and selects the letter in the i^{th} position of the string as its move in the next game (for *i* = 2, the move is D). Axelrod's tournaments involved strategies that remembered three previous games. There are 64 possibilities for the previous three games:

- CC CC CC (case 1),
- CC CC CD (case 2),
- CC CC DC (case 3),
- DD DD DC (case 63),
- DD DD DD (case 64).

Thus, a strategy can be encoded by a 64−letter string, example CDCCCDDCC CDD…. Since using the strategy requires the results of the three previous games, Axelrod actually used a 70−letter string, where the six extra letters encoded three hypothetical previous games used by the strategy to decide how to move in the first actual game. Since each locus in the string has two possible alleles (C and D), the number of possible strategies is 270. The search space is thus far too big to be searched exhaustively.

In Axelrod's first experiment, the GA had a population of 20 such strategies. The fitness of a strategy in the population was determined as follows: Axelrod had found that eight of the human−generated strategies from the second tournament were representative of the entire set of strategies, in the sense that a given strategy's score playing with these eight was a good predictor of the strategy's score playing with all 63 entries. This set of eight strategies (which did not include TIT FOR TAT) served as the "environment" for the evolving strategies in the population. Each individual in the population played iterated games with each of the eight fixed strategies, and the individual's fitness was taken to be its average score over all the games it played.

Axelrod performed 40 different runs of 50 generations each, using different random−number seeds for each run. Most of the strategies that evolved were similar to TIT FOR TAT in that they reciprocated cooperation and punished defection (although not necessarily only on the basis of the immediately preceding move). However, the GA often found strategies that scored substantially higher than TIT FOR TAT. This is a striking result, especially in view of the fact that in a given run the GA is testing only 20 × 50 = 1000 individuals out of a huge search space of 270 possible individuals.

It would be wrong to conclude that the GA discovered strategies that are "better" than any human−designed strategy. The performance of a strategy depends very much on its environment—that is, on the strategies with which it is playing. Here the environment was fixed—it consisted of eight human−designed strategies that did not change over the course of a run. The resulting fitness function is an example of a static (unchanging) fitness landscape. The highest−scoring strategies produced by the GA were designed to exploit specific weaknesses of several of the eight fixed strategies. It is not necessarily true that these high−scoring strategies would also score well in a different environment. TIT FOR TAT is a generalist, whereas the highest−scoring evolved strategies were more specialised to the given environment. Axelrod concluded that the GA is good at doing what evolution often does: developing highly specialised adaptations to specific characteristics of the environment.

To see the effects of a changing (as opposed to fixed) environment, Axelrod carried out another experiment in which the fitness of an individual was determined by allowing the individuals in the population to play with one another rather than with the fixed set of eight strategies. Now the environment changed from generation to generation because the opponents themselves were evolving. At every generation, each individual played iterated games with each of the 19 other members of the population and with itself, and its fitness was again taken to be its average score over all games. Here the fitness landscape was not static—it was a function of the particular individuals present in the population, and it changed as the population changed.

In this second set of experiments, Axelrod observed that the GA initially evolved uncooperative strategies. In the first few generations strategies that tended to cooperate did not find reciprocation among their fellow population members and thus tended to die out, but after about 10–20 generations the trend started to reverse: the GA discovered strategies that reciprocated cooperation and that punished defection (i.e., variants of TIT FOR TAT). These strategies did well with one another and were not completely defeated by less cooperative strategies, as were the initial cooperative strategies. Because the reciprocators scored above average, they spread in the population; this resulted in increasing cooperation and thus increasing fitness.

Axelrod's experiments illustrate how one might use a GA both to evolve solutions to an interesting problem and to model evolution and co-evolution in an idealised way. One can think of many additional possible experiments, such as running the GA with the probability of crossover set to 0—that is, using only the selection and mutation operators (Axelrod 1987) or allowing a more open−ended kind of evolution in which the amount of memory available to a given strategy is allowed to increase or decrease.

8.11 Hosts and Parasites: Using GAs to Evolve Sorting Networks

Designing algorithms for efficiently sorting collections of ordered elements is fundamental to computer science. Donald Knuth (1973) devoted more than half of a 700−page volume to this topic in his classic series The Art of Computer Programming. The goal of sorting is to place the elements in a data structure (example a list or a tree) in some specified order (example numerical or alphabetic) in minimal time. One particular approach to sorting described in Knuth's book is the sorting network, a parallelisable device for sorting lists with a fixed number n of elements. Figure below displays one such network that will sort lists of n = 16 elements (e0–e15). Each horizontal line represents one of the elements in the list, and each vertical arrow represents a comparison to be made between two elements.

For example, the leftmost column of vertical arrows indicates that comparisons are to be made between e0 and e1, between e2 and e3, and so on. If the elements being compared are out of the desired order, they are swapped.

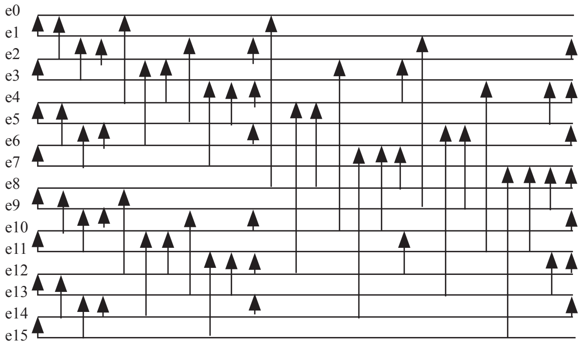


Fig. 8.4 The "Batcher sort" n=16 sorting network (adapted from Knuth 1973).

Each horizontal line represents an element in the list, and each vertical arrow represents a comparison to be made between two elements. If the elements being compared are out of order, they are swapped. Comparisons in the same column can be made in parallel. To sort a list of elements, one marches the list from left to right through the network, performing all the comparisons (and swaps, if necessary) specified in each vertical column before proceeding to the next. The comparisons in each vertical column are independent and can thus be performed in parallel. If the network is correct (as is the Batcher sort), any list will wind up perfectly sorted at the end. One goal of designing sorting networks is to make them correct and efficient (i.e., to minimise the number of comparisons).

An interesting theoretical problem is to determine the minimum number of comparisons necessary for a correct sorting network with a given n. In the 1960s there was a flurry of activity surrounding this problem for n = 16. According to Hillis, in 1962 Bose and Nelson developed a general method of designing sorting networks that required 65 comparisons for n = 16, and they conjectured that this value was the minimum. In 1964 there were independent discoveries by Batcher and by Floyd and Knuth of a network requiring only 63 comparisons (the network illustrated in figure 8.4). This was again thought by some to be minimal, but in 1969 Shapiro constructed a network that required only 62 comparisons. At this point, it is unlikely that anyone was willing to make conjectures about the network's optimality—and a good thing too, since in that same year Green found a network requiring only 60 comparisons. This was an exciting time in the small field of n = 16 sorting−network design. Things seemed to quiet down after Green's discovery, though no proof of its optimality was given.

In the 1980s, W. Daniel Hillis (1990,1992) took up the challenge again, though this time he was assisted by a genetic algorithm. In particular, Hillis presented the problem of designing an optimal n = 16 sorting network to a genetic algorithm operating on the massively parallel Connection Machine 2. As in the Prisoner's Dilemma example, the first step here was to figure out a good way to encode a sorting network as a string. Hillis's encoding was fairly complicated and more biologically realistic than those used in most GA applications. Here is how it worked: A sorting network can be specified as an ordered list of pairs, such as (2,5),(4,2),(7,14)….

These pairs represent the series of comparisons to be made ("first compare elements 2 and 5, and swap if necessary; next compare elements 4 and 2, and swap if necessary"). (Hillis's encoding did not specify which comparisons could be made in parallel, since he was trying only to minimise the total number of comparisons rather than to find the optimal parallel sorting network.) Sticking to the biological analogy, Hillis referred to ordered lists of pairs representing networks as "phenotypes." In Hillis's program, each phenotype consisted of 60–120 pairs, corresponding to networks with 60–120 comparisons. As in real genetics, the genetic algorithm worked not on phenotypes but on genotypes encoding the phenotypes.

The genotype of an individual in the GA population consisted of a set of chromosomes which could be decoded to form a phenotype. Hillis used diploid chromosomes (chromosomes in pairs) rather than the haploid chromosomes (single chromosomes) that are more typical in GA applications. As is illustrated in figure 8.5a, each individual consists of 15 pairs of 32−bit chromosomes. As is illustrated in figure 8.5b, each chromosome consists of eight 4−bit "codons." Each codon represents an integer between 0 and 15 giving a position in a 16−element list. Each adjacent pair of codons in a chromosome specifies a comparison between two list elements. Thus each chromosome encodes four comparisons. As is illustrated in figure 8.5c, each pair of chromosomes encodes between four and eight comparisons.

The chromosome pair is aligned and "read off" from left to right. At each position, the codon pair in chromosome A is compared with the codon pair in chromosome B. If they encode the same pair of numbers (that is "homozygous"), then only one pair of numbers is inserted in the phenotype; if they encode different pairs of numbers (that is are "heterozygou"), then both pairs are inserted in the phenotype. The 15 pairs of chromosomes are read off in this way in a fixed order to produce a phenotype with 60–120 comparisons. More homozygous positions appearing in each chromosome pair means fewer comparisons appearing in the resultant sorting network. The goal is for the GA to discover a minimal correct sorting network—to equal Green's network, the GA must discover an individual with all homozygous positions in its genotype that also yields a correct sorting network.

Note that under Hillis's encoding the GA cannot discover a network with fewer than 60 comparisons.

