CVR COLLEGE OF ENGINEERING

An UGC Autonomous Institution - Affiliated to JNTUH

Handout – 1

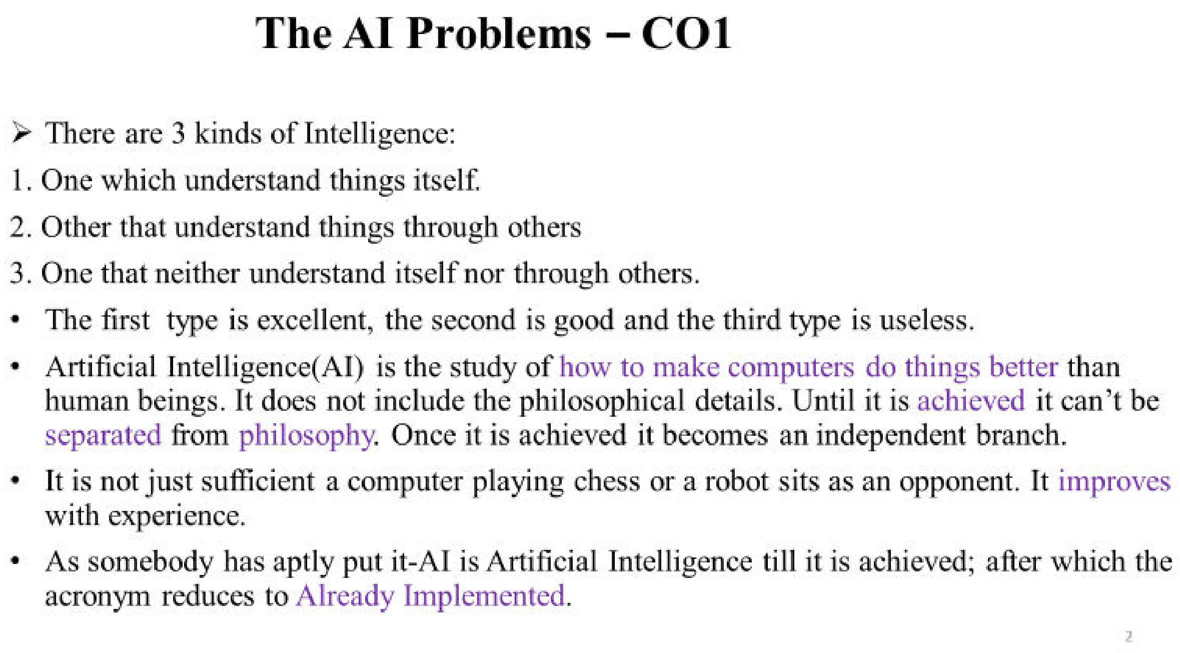
Unit - 1

Year and Semester: IVyr &II Sem

Subject**: Soft Computing**

Branch: **CSE**

Faculty: **Dr.Md.Yusuf Mulge**, Professor (CSE)



The AI Problems – CO1

- There are 3 kinds of Intelligence:

1. One which understand things itself.
2. Other that understand things through others
3. One that neither understand itself nor through others.

- The first type is excellent, the second is good and the third type is useless.
- Artificial Intelligence(AI) is the study of how to make computers do things better than human beings. It does not include the philosophical details. Until it is achieved it can't be separated from philosophy. Once it is achieved it becomes an independent branch.
- It is not just sufficient a computer playing chess or a robot sits as an opponent. It improves with experience.
- As somebody has aptly put it-AI is Artificial Intelligence till it is achieved; after which the acronym reduces to Already Implemented.

The AI Problems

- Much of early work in the field of AI focused on tasks, such as game playing and theorem proving.
- Samuel wrote a checker-playing program that not only played games with opponents but also used its experience to improve its performance at later stage.
- Chess also received a good deal of attention.
- It was able to prove several theorems from Russel’s Principia Mathematica.
- Game playing, theorem proving etc..display intelligence.
- Computers are fast in exploring large sets solutions and selecting the best one.

- Another area of AI problems is common sense reasoning for ex: An object can be at a single place at a time.
- If you let an item to fall on floor, then it may fall and break.
- To investigate this sort of reasoning, Newell, Shaw and Simon built the General Problem Solver(GPS), which they applied to several commonsense tasks as well problems with symbolic manipulations of logical expressions.
- As the research in AI progressed problems in the fields of perception(vision and speech), natural language understanding, and problems in specialized domains such as medical diagnosis.

A person who knows how to perform tasks from several of the categories shown in the figure learns the necessary skills in a standard order. First, perceptual, linguistic, and commonsense skills are learned. Later (and of course for some people, never) expert skills such as engineering, medicine, or finance are acquired. It might seem to make sense then that the earlier skills are easier and thus more amenable to computerized duplication than are the later, more specialized ones. For this reason, much of the initial AI work was concentrated in those early areas. But it turns out that this naive assumption is not right. Although expert skills require knowledge that many of us do not have, they often require much less knowledge than do the more mundane skills and that knowledge is usually easier to represent and deal with inside programs.

Mundane Tasks

- Perception
  - Vision
  - Speech
- Natural language
  - Understanding
  - Generation
  - Translation
- Commonsense reasoning
- Robot control

Formal Tasks

- Games
  - Chess
  - Backgammon
  - Checkers -Go
- Mathematics
  - Geometry
  - Logic
  - Integral calculus
  - Proving properties of programs

Expert Tasks

- Engineering
  - Design
  - Fault finding
  - Manufacturing planning
- Scientific analysis
- Medical diagnosis
- Financial analysis

Fig. 1.1 *Some of the Task Domains of Artificial Intelligence*

As a result, the problem areas where AI is now flourishing most as a practical discipline (as opposed to a purely research one) are primarily the domains that require only specialized expertise without the assistance

What is an AI Technique

- AI problems span a broad spectrum. They have very little in common and they are hard to solve.

Are there any techniques to solve these variety of problems?

- Answer to this question is yes.
- Three decades research on AI shows that the techniques used to solve AI problems exploits knowledge and knowledge possesses some less desirable properties, including:

- It is voluminous.
- It is hard to characterize accurately.
- It is constantly changing.
- It differs from data by being organized in a way that corresponds to the ways it will be used.

So then what is AI technique?

AI technique is a method that exploits knowledge that should be represented in such a way that:

- Knowledge captures generalizations. In other words, it is not necessary to represent separately each individual situation. Instead, situations sharing important properties are grouped together. If this is not the case then it is data.
- It can be understood by the people who are providing it. Although in many situations bulk of the data is obtained from variety of instruments.
- It can easily be modified to correct errors and to reflect changes in he world.
- It can be used in many situations even though it is not totally accurate or complete
- It can be used to overcome its own bulk by narrowing the range of possibility that must usually be considered.

There is some degree of independence between problems and problem solving techniques:

- It is possible to solve AI problems without using AI techniques. And it is possible to apply AI techniques to non-AI problems(Although the results may not be encouraging.)

Tic-Tac-Toe

A typical AI problem is Tic-Tac-Toe

Data structures

Board: A nine-element vector representing the board, where the elements of the vector correspond to the board position as follows:

1	2	3
4	5	6
7	8	9

An element contains the value 0 if the corresponding square is blank. 1 if it is filled with an X, 2 if it is filled with an O.