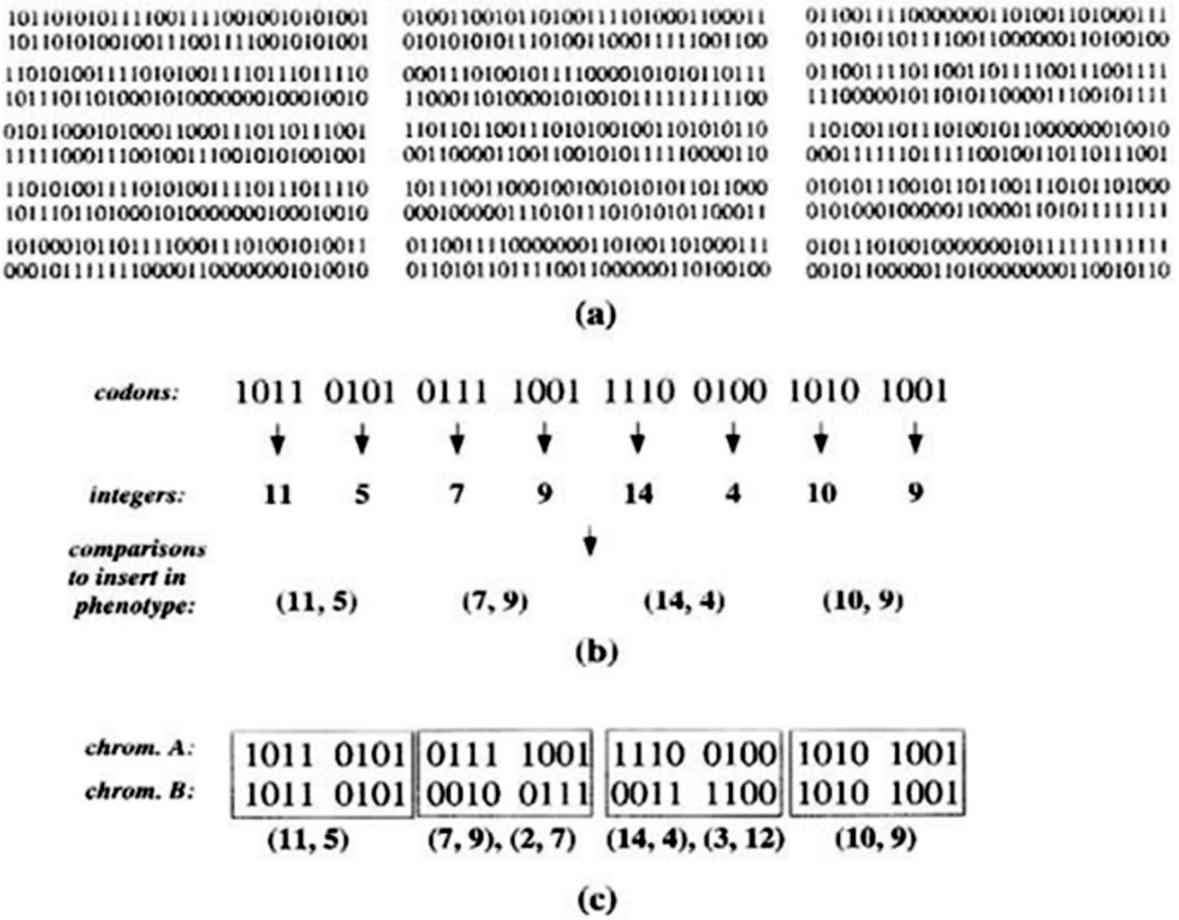


Fig. 8.5 Details of the genotype representation of sorting networks used in Hillis's experiments.

(a) An example of the genotype for an individual sorting network, consisting of 15 pairs of 32−bit chromosomes.
(b) An example of the integers encoded by a single chromosome. The chromosome given here encodes the integers 11, 5, 7, 9, 14, 4, 10, and 9; each pair of adjacent integers is interpreted as a comparison.
(c) An example of the comparisons encoded by a chromosome pair. The pair given here contains two homozygous positions and thus encodes a total of six comparisons to be inserted in the phenotype: (11, 5), (7, 9), (2, 7), (14, 4), (3, 12), and (10, 9).

In Hillis's experiments, the initial population consisted of a number of randomly generated genotypes, with one noteworthy provision: Hillis noted that most of the known minimal 16−element sorting networks begin with the same pattern of 32 comparisons, so he set the first eight chromosome pairs in each individual to (homozygously) encode these comparisons. This is an example of using knowledge about the problem domain (here, sorting networks) to help the GA get off the ground.

Most of the networks in a random initial population will not be correct networks—that is, they will not sort all input cases (lists of 16 numbers) correctly. Hillis's fitness measure gave partial credit: the fitness of a network was equal to the percentage of cases it sorted correctly. There are so many possible input cases that it was not practicable to test each network exhaustively, so at each generation each network was tested on a sample of input cases chosen at random.

Hillis's GA was a considerably modified version of the simple GA described above. The individuals in the initial population were placed on a two−dimensional lattice; thus, unlike in the simple GA, there is a notion of spatial distance between two strings. The purpose of placing the population on a spatial lattice was to foster "speciation" in the population—Hillis hoped that different types of networks would arise at different spatial locations, rather than having the whole population converge to a set of very similar networks.

he fitness of each individual in the population was computed on a random sample of test cases. Then the half of the population with lower fitness was deleted, each lower−fitness individual being replaced on the grid with a copy of a surviving neighbouring higher−fitness individual. That is, each individual in the higher−fitness half of the population was allowed to reproduce once. Next, individuals were paired with other individuals in their local spatial neighbourhoods to produce offspring. Recombination in the context of diploid organisms is different from the simple haploid crossover described above.

As figure below shows, when two individuals were paired, crossover took place within each chromosome pair inside each individual. For each of the 15 chromosome pairs, a crossover point was chosen at random, and a single “gamete” was formed by taking the codons before the crossover point from the first chromosome in the pair and the codons after the crossover point from the second chromosome in the pair. The result was 15 haploid gametes from each parent. Each of the 15 gametes from the first parent was then paired with one of the 15 gametes from the second parent to form a single diploid offspring. This procedure is roughly similar to sexual reproduction between diploid organisms in nature.

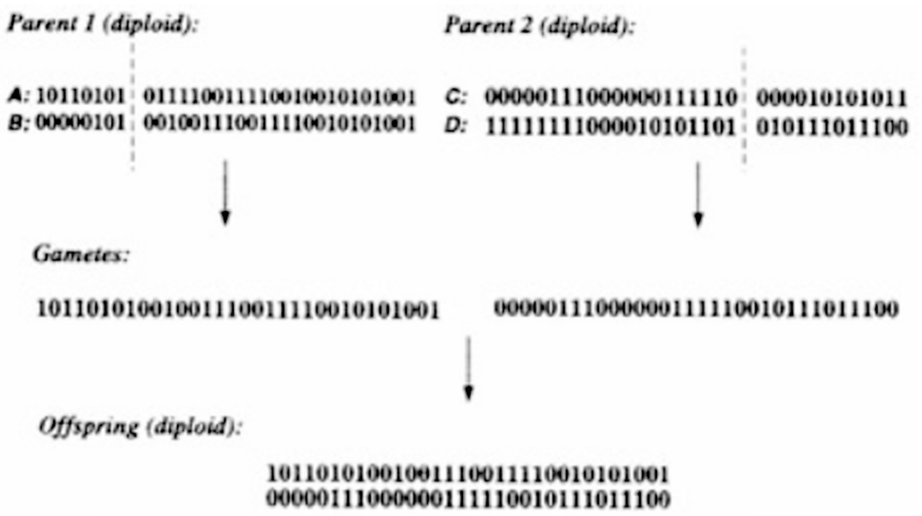


Fig. 8.6 An illustration of diploid recombination as performed in Hillis’s experiment.

Here an individual’s genotype consisted of 15 pairs of chromosomes (for the sake of clarity, only one pair for each parent is shown). A crossover point was chosen at random for each pair, and a gamete was formed by taking the codons before the crossover point in the first chromosome and the codons after the crossover point in the second chromosome. The 15 gametes from one parent were paired with the 15 gametes from the other parent to make a new individual. (Again for the sake of clarity, only one gamete pairing is shown.)

Such matings occurred until a new population had been formed. The individuals in the new population were then subject to mutation with pm = 0.001. This entire process was iterated for a number of generations. Since fitness depended only on network correctness, not on network size, what pressured the GA to find minimal networks? Hillis explained that there was an indirect pressure toward minimality, since, as in nature, homozygosity can protect crucial comparisons. If a crucial comparison is at a heterozygous position in its chromosome, then it can be lost under a crossover, whereas crucial comparisons at homozygous positions cannot be lost under crossover.

For example, in Fig. 8.6, the leftmost comparison in chromosome B (that is the leftmost eight bits, which encode the comparison (0, 5)) is at a heterozygous position and is lost under this recombination (the gamete gets its leftmost comparison from chromosome A), but the rightmost comparison in chromosome A (10, 9) is at a homozygous position and is retained (though the gamete gets its rightmost comparison from chromosome B). In general, once a crucial comparison or set of comparisons is discovered, it is highly advantageous for them to be at homozygous positions. And the more homozygous positions, the smaller the resulting network.

In order to take advantage of the massive parallelism of the Connection Machine, Hillis used very large populations, ranging from 512 to about 1 million individuals. Each run lasted about 5000 generations. The smallest correct network found by the GA had 65 comparisons, the same as in Bose and Nelson's network but five more than in Green's network. Hillis found this result disappointing—why didn't the GA do better? It appeared that the GA was getting stuck at local optima—local "hilltops" in the fitness landscape—rather than going to the globally highest hilltop.

The GA found a number of moderately good (65−comparison) solutions, but it could not proceed further. One reason was that after early generations the randomly generated test cases used to compute the fitness of each individual were not challenging enough. The networks had found a strategy that worked, and the difficulty of the test cases was staying roughly the same. Thus, after the early generations there was no pressure on the networks to change their current suboptimal sorting strategy.

To solve this problem, Hillis took another hint from biology: the phenomenon of host−parasite (or predator−prey) co-evolution. There are many examples in nature of organisms that evolve defences to parasites that attack them only to have the parasites evolve ways to circumvent the defences, which results in the hosts' evolving new defenses, and so on in an ever−rising spiral—a "biological arms race." In Hillis's analogy, the sorting networks could be viewed as hosts, and the test cases (lists of 16 numbers) could be viewed as parasites. Hillis modified the system so that a population of networks coevolved on the same grid as a population of parasites, where a parasite consisted of a set of 10–20 test cases. Both populations evolved under a GA. The fitness of a network was now determined by the parasite located at the network's grid location. The network's fitness was the percentage of test cases in the parasite that it sorted correctly.

The fitness of the parasite was the percentage of its test cases that stumped the network (i.e., that the network sorted incorrectly). The evolving population of test cases provided increasing challenges to the evolving population of networks. As the networks got better and better at sorting the test cases, the test cases got harder and harder, evolving to specifically target weaknesses in the networks. This forced the population of networks to keep changing—i.e., to keep discovering new sorting strategies—rather than staying stuck at the same suboptimal strategy. With co-evolution, the GA discovered correct networks with only 61 comparisons—a real improvement over the best networks discovered without co-evolution, but a frustrating single comparison away from rivalling Green's network.

Hillis's work is important because it introduces a new, potentially very useful GA technique inspired by co-evolution in biology, and his results are a convincing example of the potential power of such biological inspiration. However, although the host−parasite idea is very appealing, its usefulness has not been established beyond Hillis's work, and it is not clear how generally it will be applicable or to what degree it will scale up to more difficult problems (e.g., larger sorting networks). Clearly more work must be done in this very interesting area.

8.12 How do Genetic Algorithms Work?

Although genetic algorithms are simple to describe and program, their behaviour can be complicated, and many open questions exist about how they work and for what types of problems they are best suited. Much work has been done on the theoretical foundations of GAs. The traditional theory of GAs (first formulated in Holland 1975) assumes that, at a very general level of description, GAs work by discovering, emphasising, and recombining good "building blocks" of solutions in a highly parallel fashion. The idea here is that good solutions tend to be made up of good building blocks—combinations of bit values that confer higher fitness on the strings in which they are present.

Holland (1975) introduced the notion of schemas (or schemata) to formalise the informal notion of "building blocks." A schema is a set of bit strings that can be described by a template made up of ones, zeros, and asterisks, the asterisks representing wild cards (or "don't cares"). For example, the schema H = 1 * * * * 1 represents the set of all 6−bit strings that begin and end with 1. In this section we have used Goldberg's (1989a) notation, in which H stands for "hyper plane." H is used to denote schemas because schemas define hyper planes—"planes" of various dimensions—in the l-dimensional space of length−l bit strings.) The strings that fit this template (e.g., 100111 and 110011) are said to be instances of H. The schema H is said to have two defined bits (non−asterisks) or, equivalently, to be of order 2.

Its defining length (the distance between its outermost defined bits) is 5. Here I use the term "schema" to denote both a subset of strings represented by such a template and the template itself. In the following, the term's meaning should be clear from context.

Note that not every possible subset of the set of length−l bit strings can be described as a schema; in fact, the huge majority cannot. There are 2l possible bit strings of length l, and thus 22l possible subsets of strings, but there are only 3l possible schemas. However, a central tenet of traditional GA theory is that schemas are—implicitly—the building blocks that the GA processes effectively under the operators of selection, mutation, and single−point crossover.

How does the GA process schemas? Any given bit string of length l is an instance of 2l different schemas. For example, the string 11 is an instance of ** (all four possible bit strings of length 2), *1, 1*, and 11 (the schema that contains only one string, 11). Thus, any given population of n strings contains instances of between 2l and n × 21 different schemas. If all the strings are identical, then there are instances of exactly 2l different schemas; otherwise, the number is less than or equal to n × 2l. This means that, at a given generation, while the GA is explicitly evaluating the fitness's of the n strings in the population, it is actually implicitly estimating the average fitness of a much larger number of schemas, where the average fitness of a schema is defined to be the average fitness of all possible instances of that schema.

For example, in a randomly generated population of n strings, on average half the strings will be instances of 1***···* and half will be instances of 0 ***···*. The evaluations of the approximately n/2 strings that are instances of 1***···* give an estimate of the average fitness of that schema (this is an estimate because the instances evaluated in typical−size population are only a small sample of all possible instances). Just as schemas are not explicitly represented or evaluated by the GA, the estimates of schema average fitness's are not calculated or stored explicitly by the GA. However, as will be seen below, the GA's behaviour, in terms of the increase and decrease in numbers of instances of given schemas in the population, can be described as though it actually were calculating and storing these averages.

Summary

- Science arises from the very human desire to understand and control the world.
- The advent of electronic computers has arguably been the most revolutionary development in the history of science and technology.
- The goals of creating artificial intelligence and artificial life can be traced back to the very beginnings of the computer age.
- The earliest computer scientists-Alan Turing, John von Neumann, Norbert Wiener, and others were motivated in large part by visions of imbuing computer programs with intelligence.
- In the 1950s and the 1960s several computer scientists independently studied evolutionary systems with the idea that evolution could be used as an optimisation tool for engineering problems.
- Many computational problems require a computer program to be adaptive to continue to perform well in a changing environment.
- AI researchers believe that the "rules" underlying intelligence are too complex for scientists to encode by hand in a "top-down" fashion.
- Evolution can also be seen as a method for designing innovative solutions to complex problems.
- Selection operator selects chromosomes in the population for reproduction.
- Crossover operator randomly chooses a locus and exchanges the sub-sequences before and after that locus between two chromosomes to create two offspring.
- Mutation operator randomly flips some of the bits in a chromosome. For example, the string 00000100 might be mutated in its second position to yield 01000100.
- A GA is typically iterated for anywhere from 50 to 500 or more generations.
- A simple method of implementing fitness proportionate selection is "roulette−wheel sampling", which is conceptually equivalent to giving each individual a slice of a circular roulette wheel equal in area to the individual's fitness.
- AI search methods for solving the 8 puzzle do not begin with a complete search tree in which all the nodes are already stored in memory.
- Designing algorithms for efficiently sorting collections of ordered elements is fundamental to computer science.
- Although genetic algorithms are simple to describe and program, their behaviour can be complicated, and many open questions exist about how they work and for what types of problems they are best suited.

References

- Goldberg, *Genetic Algorithms*, Pearson Education India.
- Man, F. K., Tang, S. K. & Kwong, S., 1999. *Genetic Algorithms: Concepts and Designs*, 2nd ed. Springer.
- *AN INTRODUCTION TO GENETIC ALGORITHMS*, [Online] Available at: <http://www.boente.eti.br/fuzzy/ebook-fuzzy-mitchell.pdf> [Accessed 19 July 2012].
- *Chapter 3 GENETIC ALGORITHMS*, [Online] Available at: <http://sci2s.ugr.es/docencia/metaheuristicas/GeneticAlgorithms.pdf> [Accessed 19 July 2012].
- 2011. *Introduction to Genetic Algorithms*, [Video Online] Available at: <http://www.youtube.com/watch?v=zwYV11a__HQ> [Accessed 19 July 2012].
- 2011. *Mod-01 Lec-38 Genetic Algorithms*, [Video Online] Available at: <http://www.youtube.com/watch?v=Z_8MpZeMdD4> [Accessed 19 July 2012].

Recommended Reading

- Sivanandam, N. S. & Deepa, N. S., 2010. *Introduction to Genetic Algorithms*, Springer.
- Haupt, L. R. & Haupt, E. S., 2004. *Practical Genetic Algorithms*, 2nd ed. John Wiley & Sons.
- Sanchez, E., Shibata, T. & Zadeh, A. L., 1997. *Genetic Algorithms and Fuzzy Logic Systems: Soft Computing Perspectives*, World Scientific.

Self Assessment

1. Connectionism is the study of computer programs inspired by ______ systems.
   a. neural
   b. artificial
   c. intelligence
   d. fuzzy

2. A fitness landscape can be pictured as an _______ dimensional plot in which each genotype is a point in l dimensions.
   a. (l - 1)
   b. (l * 1)
   c. (l / 1)
   d. (l + 1)

3. _______ operator selects chromosomes in the population for reproduction.
   a. Mutation
   b. Selection
   c. Crossover
   d. Genetic

4. _________ operator randomly chooses a locus and exchanges the sub-sequences.
   a. Mutation
   b. Selection
   c. Crossover
   d. Genetic

5. _________ operator randomly flips some of the bits in a chromosome.
   a. Mutation
   b. Selection
   c. Crossover
   d. Genetic

6. The purpose of placing the population on a spatial lattice was to foster "_______" in the population.
   a. speciation
   b. strings
   c. grid
   d. fitness

7. Rechenberg introduced evolution strategies in ________.
   a. 1950
   b. 1960
   c. 1970
   d. 1980

8. The idea of evolution moving populations around in unchanging landscapes is biologically_________.
   a. realistic
   b. false
   c. unrealistic
   d. true

9. A __________ is a method for searching fitness landscapes for highly fit strings.
   a. fuzzy algorithms
   b. genetic algorithm
   c. fuzzy logic
   d. neural networks

10. The fitter the chromosome, the more times it is likely to be selected to_______.
    a. vanish
    b. death
    c. convert
    d. reproduce

Case Study 1

Handwriting Recognition-Neural Networks-Fuzzy Logic

Introduction
The main components of the work, or stages to learn the theme, are pre-processing, fuzzy feature extraction, and feed forward NN. The handwriting model combines both functions: segmentation and classification of a target handwritten letter. The character feature of such organising consists of the fact that functions of the components can be essentially changed.

Let us assume that a binary image of a character be the input data. Before we start the segmentation process we have to realise the pre-processing of the image which includes the following steps:

- Filtering: the aim is to reduce noise and make easier extracting the structural features
- Thinning (or skeletonising): this task removes outer pixels by iterative boundary erosion process until a skeleton of pixel chains only remains
- Searching vertices: at this step we extract a line junctions and the ends of the lines called vertices. Every vertex has a number of branches which meet each other in this point

After the pre-processing we get the thinned image with a set of vertices and its number of branches. We can perform a fuzzy feature extraction. The task of our feature extraction module is to process a binary image so that to obtain a vector of features. The main features are: a kind of segment, its orientation and size related to the character frame and position.

The features coding block includes a complex structure for input layer of a NN classifier. The input vector contains details for our character extracted from previous step. The last step is to classify the character. We apply the classic three-layer feed forward NN [Fu].