Movetable : A large vector of 19,683 elements (3^9), each element of which is a nine-element vector. The contents of this vector are chosen specifically to allow the algorithm to work.

Problems, problem spaces, and search – CO1

In previous chapter we have discussed the techniques to solve AI problems. To build a system to solve a particular problem. We need to do four things:

1. Define the problem precisely. This definition clearly specify the initial and final situations.
2. Analyze the problem. A few very important features can have an immense impact on problem solving techniques.
3. Isolate and represent the knowledge that is necessary to solve the problem.
4. Choose the best problem-solving technique(s) and apply it (them) to the particular problem.

In this chapter we discuss first two issues and in the next chapter we analyze the last two issues.

Defining the problem as a state space search

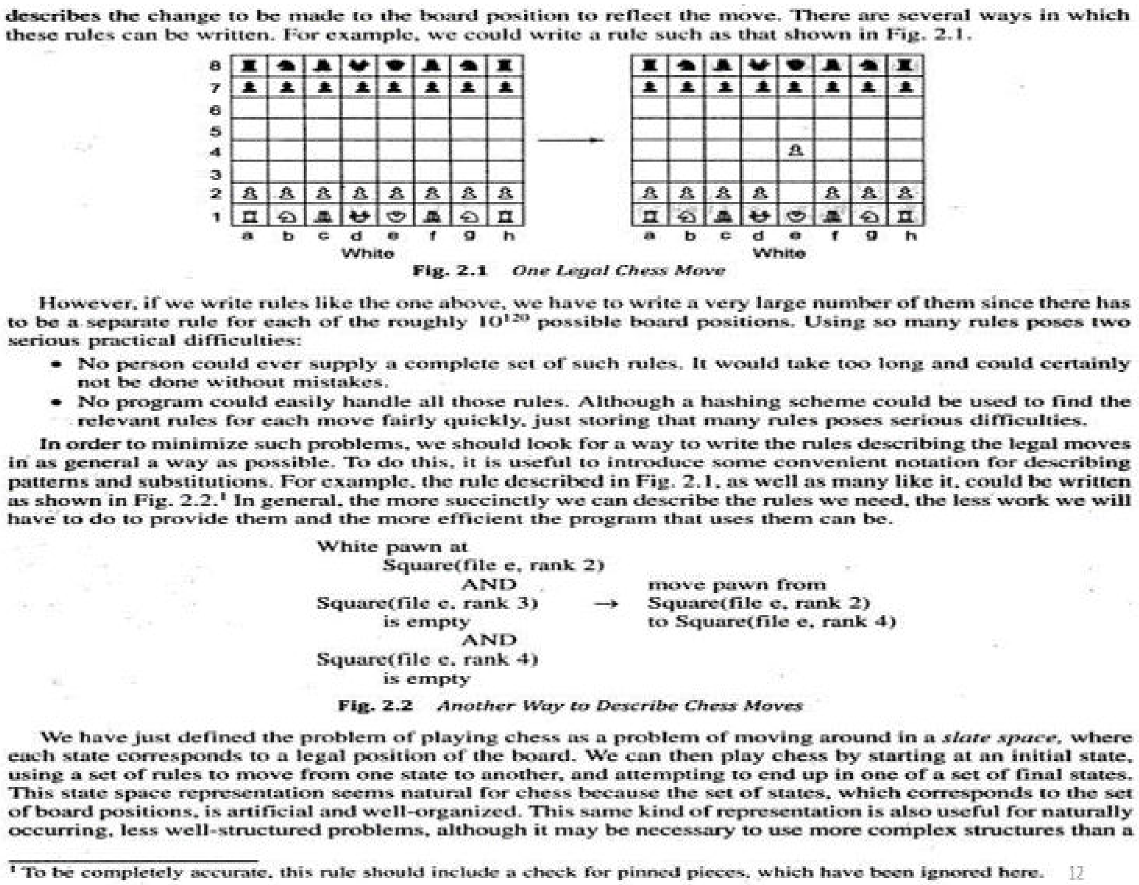
- Suppose we want to play chess.
- Here we have to specify the initial position, the rules that define the legal move, and the board position that represent a win.

Initial position – an 8x8 array where each position is occupied by a symbol.

One legal chess move is shown in the following figure

A winning situation is any board position where the opponent does not have a legal move and his/her king is under attack.

- The legal moves provide the way of getting from the initial state to a goal state. They can be described as a set of rules consisting of two parts: a left side that serves as a pattern to be matched against the current board position and a right side that describes the change to be made to the board position to reflect the move.



describes the change to be made to the board position to reflect the move. There are several ways in which these rules can be written. For example, we could write a rule such as that shown in Fig. 2.1.

8
7
6
5
4
3
2
1
a b c d e f g h
White
a b c d e f g h
White

Fig. 2.1 *One Legal Chess Move*

However, if we write rules like the one above, we have to write a very large number of them since there has to be a separate rule for each of the roughly 10^{120} possible board positions. Using so many rules poses two serious practical difficulties:

- No person could ever supply a complete set of such rules. It would take too long and could certainly not be done without mistakes.
- No program could easily handle all those rules. Although a hashing scheme could be used to find the relevant rules for each move fairly quickly, just storing that many rules poses serious difficulties.

In order to minimize such problems, we should look for a way to write the rules describing the legal moves in as general a way as possible. To do this, it is useful to introduce some convenient notation for describing patterns and substitutions. For example, the rule described in Fig. 2.1, as well as many like it, could be written as shown in Fig. 2.2.[1] In general, the more succinctly we can describe the rules we need, the less work we will have to do to provide them and the more efficient the program that uses them can be.

```
White pawn at
        Square(file e, rank 2)
                  AND                  move pawn from
Square(file e, rank 3)        →        Square(file e, rank 2)
        is empty                       to Square(file e, rank 4)
                  AND
Square(file e, rank 4)
        is empty
```

Fig. 2.2 *Another Way to Describe Chess Moves*

We have just defined the problem of playing chess as a problem of moving around in a *state space*, where each state corresponds to a legal position of the board. We can then play chess by starting at an initial state, using a set of rules to move from one state to another, and attempting to end up in one of a set of final states. This state space representation seems natural for chess because the set of states, which corresponds to the set of board positions, is artificial and well-organized. This same kind of representation is also useful for naturally occurring, less well-structured problems, although it may be necessary to use more complex structures than a

[1] To be completely accurate, this rule should include a check for pinned pieces, which have been ignored here.