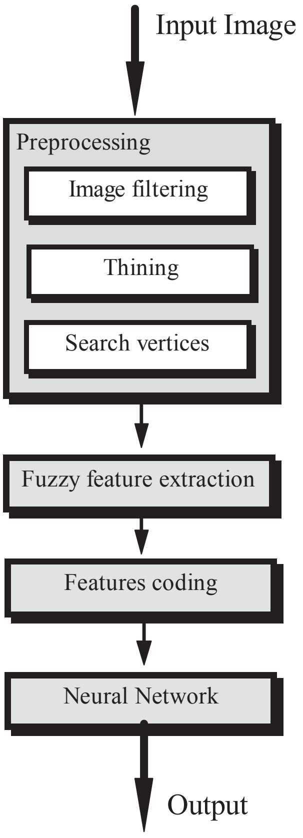


Fig. 1 The structure of handwriting recognition system

Fuzzy Feature Extraction Algorithm

The pre-processing of the image allows students studying a quite enough class of algorithms and estimates the influence of the obtained results onto quality of the recognition. Then the fuzzy feature extraction algorithm is learned in the work. The process of feature extraction of a character consists of the distinguishing typical elements (vertices) and branches or segments.

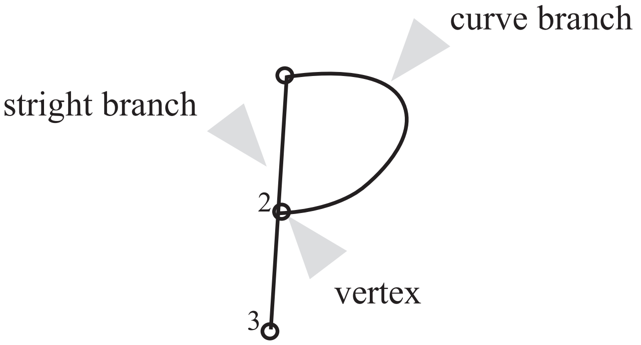


Fig. 2 Example of a character

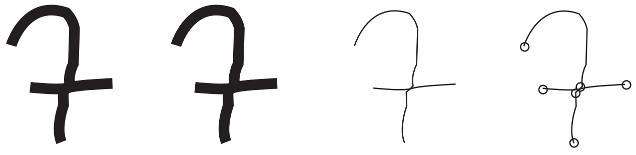
	line	right	left	loop
horizontal	—	∪	∩	○
vertical	|	)	(	
right sloped	/	╯	╭	
left sloped	\	╮	╰	

Table 1 Basic segments of a character with typical elements

Some different kinds of segments can be indicated, but taking into account the requirements we defined the simple segments shown in Table 1. It is clear that the number of segments in Table 1 can be extended, what is necessary, in particular, for signature authentication. Every vertex has its own co-ordinates *Vx* and *Vy* and the number of branches *Vn* which are crossed in point (*Vx, Vy*). The exception is for the vertex which lies on the end of the segment where *Vn* = 1.

The aim of this algorithm is to extract the defined segments from the character image. Before we start extraction, the pre-processing of the image should be perform first. As a result we achieve a thinned image I, a vertices structure V and the number of vertices *Vc*. The vertices structure *V* contains:

- Vertex co-ordinates *Vxi*,*Vyi*
- The number of branches *Vni*,, where i is an index of vertex, i = 0 .. *Vc*-1. The exact algorithm includes the following steps



Orginal noisy image

Filtered image

Thined image

Image with found vertices

Fig. 3 Main steps of pre-processing

Input data: a) m x n binary image matrix *I*
b) vertices structure *V*
c) number of vertices *Vc*

Output data: fuzzy data structure F of: a) symbols of classified segments, b) bound boxes of segments, c) position of segments, d) co-ordinates of the segment (start and end points), e) sizes of segments.

Step 1: Initialise the following variable: index of vertices i = 0, index of the branch j = 0 for the vertex Vi, m x n temporary image matrix I' with all zero elements, segment points co-ordinates matrix P of the current segment.

Step 2: Trace the segment of the branch j, starting from vertex co-ordinates Vxi, Vyi to the next found vertex:

- Check both the image matrix I and the temporary image matrix I' for the pixel If the pixel is set on I' it indicates that this pixel was previously traced and it is skipped. The tracing of the segment is stopped when tracer has no way to go
- All traced points are set on the temporary image matrix I' and their co-ordinates are saved to the matrix P. Save the start and the end point of the segment and the bound box which contains the path to the Fuzzy Data F.
- Note the number of segment pixels to k.

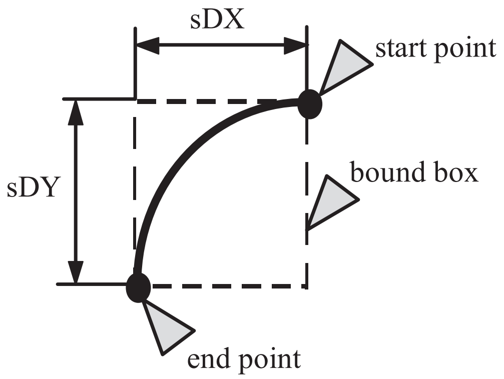


Fig. 4 Properties of the segment

Step 3: To qualify the segment as a loop, the segment must keep the condition d <= (k+d)*α / 360, where d is a distance from the beginning to the end of the segment, k is number of the segment points (number of elements in matrix P), α is the angle threshold (maximum width of an angle to classify a segment as a loop):

- If the below condition is checked the segment is classified as a loop. The size of the segment is calculated as at the step 6.
- Go to step 7.

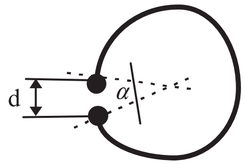


Fig. 5 Properties of the segment

Step 4: Calculate coefficients a and b of the line (from the start point to the end). We call it as a base line of the segment. Calculate the average deviation D of the distance for every segment point to the line y = ax + b, D= $\sqrt{\Sigma_{l=1}^{k-2} d_1^2}$, where k is the number of segment points, dl - distance from point l to base line y = ax + b, D is the average deviation of the distance.

To realise the classification of the segment orientation we define the following membership functions for fuzzy sets Horizontal (H), Vertical (V), Right Slope (R), Left Slope (L).

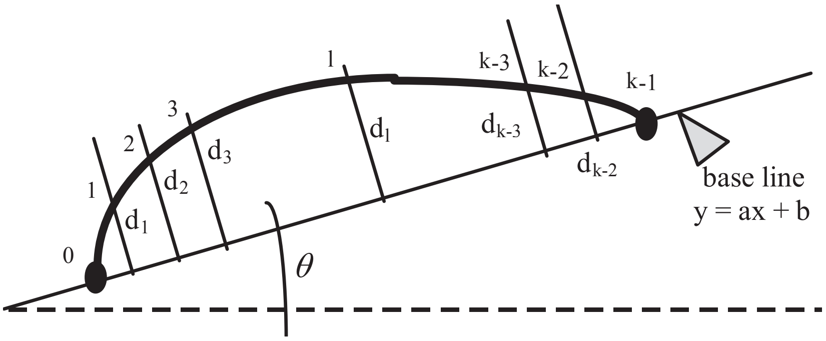
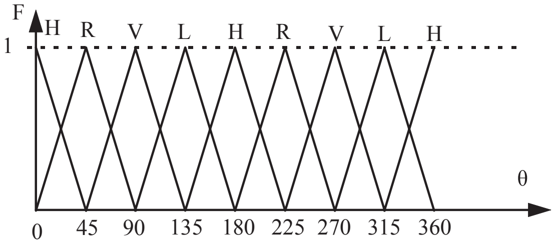


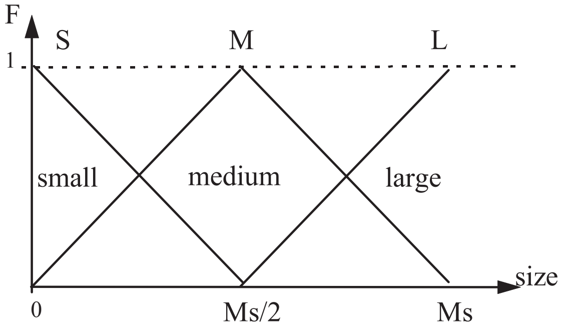
Fig. 6 Calculating the average deviation *D* of the distance for every segment point for the line *y = ax + b*.



$FH(\theta) = 1 - \min\{ \min[|\theta|, |180 - \theta|, |360 - \theta|]/45,1\}$
$FV(\theta) = 1 - \min\{ \min[|90 - \theta|, |270 - \theta|]/45, 1\}$
$FR(\theta) = 1 - \min\{ \min[|45 - \theta|, |225 - \theta|]/45, 1\}$
$FL(\theta) = 1 - \min\{ \min[|135 - \theta|, |315 - \theta|]/45, 1\}$
where $\theta = \text{arctag}\,(a)$,
F is a value of the above membership functions

Fig. 7 Calculating the average deviation *D* of the distance for every segment point for the line *y = ax + b*.

Step 5: If D <= β then the segment represents a line, where β is a curve threshold. This membership function of the fuzzy sets which has a maximum value indicates what orientation has the line segment. If D >β (there is a curve) go to step 6. Size of the segment is defined by fuzzy membership function shown below:



$$FS(size) = \begin{cases} 0 \text{ for } size > 0{,}5Ms \\ -2/Ms*size + 1 \text{ for } size <= 0{,}5Ms \end{cases}$$

$$FM(size) = -|2/Ms*size - 1| + 1$$

$$FL(size) = \begin{cases} 0 \text{ for } size < 0{,}5Ms \\ -2/S*size + 2 \text{ for } size > 0{,}5Ms \end{cases}$$

Fig. 8 Membership function for fuzzy size sets S, M, L

where, size = k, Ms = n and k - number of points belongs to the segment, n - height of the character. Go to step 7.

Step 6: Check what kind of curve corresponds to the segment.

- Calculate the number of intersections of the line LL, LR with the segment.
- Calculate coefficients of the two parallel lines LL and RL to the base line BL of the segment.
- The following curve Horizontal Left Curve (HLC), Horizontal Right Curve (HRC), Vertical Left Curve (VLC), Vertical Right Curve (VRC), Right Slope Right Curve (RR), Right Slope Left Curve (RLC), Left Slope Right Curve (LRC), Left Slope Left Curve (LRC) are defined by by the conditions:

if b∈H and IL>=2 and IR=0 then b∈HLC
if b∈V and IL>=2 and IR=0 then b∈VLC
if b∈R and IL>=2 and IR=0 then b∈RLC
if b∈L and IL>=2 and IR=0 then b∈LLC

if b∈H and IL=0 and IR>=2 then b∈HRC
if b∈V and IL=0 and IR>=2 then b∈VRC
if b∈R and IL=0 and IR>=2 then b∈RRC
if b∈L and IL=0 and IR>=2 then b∈LRC

where b - branch, IL - number of the left line intersections with segment, IR - number of the right line intersections with segment, H, V, R, L - kind of base line define by membership functions FH, FV, FL, FR.

(Calculate the size of the segment. The size of the segment is defined by the same way as at step 5 but size = sDX * sDY, Ms = m * n, where sDX, sDY is the width and the height of the bound box, m is the width of the character, n is the height of the character.

Step 7: If $j < Vni$ (there is another branch to check) go to step 2 else if $i < Vc$ (there is another vertex to start from) increase index of vertices ($i = i + 1$) and go to step 2. If $i >= Vc$ that means there is no vertex left and stop the algorithm.

Features Coding

This stage supposes preparing the initial data for the NN. Students are requested to learn and research some methods of feature coding when handwriting character recognising. The choice of the feature coding method is essentially influences onto the learn characteristics of the NN and the quality of recognition. Let us consider briefly one of the methods that we suggest to students. First of all we need to determine the maximal number of components because of fixed number of neurones in input layer. All features extracted from character are coded into the input vector. In this vector we can distinguish two parts. The first part includes a kind of segment, its orientation and size. The second part encodes the relationships between the components. For each pair of distinct components the segment relative position record is defined (with information about what direction has to go from the first to reach the second and whether the objects touch).

Multilayer Feed-Forward NN

The classic three-layer feed forward NN is used in the laboratory work. The activation function of every input unit is the linear function $f(x) = x$, and activation function of every hidden and output unit is the sigmoid function $f(f) = 1/(1 + e^{-x})$. In our case, the inputs of the neural NN will be components of the vector which represent a collection of segments, together with information about how the segments are related. To train a NN classifier, the preparing of a set of training data and test data is required.

The number of input nodes depends on the number of a maximal number of the components. The number of hidden nodes is a variable that can be adjusted by user but there are no special rules to do that. We will propose several tests to find convenient values for the number of hidden neurones.

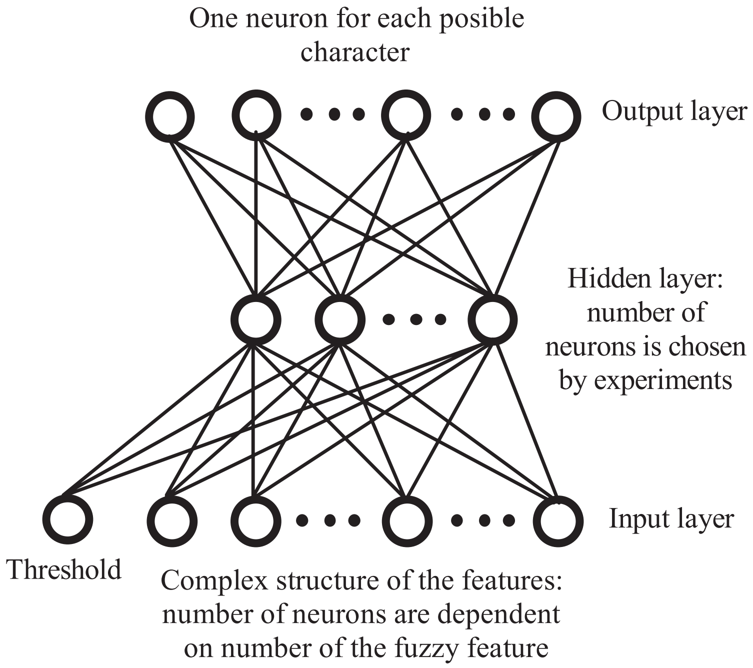

One neuron for each posible character
Output layer
Hidden layer: number of neurons is chosen by experiments
Input layer
Threshold
Complex structure of the features: number of neurons are dependent on number of the fuzzy feature

Fig. 9 The structure of the NN

To have the NN to perform the algorithm of the recognition we must adjust weight values in order to obtain a desired network performance. For this architecture, a back propagation training algorithm is convenient. We use standard Kohonen self-organisation feature map, which accurately represents the fuzziness of the character classes. Student can see the weight vectors of a map that trained using 24x18 binary images of characters.

(Source: Gilewski, J., Phillips, P., Yanushkevich, S. & Popel, D., Handwriting Recognition-Neural Networks-Fuzzy Logic, [Online] Available at: <http://users.du.se/~jwe/fuzzy/NFL/F12.PDF> [Accessed 20 July 2012]).

Questions

1. Name the steps included by pre-processing of the image.

   Answer

   Filtering, Thinning and Searching vertices are the steps included by pre-processing of the image.

2. What is the aim of Fuzzy Feature Extraction Algorithm?

   Answer

   The aim of this algorithm is to extract the defined segments from the character image.

3. Write short note on Features Coding.

   Answer

   This stage supposes preparing the initial data for the NN. Students are requested to learn and research some methods of feature coding when handwriting character recognising. The choice of the feature coding method is essentially influences onto the learn characteristics of the NN and the quality of recognition. Let us consider briefly one of the methods that we suggest to students. First of all we need to determine the maximal number of components because of fixed number of neurones in input layer. All features extracted from character are coded into the input vector. In this vector we can distinguish two parts. The first part includes a kind of segment, its orientation and size. The second part encodes the relationships between the components. For each pair of distinct components the segment relative position record is defined (with information about what direction has to go from the first to reach the second and whether the objects touch).

Case study II

Blue Eye Technology Using Artificial Intelligence

Human error is still one of the most frequent causes of catastrophe and ecological disasters. The main reason is that the monitoring systems concern only the state of the processes where as human contribution to the overall performance of the system is left unsupervised. Since the control instruments are automated to a large extent, a human operator becomes a passive observer of the supervised system, which results in weariness and vigilance drop.

Thus, he may not notice important changes of indications causing financial or ecological consequences and a threat to human life. It therefore is crucial to assure that the operator's conscious brain is involved in an active system supervising over the whole work time period. It is possible to measure indirectly the level of the operator's conscious brain involvement using eye motility analysis.

What is Blue Eyes ?

BLUE EYES is a technology, which aims at creating computational machines that have perceptual and sensory abilities like those of human beings. The basic idea behind this technology is to give computer human power. For example, we can understand humans' emotional state by his facial expressions. If we add these perceptual abilities to computers, we would enable them to work together with human beings as intimate partners. It provides technical means for monitoring and recording human-operator's physiological condition.

It doesn't predict nor interfere with operator's thoughts. Cannot force directly the operator to work How was the term blue-eyes coined. Blue in this term stands for Bluetooth, which enables reliable wireless communication and the Eyes because the eye movement enables us to obtain a lot of interesting and important information.

How are blue and eyes related As the idea is to monitor and record operator's basic physiological parameters, the most important physiological activity is the movement of eyes. For a computer to sense the eye movement, wiring between operator and the system is required.

But, this is a serious limitation of the operator's mobility and disables his operations in large control rooms. So utilisation of wireless technology becomes essential which can be implemented through blue tooth technology.

Need for Blue Eyes

Is it necessary to make computer function what a human brain does Yes, human error is still one of the most frequent causes of catastrophes (calamity) and ecological disasters, because human contribution to the overall performance of the system is left unsupervised. The control instruments within the machine have automated it to large extent, thus Human operator becomes a passive observer of the supervised system, resulting in weariness and vigilance drop, but the user needs to active.

Is it really needed that a human brain be active He may not notice important changes of indications causing financial or ecological consequences, which is a threat to human life. Thus, it's crucial that operator's brain is involved in an active system supervising over the whole work time period.

What can ee do with Blue Eyes Technology?

It has the ability to gather information about you and interact with you through special techniques like facial recognition, speech recognition, etc. It can even understand your emotions at the touch of the mouse. It can verify your identity, feel your presence, and start interacting with you.

The machine can understand what a user wants, where he is looking at, and even realise his physical or emotional states. It realises the urgency of the situation through the mouse. For instance if you ask the computer to dial to your friend at his office, it understands the situation and establishes a connection. It can reconstruct the course of operator's work.

Key Features of the System

Visual attention monitoring (eye motility analysis),
Physiological condition monitoring (pulse rate, blood oxygenation)
Operator's position detection (standing, lying)
Wireless data acquisition using Bluetooth technology
Real-time user-defined alarm triggering,
Physiological data, operator's voice and overall view of the control room recording recorded data playback.

How can we give computer the human power?

It uses non-obtrusive sensing method, employing most modern video cameras and microphones to identify the users' actions through the use of imparted sensory abilities. The blue eyes system checks the physiological parameters like eye movement, heart beat rate and blood oxygenation against abnormal and undesirable values and triggers user-defined alarms when necessary.

Blue eyes technology requires designing a personal area network linking all the operators and the supervising system. As the operator using his sight and hearing, senses the state of the controlled system, the supervising system will look after his physiological condition. The use of a miniature CMOS camera integrated into the eye movement sensor will enable the system to calculate the point of gaze and observe what the operator is actually looking at. Introducing voice recognition algorithm will facilitate the communication between the operator and the central system and simplify authorisation process.

Structure

Blue eyes system consists of a mobile measuring device called Data Acquisition Unit (DAU) and a central analytical system called Central System Unit (CSU).

Data Acquisition Unit

- It maintains Bluetooth connections.
- Gets information from the sensor.
- Sends information over the wireless connection.
- Delivers the alarm messages sent from the Central System Unit (CSU) to the operator.
- Handles personalised ID cards.

Jazz Multisensor:

- It's an eye movement sensor, to provide necessary physiological data in Data Acquisition Unit (DAU).
- It supplies raw digital data regarding eye position, the level of blood oxygenation acceleration along horizontal and vertical axes and ambient light intensity.
- Eye movement can be measured using direct infrared oculographic transducers.

Central System Unit

- Maintains blue tooth connections in the other side.
- Buffers incoming sensor data.
- Performs on-line data analysis.
- Records the conclusion for further exploration.
- Provides visualisation interface.

Types of Users

Users belong to three categories:

- Operators
- Supervisors
- System administrators

Operator

Operator is a person whose physiological parameters are supervised.
The operator wears the DAU.
The only functions offers to the operator are Authorisation to the system and receiving alarm alerts.