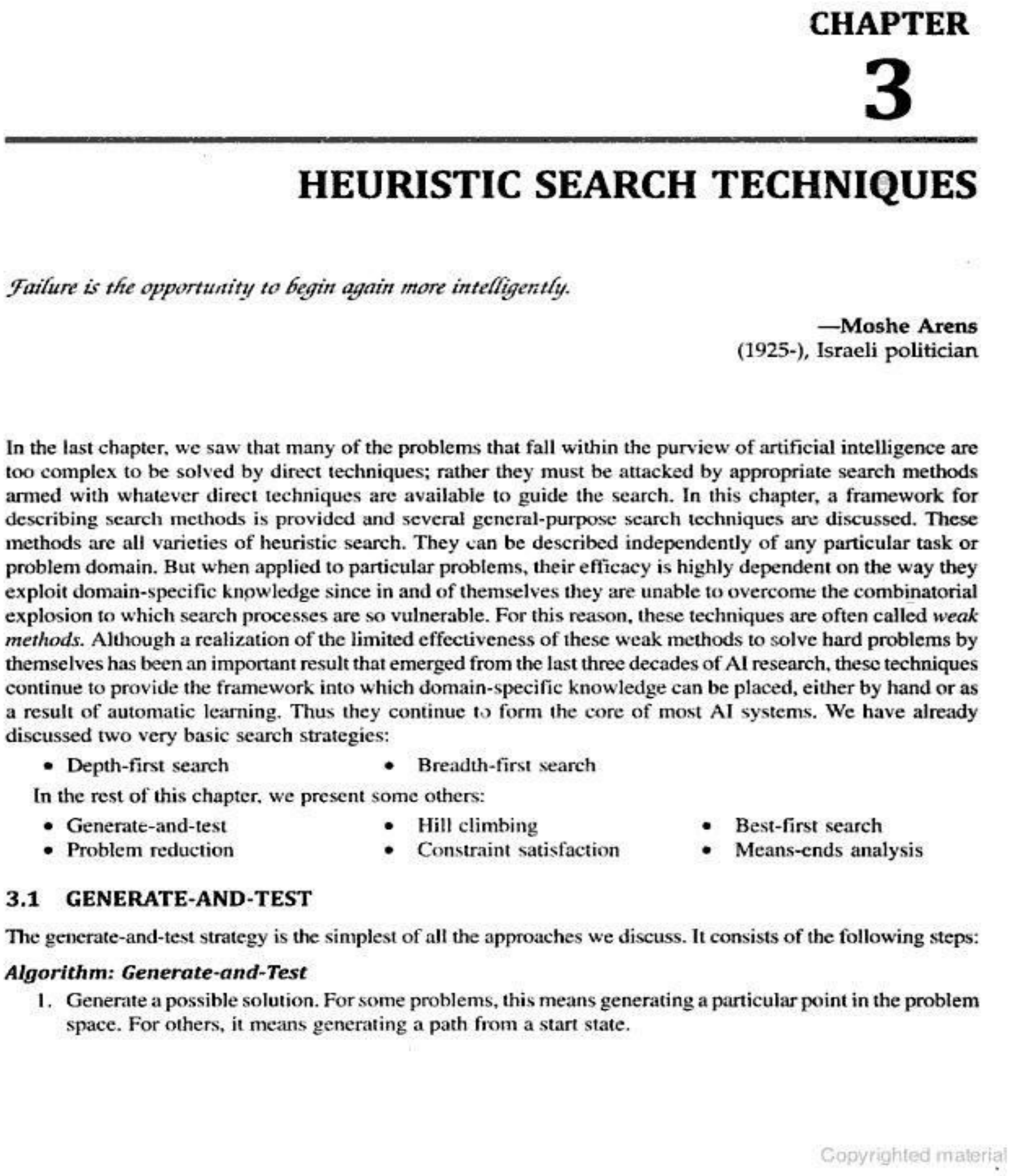
- In order to provide formal description of a problem, we must do the following:

1. Define a state space that contains all the possible configuration of the relevant objects. It is of course, possible to define this space without explicitly enumerating all the state it contains.
2. Specify one or more states within that space that describe possible situation from which the problem solving process may start. These states are called the *initial states*.
3. Specify a set of rules that would be acceptable as solutions to the problem. These states are called as *goal states*.
4. Specify a set of rules that describe the actions (operators) available. Doing this will require giving thought to the following issues:

- What unstated assumptions are present in the informal problem description?
- How general should the rules be?
- How much of the work required to solve the problem should be precomputed and represented in the rules?

- The problem can be solved by using rules, in combination with an appropriate control strategy to move through the problem space until a path from initial state to a goal state is found.
- Thus the process of search is fundamental to the problem-solving process.
- This does not mean that, more direct methods cannot be exploited. Whenever possible, they can be included as steps in the search by encoding them into the rules.

Heuristics Search Techniques CO1



CHAPTER

3

HEURISTIC SEARCH TECHNIQUES

Failure is the opportunity to begin again more intelligently.

—**Moshe Arens**
(1925-), Israeli politician

In the last chapter, we saw that many of the problems that fall within the purview of artificial intelligence are too complex to be solved by direct techniques; rather they must be attacked by appropriate search methods armed with whatever direct techniques are available to guide the search. In this chapter, a framework for describing search methods is provided and several general-purpose search techniques are discussed. These methods are all varieties of heuristic search. They can be described independently of any particular task or problem domain. But when applied to particular problems, their efficacy is highly dependent on the way they exploit domain-specific knowledge since in and of themselves they are unable to overcome the combinatorial explosion to which search processes are so vulnerable. For this reason, these techniques are often called *weak methods.* Although a realization of the limited effectiveness of these weak methods to solve hard problems by themselves has been an important result that emerged from the last three decades of AI research, these techniques continue to provide the framework into which domain-specific knowledge can be placed, either by hand or as a result of automatic learning. Thus they continue to form the core of most AI systems. We have already discussed two very basic search strategies:

- Depth-first search
- Breadth-first search

In the rest of this chapter, we present some others:

- Generate-and-test
- Problem reduction
- Hill climbing
- Constraint satisfaction
- Best-first search
- Means-ends analysis

3.1 GENERATE-AND-TEST

The generate-and-test strategy is the simplest of all the approaches we discuss. It consists of the following steps:

Algorithm: Generate-and-Test

1. Generate a possible solution. For some problems, this means generating a particular point in the problem space. For others, it means generating a path from a start state.

2. Test to see if this is actually a solution by comparing the chosen point or the endpoint of the chosen path to the set of acceptable goal states.
3. If a solution has been found, quit. Otherwise, return to step 1.

If the generation of possible solutions is done systematically, then this procedure will find a solution eventually, if one exists. Unfortunately, if the problem space is very large, "eventually" may be a very long time.

The generate-and-test algorithm is a depth-first search procedure since complete solutions must be generated before they can be tested. In its most systematic form, it is simply an exhaustive search of the problem space. Generate-and-test can, of course, also operate by generating solutions randomly, but then there is no guarantee that a solution will ever be found. In this form, it is also known as the British Museum algorithm, a reference to a method for finding an object in the British Museum by wandering randomly.[1] Between these two extremes lies a practical middle ground in which the search process proceeds systematically, but some paths are not considered because they seem unlikely to lead to a solution. This evaluation is performed by a heuristic function, as described in Section 2.2.2.

The most straightforward way to implement systematic generate-and-test is as a depth-first search tree with backtracking. If some intermediate states are likely to appear often in the tree, however, it may be better to modify that procedure, as described above, to traverse a graph rather than a tree.