Authorisation: Operator has to enter his personal PIN into DAU, if PIN is accepted, authorisation is said to be complete.
Receiving Alerts: This function supplies the operator with the most important alerts about his and his co-workers' condition and mobile device state.

Supervisor

He is the person responsible for analyzing operators' condition and performance. The supervisor receives tools for inspecting present values of the parameters (on-line browsing) as well as browsing the results of the long-term analysis (off-line browsing).

System Administrator: He is the user that maintains the system. The administrator is delivered tools for adding new operators to the database. Defining alarm conditions, Configuring logging tools, Creating new analyzer modules

Administrative applications

Blue Eyes system can be applied in every working environment requiring permanent operator's attention.

- At power plant control rooms.
- At captain bridges.
- At flight control centres.
- Professional drivers.

Conclusion

The wireless link between the sensors worn by the operator and the supervising system makes it possible to improve overall reliability, safety and assures proper quality of system performance. These new possibilities can cover areas such as industry, transportation, military command centres or operation theatres. Researchers are attempting to add more capabilities to computers that will allow them to interact like humans, recognise human presents, talk, listen, or even guess their feelings. It avoids potential threats resulting from human errors, such as weariness, oversight, tiredness.

(Source: 2008. Blue Eyes - Monitoring Human - Operator System, [Online] Available at: <http://www.articlesbase.com/networks-articles/blue-eyes-monitoring-human-operator-system-420263.html> [Accessed 20 July 2012]).

Questions

1. Write a short note on Blue Eyes Technology.
2. What are the characteristics of Data Acquisition Unit and Jazz Multisensor?
3. Write the scope of Blue eye technology.

Case Study III

Offline Handwriting Recognition using Genetic Algorithm

Introduction

Handwriting recognition refers to the identification of written characters. The problem can be viewed as a classification problem where we need to identify the most appropriate character the given figure matches to. Offline character recognition refers to the recognition technique where the final figure is given to us. We have no idea of how the writer wrote the letter. This is contrary to the online character recognition systems where the data can be sampled while the character is being written.

An example of this is writing a character on a touch screen with a pointing device. Operating in offline mode gives as input the complete picture of character that we need to recognise. The complexity of the recognition is usually associated with the size of the language being considered. If the language contains more number of characters, the identification would be much more difficult than the case when the language contains lesser number of characters.

Similarly we need to consider how the various characters are written and the differences between the various characters. They always have an effect on the performance of the handwriting recognition system. In this paper we propose the use of Genetic Algorithms for solving this problem. The basic idea of genetic algorithm comes from the fact that it can be used as an excellent means of combining various styles of writing a character and generating new styles. Closely observing the capability of human mind in the recognition of handwriting, we find that humans are able to recognise characters even though they might be seeing that style for the first time. This is possible because of their power to visualise parts of the known styles into the unknown character.

Algorithm

In this section we will take a deep insight into the algorithm and its working. We discuss about the handwriting recognition general procedure, the algorithmic assumptions and its working. We know that we are given an unknown character that needs to be recognised. For this we have diverse form of training data available for each and every character. In this algorithm we try to match the input to the training data and the data generated from intermixing of training data, to find the best match for the given input data.

General Procedure

Handwriting recognition is a famous problem which involves the recognition of whatever input is given in form of image, scanned paper, etc. The handwriting recognition generally involves the following steps:

Segmentation

This step deals with the breaking of the lines, words and finally getting all the characters separated. This step involves the identification of the boundaries of the character and separating them for further processing. In this algorithm we assume that this step is already done. Hence the input to our system is a single character.

Pre-processing

This step involves the initial processing of the image, so that it can be used as an input for the recognition system. In this algorithm we assume that a part of this step has been done. We assume that the character segmented is made thin to a unit pixel thickness. Various algorithms may be used for this purpose. The further processing is done by our algorithm.

Recognition

Once the input image is available in good condition, it may be processed for recognition. The role of the recognition system is to identify the character. Our algorithm uses an image as an input for the same.

Procedure

Once the prerequisites are met, the image input is given to the system. This is then recognised by the algorithm. The algorithm is as given below:

Handwriting Recognition (Language, Training Data, Input Image)

```
Step1: For every character c in language
Step2: For every input i for the character c in test data
Step3: Generate Graph gci of i
Step4: Generate graph t of input image
Step5: For every character c in language
Step6: Use Genetic Algorithm to generate hybrid graphs
Step7: Return character corresponding to graph with the minimum most fitness function (out of the graphs generated
       in any genetic operation)
```

Seeing the previous algorithm it is clear that we first need to generate graphs and then used genetic algorithm to mix these graphs and find the most optimal solution.

Generation of Graph: This algorithm takes as input an image, and returns the graph of the same. The whole procedure of the algorithm requires the principles of graph theory and coordinate geometry. The algorithm is given in figure 2.

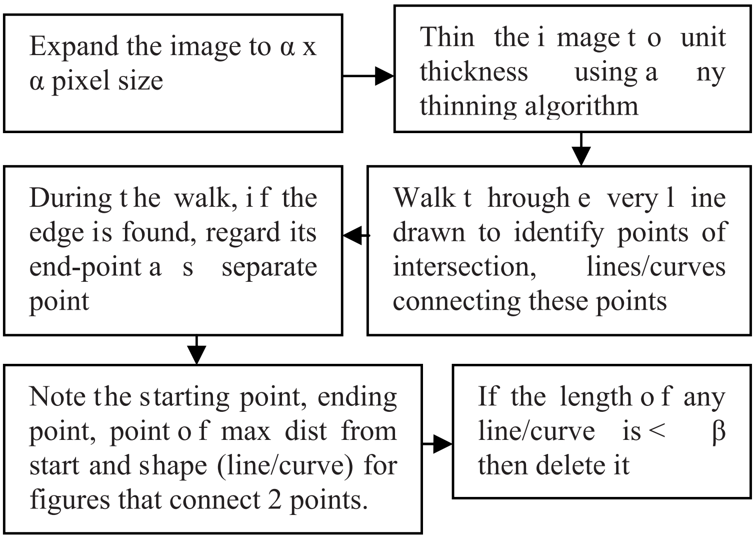

Expand the image to α x α pixel size
Thin the image to unit thickness using any thinning algorithm
Walk through every line drawn to identify points of intersection, lines/curves connecting these points
During the walk, if the edge is found, regard its end-point as separate point
Note the starting point, ending point, point of max dist from start and shape (line/curve) for figures that connect 2 points.
If the length of any line/curve is < β then delete it

Fig. 1 The graph generation algorithm

Here a graph represents the vertices and the edges. The edges are the lines or curves connecting any 2 points. Every point where an edge ends/starts is regarded as a vertex. We are also interested in knowing the point for every line/curve, which is at the maximum distance from the start point. This is useful when the graph may contain a closed curve E.g. O would be regarded as a curve from a vertex that ends at the same vertex.

Each edge must hence represent the start vertex, end vertex, shape (line/curve) and the point of maximum separation from start vertex. The image expansion is done by calculating the expansion factor (final image size/initial image size), for both the x and y coordinates. The start and end coordinate of each pixel in the new image are then measured by multiplying with the expansion factor. The lines and curves are differentiated from the maximum and minimum angle subtended by the start of the line/curve, a point situated γ units further from the start and all points 2γ units from the start.

For a line the difference between the maximum and minimum angle must be almost 0 degrees. Law of cosines is used for the purpose of finding angles. Edges are detected by using a similar logic. Figure 4 shows the graph generated when the input was J.

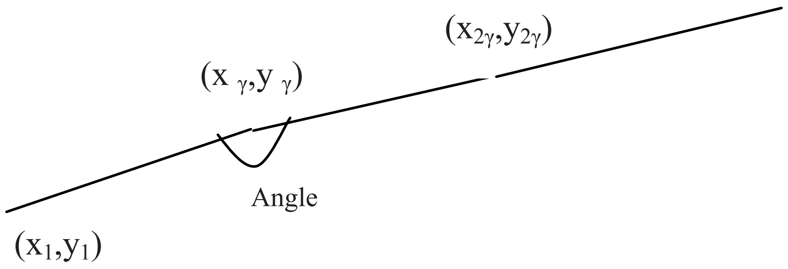


Fig. 2 Difference between curve and line

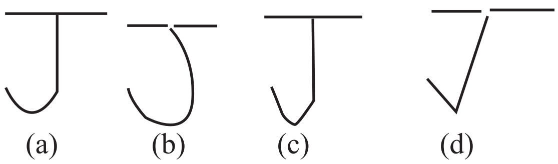


Fig. 3 Genetic algorithm

Genetic algorithms are a very good means of optimisations in such problems. They optimise the desired property by generating hybrid solutions from the presently existing solutions. These hybrid solutions are added to the solution pool and may be used to generate more hybrids. These solutions may be better than the solutions already generated. All this is done by the genetic operators, which are defined and applied over the problem. We already have a set of graphs generated from training data for any character. The use of genetic algorithm is to mix 2 such graphs and to generate new graphs. These newly generated graphs may happen to match the character better than the existing graphs. Hence genetic algorithms are a good means of optimisations. We discuss each of the points in detail in the coming sections.

Fitness Function

In Genetic Algorithms, the fitness function is used to test the goodness of the solution. This function, when applied on any of the solution from the solution pool, tells the level of goodness. In our problem, we have used fitness function to measure the deviation of the graph of the solution, to that of the unknown input. If the two graphs are very similar, the deviation would be low and hence the value of the fitness function would be low. The lower the value of the fitness function, the better would be the matching. Hence the graph with the lowest value of fitness function would be the most probable answer. We first devise a formula to find the deviation between any two edges. This would be then used as a means of finding the deviation between two graphs.

Deviation between Two Edges

For finding out the deviation between two edges, we first define a function D (e1,e2) that finds the deviation between any edges of a graph (e1) with any edge of the other graph (e2). Here an edge may represent a line or a curve. But the start point of the edge e1 may match with the start point of the edge of e2 and the end point of e1 may match with the end point of e2. It may also be possible for the converse to be true. The start point of e1 may match with the end point of e2 and the end point of e1 may match with the start point of e2. This is shown in figure 4(a)-(c). Hence we calculate the deviation using two separate cases (D1 and D2) and the minimum of the two is the actual deviation. D1 represents the case where the start vertex of e1 matches with the start vertex of e2. The end vertex of e1 matched with the end vertex of e2.