For simple problems, exhaustive generate-and-test is often a reasonable technique. For example, consider the puzzle that consists of four six-sided cubes, with each side of each cube painted one of four colors. A solution to the puzzle consists of an arrangement of the cubes in a row such that on all four sides of the row one block face of each color is showing. This problem can be solved by a person (who is a much slower processor for this sort of thing than even a very cheap computer) in several minutes by systematically and exhaustively trying all possibilities. It can be solved even more quickly using a heuristic generate-and-test procedure. A quick glance at the four blocks reveals that there are more, say, red faces than there are of other colors. Thus when placing a block with several red faces, it would be a good idea to use as few of them as possible as outside faces. As many of them as possible should be placed to abut the next block. Using this heuristic, many configurations need never be explored and a solution can be found quite quickly.

Unfortunately, for problems much harder than this, even heuristic generate-and-test, all by itself, is not a very effective technique. But when combined with other techniques to restrict the space in which to search even further, the technique can be very effective.

For example, one early example of a successful AI program is DENDRAL [Lindsay *et al.*, 1980], which infers the structure of organic compounds using mass spectrogram and nuclear magnetic resonance (NMR) data. It uses a strategy called *plan-generate-test* in which a planning process that uses constraint-satisfaction techniques (see Section 3.5) creates lists of recommended and contraindicated substructures. The generate-and-test procedure then uses those lists so that it can explore only a fairly limited set of structures. Constrained in this way, the generate-and-test procedure has proved highly effective.

This combination of planning, using one problem-solving method (in this case, constraint satisfaction) with the use of the plan by another problem-solving method, generate-and-test, is an excellent example of the way techniques can be combined to overcome the limitations that each possesses individually. A major weakness of planning is that it often produces somewhat inaccurate solutions since there is no feedback from the world. But by using it only to produce pieces of solutions that will then be exploited in the generate-and-test process, the lack of detailed accuracy becomes unimportant. And, at the same time, the combinatorial problems that arise in simple generate-and-test are avoided by judicious reference to the plans.

[1] Or, as another story goes, if a sufficient number of monkeys were placed in front of a set of typewriters and left alone long enough, then they would eventually produce all of the works of Shakespeare.

3.2 HILL CLIMBING

Hill climbing is a variant of generate-and-test in which feedback from the test procedure is used to help the generator decide which direction to move in the search space. In a pure generate-and-test procedure, the test function responds with only a yes or no. But if the test function is augmented with a heuristic function[2] that provides an estimate of how close a given state is to a goal state, the generate procedure can exploit it as shown in the procedure below. This is particularly nice because often the computation of the heuristic function can be done at almost no cost at the same time that the test for a solution is being performed. Hill climbing is often used when a good heuristic function is available for evaluating states but when no other useful knowledge is available. For example, suppose you are in an unfamiliar city without a map and you want to get downtown. You simply aim for the tall buildings. The heuristic function is just distance between the current location and the location of the tall buildings and the desirable states are those in which this distance is minimized.

Recall from Section 2.3.4 that one way to characterize problems is according to their answer to the question, "Is a good solution absolute or relative?" Absolute solutions exist whenever it is possible to recognize a goal state just by examining it. Getting downtown is an example of such a problem. For these problems, hill climbing can terminate whenever a goal state is reached. Only relative solutions exist, however, for maximization (or minimization) problems, such as the traveling salesman problem. In these problems, there is no *a priori* goal state. For problems of this sort, it makes sense to terminate hill climibing when there is no reasonable alternative state to move to.

3.2.1 Simple Hill Climbing

The simplest way to implement hill climbing is as follows.

Algorithm: Simple Hill Climbing

```
1. Evaluate the initial state. If it is also a goal state, then return it and quit. Otherwise, continue with the
   initial state as the current state.
2. Loop until a solution is found or until there are no new operators left to be applied in the current state:
   (a) Select an operator that has not yet been applied to the current state and apply it to produce a new state.
   (b) Evaluate the new state.
       (i) If it is a goal state, then return it and quit.
       (ii) If it is not a goal state but it is better than the current state, then make it the current state.
       (iii) If it is not better than the current state, then continue in the loop.
```

The key difference between this algorithm and the one we gave for generate-and-test is the use of an evaluation function as a way to inject task-specific knowledge into the control process. It is the use of such knowledge that makes this and the other methods discussed in the rest of this chapter *heuristic* search methods, and it is that same knowledge that gives these methods their power to solve some otherwise intractable problems.

Notice that in this algorithm, we have asked the relatively vague question, "Is one state *better* than another?" For the algorithm to work, a precise definition of *better* must be provided. In some cases, it means a higher value of the heuristic function. In others, it means a lower value. It does not matter which, as long as a particular hill-climbing program is consistent in its interpretation.

To see how hill climbing works, let's return to the puzzle of the four colored blocks. To solve the problem, we first need to define a heuristic function that describes how close a particular configuration is to being a solution. One such function is simply the sum of the number of different colors on each of the four sides. A solution to the puzzle will have a value of 16. Next we need to define a set of rules that describe ways of transforming one configuration into another. Actually, one rule will suffice. It says simply pick a block and

[2] What we are calling the heuristic function is sometimes also called the *objective function*, particularly in the literature of mathematical optimization.

rotate it 90 degrees in any direction. Having provided these definitions, the next step is to generate a starting configuration. This can either be done at random or with the aid of the heuristic function described in the last section. Now hill climbing can begin. We generate a new state by selecting a block and rotating it. If the resulting state is better, then we keep it. If not, we return to the previous state and try a different perturbation.

3.2.2 Steepest-Ascent Hill Climbing

A useful variation on simple hill climbing considers all the moves from the current state and selects the best one as the next state. This method is called *steepest-ascent hill climbing* or *gradient search.* Notice that this contrasts with the basic method in which the first state that is better than the current state is selected. The algorithm works as follows.

Algorithm: Steepest-Ascent Hill Climbing

```
1. Evaluate the initial state. If it is also a goal state, then return it and quit. Otherwise, continue with the
   initial state as the current state.
2. Loop until a solution is found or until a complete iteration produces no change to current state:
   (a) Let SUCC be a state such that any possible successor of the current state will be better than SUCC.
   (b) For each operator that applies to the current state do:
        (i) Apply the operator and generate a new state.
        (ii) Evaluate the new state. If it is a goal state, then return it and quit. If not, compare it to SUCC.
             If it is better, then set SUCC to this state. If it is not better, leave SUCC alone.
   (c) If the SUCC is better than current state, then set current state to SUCC.
```

To apply steepest-ascent hill climbing to the colored blocks problem, we must consider all perturbations of the initial state and choose the best. For this problem, this is difficult since there are so many possible moves. There is a trade-off between the time required to select a move (usually longer for steepest-ascent hill climbing) and the number of moves required to get to a solution (usually longer for basic hill climbing) that must be considered when deciding which method will work better for a particular problem.