In general D1(e1,e2)=square of distance of start points of e1 and e2 + square of distance of end points of e1 and e2 If however, e1 is a line and e2 is a curve or vice versa, an overhead cost of η is added. If both e1 and e2 are curves, and start and end points of e1 are less than β units apart (it is almost circle), then we take point of maximum distance in place of end points in the above formula. Here point of maximum distance is the point in the curve which is at maximum distance from the start point of the curve. This is shown in Figure 4(a) and Figure 4(b).

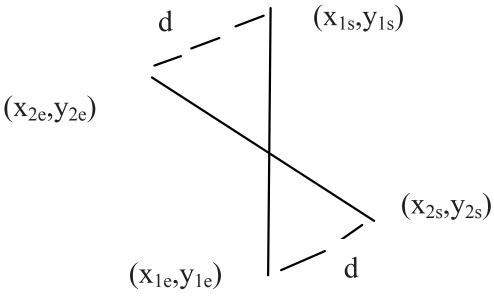


Fig. 4(a): Calculating D1 with 2 lines

D2 represents the case where the start vertex of e1 matches with the end vertex of e2. The end vertex of e1 matched with the start vertex of e2. Similarly we calculate D2(e1,e2) by the following formula D2(e1,e2)=square of distance of start point of e1 and end point of e2 + square of distance of end point of e1 and start point of e2.

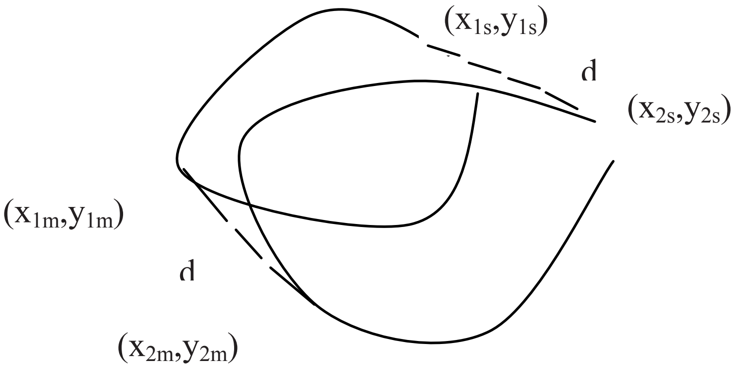


Fig. 4(b): Calculating D1 with 2 curves

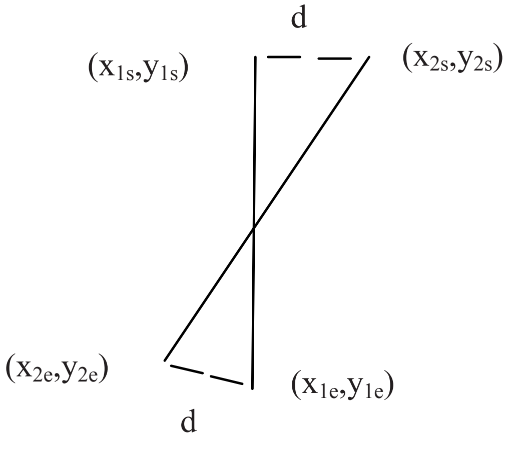


Fig. 4(c): Calculating D2

Other specifications remain same as used in calculating D1 (e1, e2). This is shown in Figure 4(c). The deviation between two edges is calculated by the following formula: D (e1,e2)=min{D1(e1,e2), D2 (e1,e2)} We even generalise the formula to the condition when either e1 or e2 is null. This means that we can find the deviation of a line or a curve with nothing. This is a feature useful in finding graph deviation when there is unequal number of edges in two graphs. In such cases the formula is:

D(e1,null) or D(null,e1) = Distance between the starting point and end point of line/curve.

If e1 is curve, and start and end points of e1 are less than β units apart (it is almost circle), then we take point of maximum distance in place of end points in the above formula. Here point of maximum distance is the point in the curve which is at maximum distance from the start point of the curve. Suppose that the edge e1 in first graph has start points as (x1s, y1s) and end points as (x1e, y1e).

Similarly suppose that the edge e2 in second graph has start points as (x2s, y2s) and end points as (x2e, y2e). The point of maximum distance of e1 is (x1m, y1m) and e2 is (x2m, y2m). The formula explained above can also be stated as:

D1 (e1,e2) = (x1s-x2s)2+(y1s-y2s)2+(x1e-x2e)2+(y1e-y2e)2 if (e1 is line and e2 is line) or (e1 is curve and e2 is curve and the distance between (x1s,y1s) and (x1e,y1e) is less than β units).

D1 (e1, e2) = (x1s-x2s)2+(y1s-y2s)2+(x1e-x2e)2+(y1e-y2e)2 + η if (e1 is line and e2 is curve) or (e1 is curve and e2 is line).

D1(e1,e2)= (x1s-x2s)2+(y1s-y2s)2+(x1m-x2m)2+(y1m-y2m)2 if (e1 is curve and e2 is curve and the distance between (x1s,y1s) and (x1e,y1e) is less than β units).

D2(e1,e2) = (x1s-x2e)2+(y1s-y2e)2+(x1e-x2s)2+(y1e-y2s)2 if (e1 is line and e2 is line) or (e1 is curve and e2 is curve and the distance between (x1s,y1s) and (x1e,y1e) is more than β units).

D2 (e1, e2) = (x1s-x2e)2+(y1s-y2e)2+(x1e-x2s)2+(y1e-y2s)2 + η if (e1 is line and e2 is curve) or (e1 is curve and e2 is line).

D2 (e1,e2) = (x1s-x2m)2+(y1s-y2m)2+(x1m-x2s)2+(y1m-y2s)2 if (e1 is curve and e2 is curve and the distance between (x1s,y1s) and (x1e,y1e) is less than β units).

D (e1, e2) =min{D1(e1,e2),D2(e1,e2)} if both e1 and e2 are not null

D (e1, e2) = (x1s-x1e)2+(y1s-y1e)2 if e2 is null

D (e1, e2) = (x2s-x2e)2+(y2s-y2e)2 if e1 is null

Deviation of a graph

Once we know the deviation of an edge with another edge, the deviation of a graph can be found out easily. This deviation is found out by pairing up of edges and iterating through all the edges. The algorithm is as given below.

```
Deviation (G1,G2)
Step1: dev ← 0
Step2: While G1 has no edges or G2 has no edges
Step3: Find the edges e1 from first graph and e2 from second graph such that deviation between e1 and e2 is the minimum for any pair of e1 and e2.
Step4: Add its deviation to dev
Step5: Remove e1 from first graph and e2 from second graph
Step6: For all edges e1 left in first graph
```

```
Step7: Add deviation of e1 and null to dev
Step8: For all edges e2 left in second graph
Step9: Add deviation of null and e2 to dev
Step10: Return dev
```

Here we see that we try to minimise the total deviation. For this we select the pair of edges, one from each graph such that their deviation is minimal. We keep selecting such pairs till one graph gets empty. Hence we keep proceeding by keeping the total deviation minimum. In the end we add deviation of all the left edges or curves. This way we find the minimum deviation between the two graphs.

(Source: Kala, R., Vazirani, H., Shukla, A. & Tiwari, R., Offline Handwriting Recognition using Genetic Algorithm, [Online] Available at: <http://www.ijcsi.org/papers/7-2-1-16-25.pdf> [Accessed 20 July 2012])

Questions

1. What does handwriting recognition refers?
2. Write a short note on fitness function.
3. Write Handwriting Recognition algorithm.

Bibliography

References

- 2007. *The Next Generation of Neural Networks*, [Video Online] Available at: <http://www.youtube.com/watch?v=AyzOUbkUf3M&feature=results_main&playnext=1&list=PLA47195C498F5DD59> [Accessed 18 July 2012].
- 2008. *Fuzzy Logic Why and What 1*, Video Online] Available at: <http://www.youtube.com/watch?v=YM4PVm5HDDY> [Accessed 19 July 2012].
- 2009. *Intro to Neural Networks*, [Video Online] Available at: <http://www.youtube.com/watch?v=DG5-UyRBQD4> [Accessed 18 July 2012].
- 2010. *Fuzzy Sets Example*, [Video Online] Available at: <http://www.youtube.com/watch?v=5TqyJGfgwbw> [Accessed 19 July 2012].
- 2011. *Introduction to Genetic Algorithms*, [Video Online] Available at: <http://www.youtube.com/watch?v=zwYV11a__HQ> [Accessed 19 July 2012].
- 2011. *Lecture 05, part 1 | Pattern Recognition*, [Video Online] Available at: <http://www.youtube.com/watch?v=ZeiJmSHx6qI> [Accessed 18 July 2012].
- 2011. *Mod-01 Lec-38 Genetic Algorithms*, [Video Online] Available at: <http://www.youtube.com/watch?v=Z_8MpZeMdD4> [Accessed 19 July 2012].
- 2012. *Lecture 10 - Neural Networks (May 3, 2012)*, [Video Online] Available at: <http://www.youtube.com/watch?v=Ih5Mr93E-2c> [Accessed 19 July 2012].
- 2012. *Neural network tutorial: The back-propagation algorithm (Part 2)*, [Video Online] Available at: <http://www.youtube.com/watch?v=zpykfC4VnpM&feature=related> [Accessed 19 July 2012].
- Abraham, A., *Artificial Neural Networks*, [Online] Available at: <http://www.softcomputing.net/ann_chapter.pdf> [Accessed 18 July 2012].
- *AN INTRODUCTION TO GENETIC ALGORITHMS*, [Online] Available at: <http://www.boente.eti.br/fuzzy/ebook-fuzzy-mitchell.pdf> [Accessed 19 July 2012].
- 2012. *Artificial Intelligence Lecture No. 1* [Video Online] Available at: <http://www.youtube.com/watch?v=katiy95_mxo>[Accessed 20 July 2012].
- *Artificial Intelligence*, [Online] Available at: <http://artint.info/html/ArtInt_182.html>[Accessed 20 July 2012].
- Bobrow, D. G., *Artificial Intelligence*, [Online] Available at: <http://www2.parc.com/istl/groups/hdi/papers/stefik-molgen-part-1.pdf>[Accessed 20 July 2012].
- Bollobás, B., Kozma, R. & Miklós, D., 2009. *Handbook of Large-Scale Random Networks*, Springer.
- Bower, A., *SOFT COMPUTING*, [Online] Available at: <http://www.tessella.com/wp-content/uploads/2008/05/softcomputing.pdf> [Accessed 18 July 2012].
- Braspenning, J. P., Thuijsman F. & Weijters, A. J. M. M., 1995. *Artificial Neural Networks: An Introduction To Ann Theory And Practice*, Springer.
- Chakraborty, C. R., *Introduction Basics of Soft Computing*, [Online] Available at: <http://www.myreaders.info/01_Introduction_to_Soft_Computing.pdf> [Accessed 18 July 2012].
- *Chapter 3 GENETIC ALGORITHMS*, [Online] Available at: <http://sci2s.ugr.es/docencia/metaheuristicas/GeneticAlgorithms.pdf> [Accessed 19 July 2012].
- Chaturvedi, K. D., 2008. *Soft Computing: Techniques and Its Applications in Electrical Engineering*, Springer.
- Cheung, V. & Cannons, K., *An Introduction to Neural Networks*, [Online] Available at: <http://www2.econ.iastate.edu/tesfatsi/NeuralNetworks.CheungCannonNotes.pdf> [Accessed 18 July 2012].
- Dreyfus, G., 2005. *Neural Networks: Methodology And Applications*, Birkhäuser.