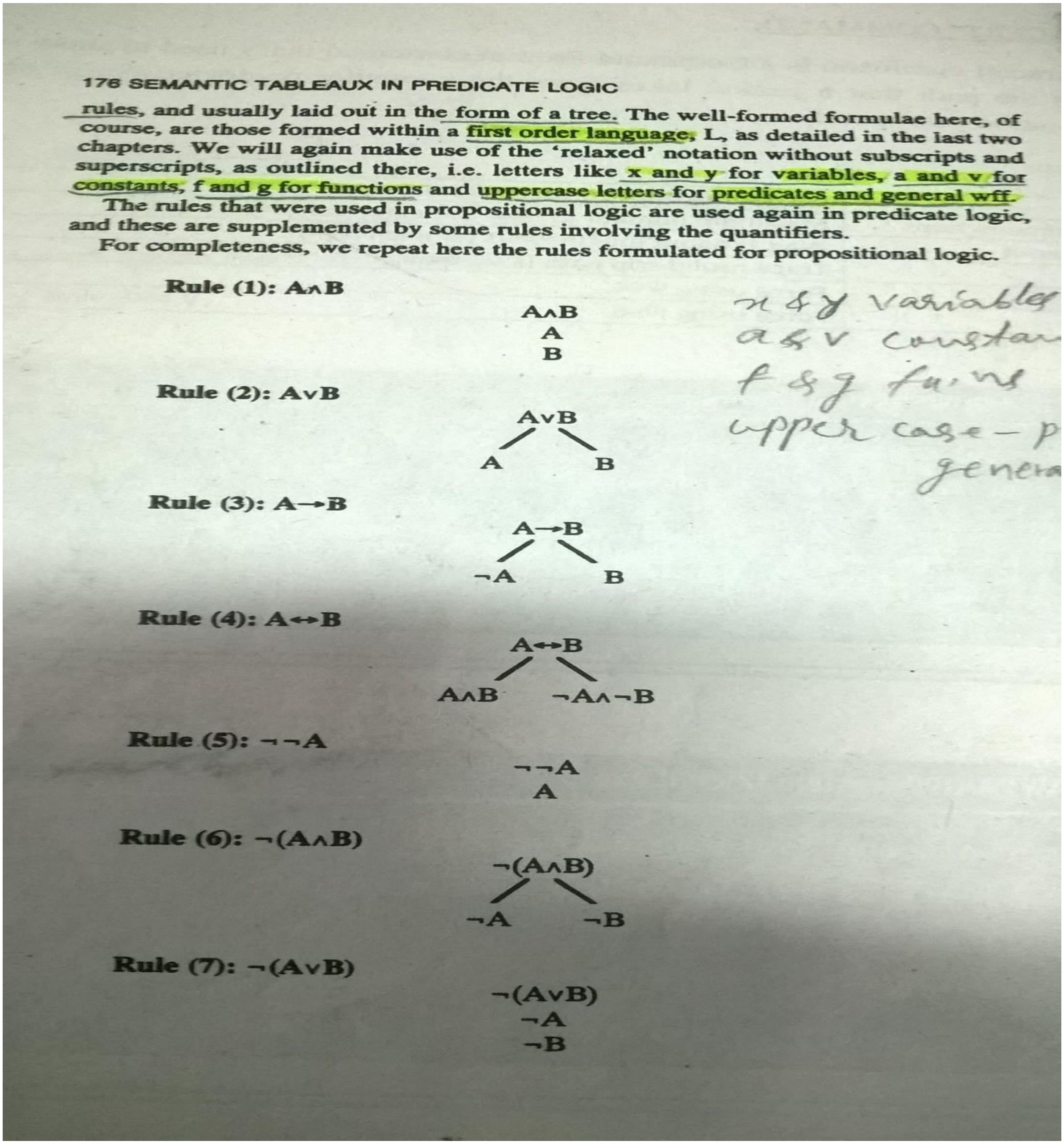
Both basic and steepest-ascent hill climbing may fail to find a solution. Either algorithm may terminate not by finding a goal state but by getting to a state from which no better states can be generated. This will happen if the program has reached either a local maximum, a plateau, or a ridge.

> A *local maximum* is a state that is better than all its neighbors but is not better than some other states farther away. At a local maximum, all moves appear to make things worse. Local maxima are particularly frustrating because they often occur almost within sight of a solution. In this case, they are called *foothills.*
>
> *A plateau* is a flat area of the search space in which a whole set of neighboring states have the same value. On a plateau, it is not possible to determine the best direction in which to move by making local comparisons.
>
> A *ridge* is a special kind of local maximum. It is an area of the search space that is higher than surrounding areas and that itself has a slope (which one would like to climb). But the orientation of the high region, compared to the set of available moves and the directions in which they move, makes it impossible to traverse a ridge by single moves.

There are some ways of dealing with these problems, although these methods are by no means guaranteed:

- Backtrack to some earlier node and try going in a different direction. This is particularly reasonable if at that node there was another direction that looked as promising or almost as promising as the one that was chosen earlier. To implement this strategy, maintain a list of paths almost taken and go back to one of them if the path that was taken leads to a dead end. This is a fairly good way of dealing with local maxima.
- Make a big jump in some direction to try to get to a new section of the search space. This is a particularly good way of dealing with plateaus. If the only rules available describe single small steps, apply them several times in the same direction.
- Apply two or more rules before doing the test. This corresponds to moving in several directions at once. This is a particularly good strategy for dealing with ridges.

Predicate Logic CO1



rules, and usually laid out in the form of a tree. The well-formed formulae here, of course, are those formed within a first order language, L, as detailed in the last two chapters. We will again make use of the 'relaxed' notation without subscripts and superscripts, as outlined there, i.e. letters like x and y for variables, a and v for constants, f and g for functions and uppercase letters for predicates and general wff.

The rules that were used in propositional logic are used again in predicate logic, and these are supplemented by some rules involving the quantifiers.

For completeness, we repeat here the rules formulated for propositional logic.

Rule (1): $A \wedge B$

$A \wedge B$
A
B

Rule (2): $A \vee B$

$A \vee B$
/ \
A B

Rule (3): $A \rightarrow B$

$A \rightarrow B$
/ \
$\neg A$ B

Rule (4): $A \leftrightarrow B$

$A \leftrightarrow B$
/ \
$A \wedge B$ $\neg A \wedge \neg B$

Rule (5): $\neg\neg A$

$\neg\neg A$
A

Rule (6): $\neg(A \wedge B)$

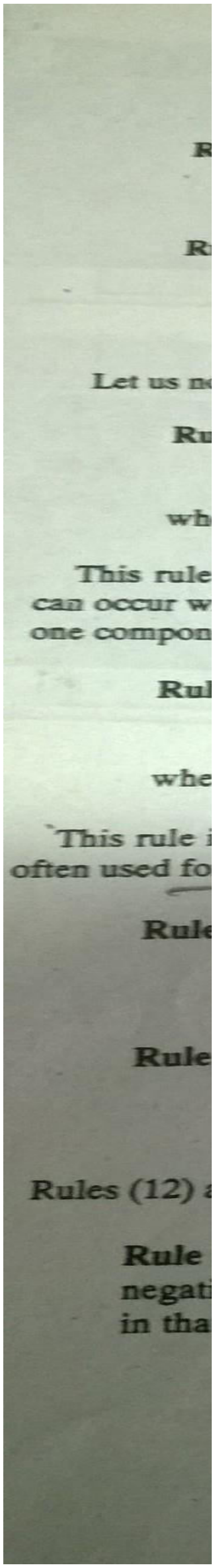
$\neg(A \wedge B)$
/ \
$\neg A$ $\neg B$

Rule (7): $\neg(A \vee B)$

$\neg(A \vee B)$
$\neg A$
$\neg B$

x & y variables
a & v constan
f & g fu'ns
upper case - p
genera

Knowledge Representation CO1



R

R

Let us n

Ru

wh

This rule
can occur w
one compon

Rul

whe

`This rule i
often used fo

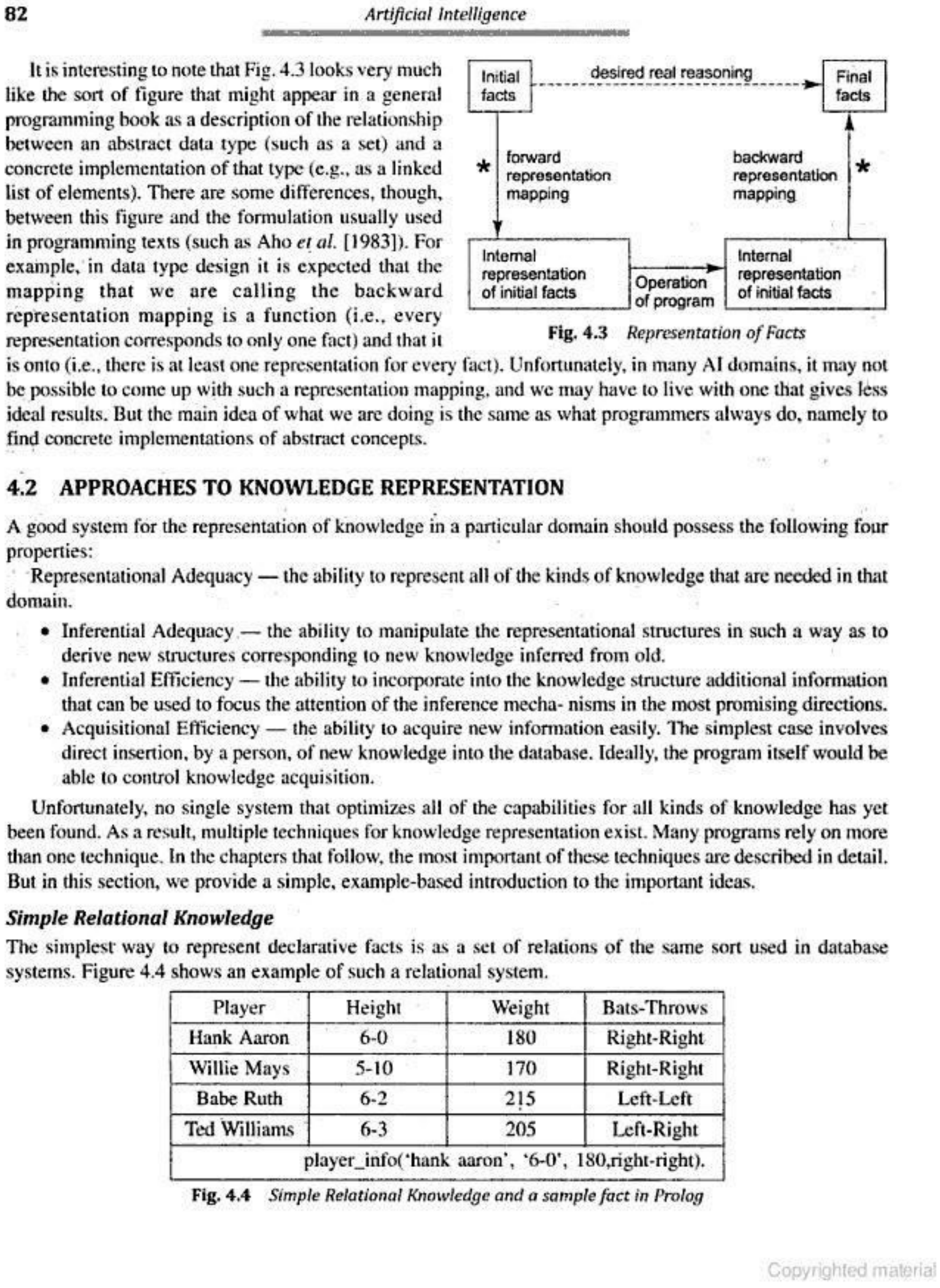
Rule

Rule

Rules (12) a

Rule
negati
in tha

Knowledge Representation CO1



It is interesting to note that Fig. 4.3 looks very much like the sort of figure that might appear in a general programming book as a description of the relationship between an abstract data type (such as a set) and a concrete implementation of that type (e.g., as a linked list of elements). There are some differences, though, between this figure and the formulation usually used in programming texts (such as Aho *et al.* [1983]). For example, in data type design it is expected that the mapping that we are calling the backward representation mapping is a function (i.e., every representation corresponds to only one fact) and that it is onto (i.e., there is at least one representation for every fact). Unfortunately, in many AI domains, it may not be possible to come up with such a representation mapping, and we may have to live with one that gives less ideal results. But the main idea of what we are doing is the same as what programmers always do, namely to find concrete implementations of abstract concepts.

Initial facts
desired real reasoning
Final facts
*
forward representation mapping
backward representation mapping
*
Internal representation of initial facts
Operation of program
Internal representation of initial facts

Fig. 4.3 *Representation of Facts*

4.2 APPROACHES TO KNOWLEDGE REPRESENTATION

A good system for the representation of knowledge in a particular domain should possess the following four properties:

Representational Adequacy — the ability to represent all of the kinds of knowledge that are needed in that domain.

- Inferential Adequacy — the ability to manipulate the representational structures in such a way as to derive new structures corresponding to new knowledge inferred from old.
- Inferential Efficiency — the ability to incorporate into the knowledge structure additional information that can be used to focus the attention of the inference mecha- nisms in the most promising directions.
- Acquisitional Efficiency — the ability to acquire new information easily. The simplest case involves direct insertion, by a person, of new knowledge into the database. Ideally, the program itself would be able to control knowledge acquisition.

Unfortunately, no single system that optimizes all of the capabilities for all kinds of knowledge has yet been found. As a result, multiple techniques for knowledge representation exist. Many programs rely on more than one technique. In the chapters that follow, the most important of these techniques are described in detail. But in this section, we provide a simple, example-based introduction to the important ideas.

Simple Relational Knowledge

The simplest way to represent declarative facts is as a set of relations of the same sort used in database systems. Figure 4.4 shows an example of such a relational system.

Player	Height	Weight	Bats-Throws
Hank Aaron	6-0	180	Right-Right
Willie Mays	5-10	170	Right-Right
Babe Ruth	6-2	215	Left-Left
Ted Williams	6-3	205	Left-Right
player_info('hank aaron', '6-0', 180,right-right).			

Fig. 4.4 *Simple Relational Knowledge and a sample fact in Prolog*

The reason that this representation is simple is that standing alone it provides very weak inferential capabilities But knowledge represented in this form may serve as the input to more powerful inference engines. For example, given just the facts of Fig. 4.4, it is not possible even to answer the simple question, "Who is the heaviest player?" But if a procedure for finding the heaviest player is provided, then these facts will enable the procedure to compute an answer. If, instead, we are provided with a set of rules for deciding which hitter to put up against a given pitcher (based on right- and left-handedness, say), then this same relation can provide at least some of the information required by those rules.

Providing support for relational knowledge is what database systems are designed to do. Thus we do not need to discuss this kind of knowledge representation structure further here. The practical issues that arise in linking a database system that provides this kind of support to a knowledge representation system that provides some of the other capabilities that we are about to discuss have already been solved in several commercial products.