- *Fuzzy Logic*, [Online] Available at: <http://page.mi.fu-berlin.de/rojas/neural/chapter/K11.pdf> [Accessed 19 July 2012].
- Ganesh, M., 2006. *Introduction to Fuzzy Sets And Fuzzy Logic*, PHI Learning Pvt. Ltd.
- Goldberg, *Genetic Algorithms*, Pearson Education India.
- Gurney, K., Gurney, N. K., 1997. *An Introduction to Neural Networks*, UCL Press.
- Haugeland, J., 1989. *Artificial Intelligence: The Very Idea*, MIT Press.
- Horvath, S., 2011. *Weighted Network Analysis: Applications in Genomics and Systems Biology*, Springer.
- Kala, R., 2012. *Soft Computing Lecture - Hour 1*, [Video Online] Available at: <http://www.youtube.com/watch?v=DzidvDNE-Ik> [Accessed 18 July 2012].
- Kala, R., 2012. *Soft Computing Lecture - Hour 2*, [Video Online] Available at: <http://www.youtube.com/watch?v=O5kZOqw_tfc&feature=relmfu> [Accessed 18 July 2012].
- Karayiannis, B. N. &Venetsanopoulos, N. A., 1992. *Artificial Neural Networks: Learning Algorithms, Performance Evaluation, and Applications*, Springer.
- Klir, J. G., Yuan, B., 1995. *Fuzzy sets and fuzzy logic: Theory and Applications*, Prentice Hall PTR.
- Konar, A., 2000. *Artificial Intelligence and Soft Computing: Behavioral and Cognitive Modeling of the Human Brain*, Volume 1, CRC Press.
- *Lecture - 1 Introduction to Artificial Intelligence*, 2008. [Video Online] Available at: <http://www.youtube.com/watch?v=fV2k2ivttL0>[Accessed 20 July 2012].
- Man, F. K., Tang, S. K. & Kwong, S., 1999. *Genetic Algorithms: Concepts and Designs*, 2nd ed. Springer.
- Newman, J. E. M., *Analysis of weighted networks*, [Online] Available at: <http://arxiv.org/pdf/condmat/0407503.pdf> [Accessed 19 July 2012].
- Prof. Hamprecht, F., 2011. *Lecture 05, part 1 | Pattern Recognition*, [Video Online] Available at: <http://www.youtube.com/watch?v=ZeiJmSHx6qI&feature=endscreen&NR=1> [Accessed 18 July 2012].
- Prof. Sarkar, S. & Prof. Basu, A., 2008. *Lecture - 30 Fuzzy Reasoning – I,* [Video Online] Available at: <http://www.youtube.com/watch?v=p-9G7mEF9D4> [Accessed 18 July 2012].
- Prof. Sengupta, S., 2009. *Lec-1 Introduction to Artificial Neural Networks*, [Video Online] Available at: <http://www.youtube.com/watch?v=xbYgKoG4x2g> [Accessed 18 July 2012].
- Prof. Smith, L., *An Introduction to Neural Networks*, [Online] Available at: <http://www.cs.stir.ac.uk/~lss/NNIntro/InvSlides.html> [Accessed 18 July 2012].
- Russell, 2003. *Artificial Intelligence: A Modern Approach*, 2nd ed., Pearson Education India.
- Serrano-Gotarredona, T., Linares-Barranco, B. & Andreou, G. A., 1998. *Adaptive Resonance Theory Microchips*, Springer.
- Tanaka, T. & Weitzenfeld1, A., 8 *Adaptive Resonance Theory*, [Online] Available at: <http://www.ica.luz.ve/dfinol/NeuroCienciaCognitiva/NeuralNetworkModels/ART%20INTRO%20-%20157_170.CH08.pdf> [Accessed 18 July 2012].
- *The Biological Paradigm*, [Online] Available at: <http://page.mi.fu-berlin.de/rojas/neural/chapter/K1.pdf> [Accessed 18 July 2012].
- Watkins, T., Valley, S., & Alley, T., *Fuzzy Logic: The Logic of Fuzzy Sets*, [Online] Available at: <http://www.sjsu.edu/faculty/watkins/fuzzysets.htm> [Accessed 19 July 2012].
- *Weighted Networks – The Perceptron*, [Online] Available at: <http://page.mi.fu-berlin.de/rojas/neural/chapter/K3.pdf> [Accessed 19 July 2012].
- Yegnanarayana, B., 2004. *Artificial Neural Networks*, PHI Learning Pvt. Ltd.
- Zacharie, M., *Adaptive Resonance Theory 1* (ART1) Neural Network Based Horizontal and Vertical Classification of 0-9 Digits Recognition, [Online] Available at: <citeseerx.ist.psu.edu/viewdoc/download?doi=10.1.1.153...> [Accessed 18 July 2012].

Recommended Reading

- Akerkar, R. & Sajja, P., 2010. *Knowledge-Based Systems*, Jones & Bartlett Learning.
- Bojadziev, G. & Bojadziev, M., 1995. *Fuzzy Sets, Fuzzy Logic, Applications*, World Scientific.
- Buckley, J. J. & Eslami, E., 2002. *An Introduction to Fuzzy Logic and Fuzzy Sets*, Springer.
- Carpenter, A. G. & Grossberg, S., *Adaptive Resonance Theory*, Boston University.
- Freeman, 1991. *Neural Networks: Algorithms, Applications, And Programming Techniques*, Pearson Education India.
- Gupta, M. M., 2002. *Soft Computing and Intelligent Systems: Theory and Applications*, Elsevier.
- Haupt, L. R. & Haupt, E. S., 2004. *Practical Genetic Algorithms*, 2nd ed. John Wiley & Sons.
- Haykin, S. S., 1999. *Neural Networks: A Comprehensive Foundation*, 2nd ed. Prentice Hall.
- Jackson, P. C., 1985. *Introduction to Artificial Intelligence*, 2nd ed., Dover Publications.
- Jones, M. T., 2008. *Artificial Intelligence: A Systems Approach (Computer Science Series)*, 1st ed., Jones and Bartlett Publishers.
- Kussul, E., Baidyk, T. & Wunsch, C. D., 2009. *Neural Networks and Micromechanics*, Springer.
- Mehrotra, K., Mohan, K. C. & Ranka, S., 1997. *Elements of Artificial Neural Networks*, 2nd ed. MIT Press.
- Neruda, R. & Koutník, J., 2008. *Artificial Neural Networks - ICANN 2008: 18th International Conference, Prague, Czech Republic, September 3-6, 2008, Proceedings*, Springer.
- Priddy, L. K. & Keller, E. P., 2005. *Artificial Neural Networks: An Introduction*, SPIE Press.
- Rojas, R., 1996. *Neural Networks: A Systematic Introduction*, Springer.
- Russell, S. & Norvig, P., 2009. *Artificial Intelligence: A Modern Approach*, 3rd ed., Prentice Hall.
- Saddington, P., 2002. *Adaptive Resonance Theory: Theory and Application to Synthetic Aperture Radar*, University of Surrey.
- Sanchez, E., Shibata, T. & Zadeh, A. L., 1997. *Genetic Algorithms and Fuzzy Logic Systems: Soft Computing Perspectives*, World Scientific.
- Schalkoff, *Artificial Neural Networks*, Tata McGraw-Hill Education.
- Sivanandam, N. S. & Deepa, N. S., 2010. *Introduction to Genetic Algorithms*, Springer.
- Tosh, C. & Ruxton, D. G., 2010. *Modelling Perception with Artificial Neural Networks*, Cambridge University Press.
- Varshney, M. A. R., *An Introduction to Soft Computing*, A. B. Publication.
- Webb, R. A., Copsey, D. K. & Cawley, G., 2011. *Statistical Pattern Recognition*, 3rd ed. John Wiley & Sons.
- Zadeh, A. L., Klir, J. G. & Yuan, B., 1996. *Fuzzy Sets, Fuzzy Logic, and Fuzzy Systems: Selected Papers*, World Scientific.

Self Assessment Answers

Chapter I

1. a
2. b
3. c
4. b
5. d
6. c
7. b
8. a
9. c
10. b

Chapter II

1. b
1. a
2. a
3. d
4. c
5. a
6. c
7. b
8. a
9. d
10. d

Chapter III

1. b
2. a
3. a
4. c
5. d
6. a
7. b
8. c
9. a
10. d

Chapter IV

1. a
2. b
3. d
4. c
5. b
6. d
7. c
8. a
9. b
10. a

Chapter V

1. a
2. d
3. b
4. c
5. a
6. a
7. d
8. b
9. c
10. a

Chapter VI

1. a
2. c
3. c
4. b
5. b
6. d
7. a
8. a
9. b
10. d

Chapter VII

1. a
2. b
3. a
4. d
5. b
6. c
7. a
8. c
9. a
10. d

Chapter VIII

1. a
2. d
3. b
4. c
5. a
6. a
7. b
8. c
9. b
10. d