Inheritable Knowledge

The relational knowledge of Fig. 4.4 corresponds to a set of attributes and associated values that together describe the objects of the knowledge base. Knowledge about objects, their attributes, and their values need not be as simple as that shown in our example. In particular, it is possible to augment the basic representation with inference mechanisms that operate on the structure of the representation. For this to be effective, the structure must be designed to correspond to the inference mechanisms that are desired. One of the most useful forms of inference is *property inheritance*, in which elements of specific classes inherit attributes and values from more general classes in which they are included.

In order to support property inheritance, objects must be organized into classes and classes must be arranged in a generalization hierarchy. Figure 4.5 shows some additional baseball knowledge inserted into a structure that is so arranged. Lines represent attributes. Boxed nodes represent objects and values of attributes of objects. These values can also be viewed as objects with attributes and values, and so on. The arrows on the lines point from an object to its value along the corresponding attribute line, The structure shown in the figure is a *slot-and-filler structure*. It may also be called a *semantic network* or a collection *of frames*. In the latter case, each individual frame represents the collection of attributes and values associated with a particular node. Figure 4.6 shows the node for baseball player displayed as a frame.

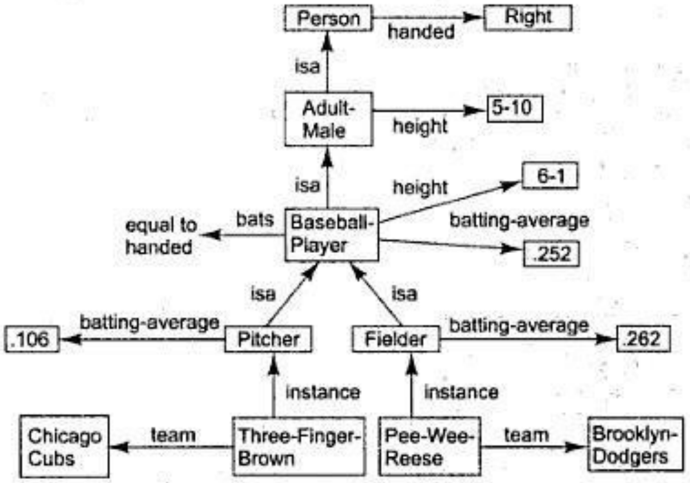


Figure 4.5 *Inheritable Knowledge*

Baseball-Player

isa:	*Adult-Male*
bats:	(EQUAL *handed)*
height:	6-1
batting-average:	.252

Fig. 4.6 *Viewing a Node as a Frame*

Do not be put off by the confusion in terminology here. There is so much flexibility in the way that this (and the other structures described in this section) can be used to solve particular representation problems that it is difficult to reserve precise words for particular representations. Usually the use of the term *frame system* implies somewhat more structure on the attributes and the inference mechanisms that are available to apply to them than does the term *semantic network.*

In Chapter 9 we discuss structures such as these in substantial detail. But to get an idea of how these structures support inference using the knowledge they contain, we discuss them briefly here. All of the objects and most of the attributes shown in this example have been chosen to correspond to the baseball domain, and they have no general significance. The two exceptions to this are the attribute *isa,* which is being used to show class inclusion, and the attribute *instance,* which is being used to show class membership. These two specific (and generally useful) attributes provide the basis for property inheritance as an inference technique. Using this technique, the knowledge base can support retrieval both of facts that have been explicitly stored and of facts that can be derived from those that are explicitly stored.

An idealized form of the property inheritance algorithm can be stated as follows:

Algorithm: Property Inheritance

To retrieve a value *V* for attribute A of an instance object *O:*

```
1. Find O in the knowledge base.
2. If there is a value there for the attribute A, report that value.
3. Otherwise, see if there is a value for the attribute instance. If not, then fail.
4. Otherwise, move to the node corresponding to that value and look for a value for the attribute A. If one
   is found, report it.
5. Otherwise, do until there is no value for the isa attribute or until an answer is found:
   (a) Get the value of the isa attribute and move to that node.
   (b) See if there is a value for the attribute A. If there is, report it.
```

This procedure is simplistic. It does not say what we should do if there is more than one value of the *instance* or *isa* attribute. But it does describe the basic mechanism of inheritance. We can apply this procedure to our example knowledge base to derive answers to the following queries:

- *team(Pee-Wee-Reese) = Brooklyn-Dodgers.* This attribute had a value stored explicitly in the knowledge base.
- *batting-average(Three-Finger Brown)* = .106. Since there is no value for batting average stored explicitly for Three Finger Brown, we follow the *instance* attribute to *Pitcher* and extract the value stored there. Now we observe one of the critical characteristics of property inheritance, namely that it may produce default values that are not guaranteed to be correct but that represent "best guesses" in the face of a lack of more precise information. In fact, in 1906, Brown's batting average was .204.
- *height(Pee-Wee-Reese)* = 6-1. This represents another default inference. Notice here that because we get to it first, the more specific fact about the height of baseball players overrides a more general fact about the height of adult males.

CVR COLLEGE OF ENGINEERING

An UGC Autonomous Institution - Affiliated to JNTUH

Handout – 2

Unit - 2

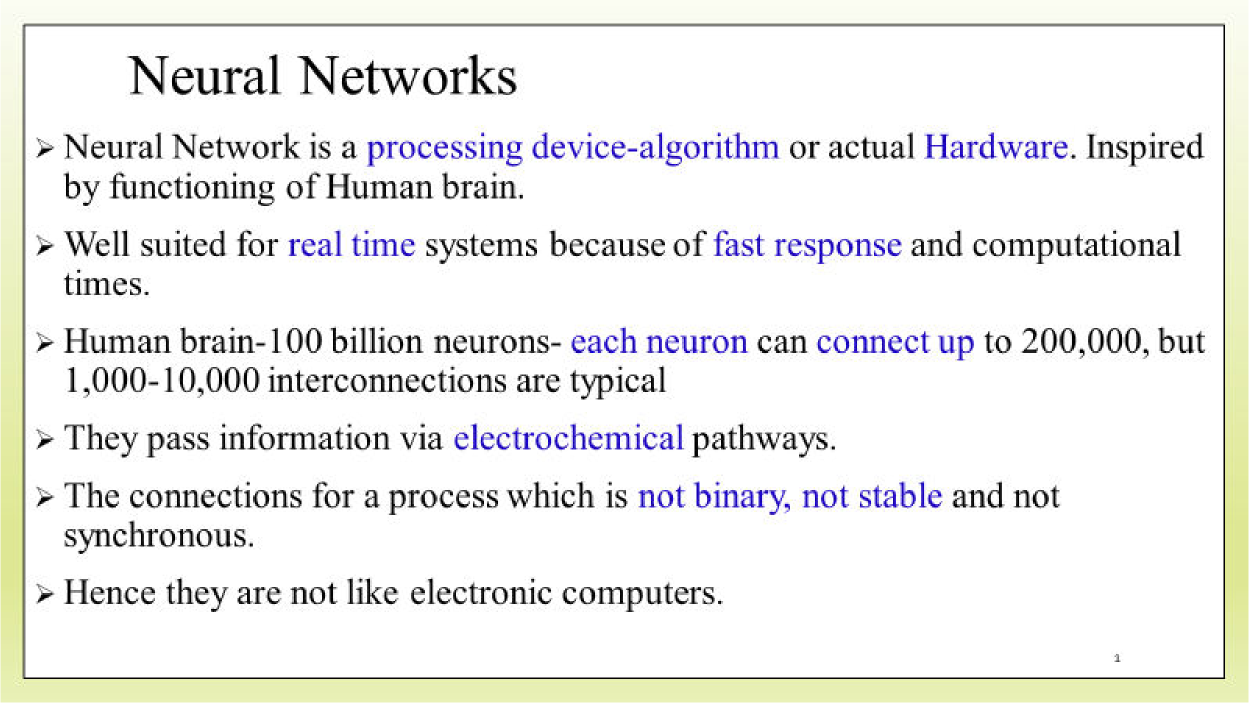
Year and Semester: IVyr &II Sem

Subject**: Soft Computing**

Branch: **CSE**

Faculty: **Dr.Md.Yusuf Mulge**, Professor (CSE)

Neural Networks CO2



Neural Networks

- Neural Network is a processing device-algorithm or actual Hardware. Inspired by functioning of Human brain.
- Well suited for real time systems because of fast response and computational times.
- Human brain-100 billion neurons- each neuron can connect up to 200,000, but 1,000-10,000 interconnections are typical
- They pass information via electrochemical pathways.
- The connections for a process which is not binary, not stable and not synchronous.
- Hence they are not like electronic computers.

1

Artificial Neural Networks

- An ANN is an information-processing model that is inspired by functioning of biological nervous system such as brain.
- Large number of highly interconnected processing elements that Learn by example.
- Configured for a specific application such as pattern recognition or data classification.
- In biological systems learning involves adjustments to synaptic connections that exist b/w neurons. So also in ANNs
- Many neural networks are statistically quite accurate.
- 85-90% accurate, unfortunately very few systems tolerate this level of accuracy.

Advantages of Neural Networks

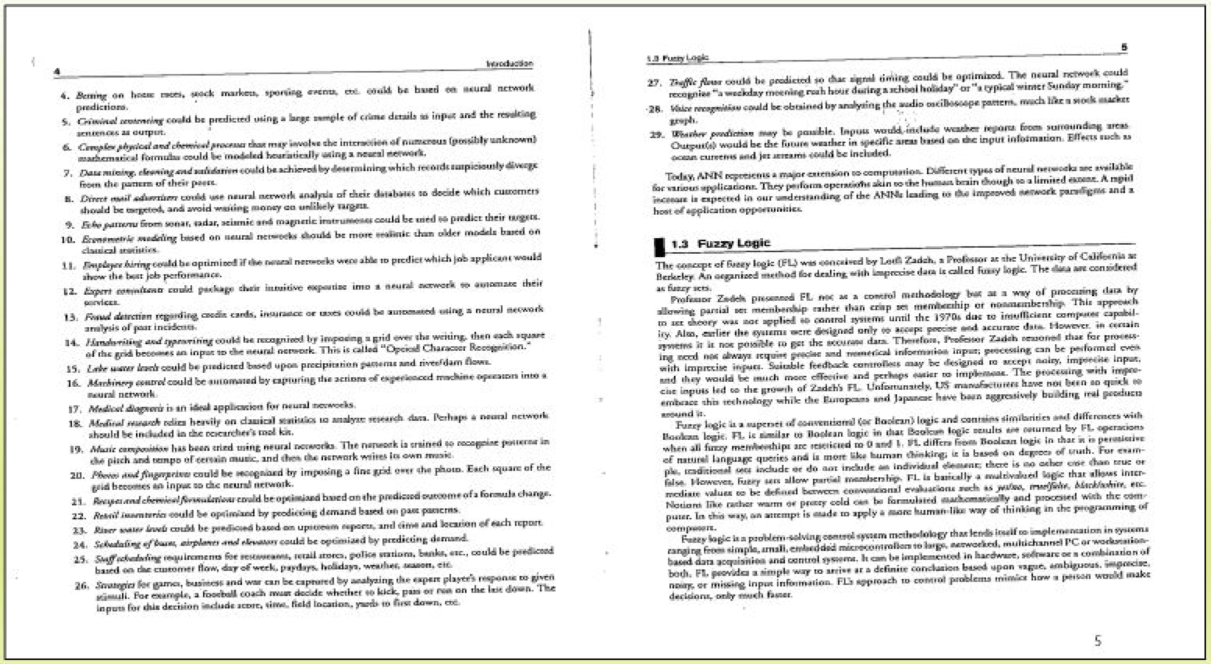
- NNs with their ability to derive meaning from complicated or imprecise data can be used to extract patterns and detect trends that are too complex for humans or other computer techniques.
- A trained ANN is an expert in a particular category of information.
- Other advantages of ANNs are:

- Adaptive learning: based on given data for training or initial experience.
- Self-organization
- Real-time operation
- Fault tolerance via redundant information coding: partial destruction of a NN leads to degradation of performance, but some n/w capabilities are retained

Applications of Neural Networks

- ➢NNs can be used by financial institutions to approve loans. The decision taken by NNs is more accurate than the human decision.
- ➢Some financial institutions are using NNs for screening the credit card applications.
- ➢Other application areas are:

1. Air traffic control: could be automated based on location, altitude, direction and speed of each radar blip taken as i/p to the n/w.
2. Animal behavior, predator/prey relationship and population cycles may be suitable for analysis by neural networks.
3. Appraisal and valuation of property, buildings, automobiles, machinery etc..



4. *Betting* on horse races, stock markets, sporting events, etc. could be based on neural network predictions.
5. *Criminal sentencing* could be predicted using a large sample of crime details as input and the resulting sentences as output.
6. *Complex physical and chemical processes* that may involve the interaction of numerous (possibly unknown) mathematical formulas could be modeled heuristically using a neural network.
7. *Data mining, cleaning and validation* could be achieved by determining which records suspiciously diverge from the pattern of their peers.
8. *Direct mail advertisers* could use neural network analysis of their databases to decide which customers should be targeted, and avoid wasting money on unlikely targets.
9. *Echo patterns* from sonar, radar, seismic and magnetic instruments could be used to predict their targets.
10. *Econometric modeling* based on neural networks should be more realistic than older models based on classical statistics.
11. *Employee hiring* could be optimized if the neural networks were able to predict which job applicant would show the best job performance.
12. *Expert consultants* could package their intuitive expertise into a neural network to automate their services.
13. *Fraud detection* regarding credit cards, insurance or taxes could be automated using a neural network analysis of past incidents.
14. *Handwriting and typewriting* could be recognized by imposing a grid over the writing, then each square of the grid becomes an input to the neural network. This is called "Optical Character Recognition."
15. *Lake water levels* could be predicted based upon precipitation patterns and river/dam flows.
16. *Machinery control* could be automated by capturing the actions of experienced machine operators into a neural network.
17. *Medical diagnosis* is an ideal application for neural networks.
18. *Medical research* relies heavily on classical statistics to analyze research data. Perhaps a neural network should be included in the researcher's tool kit.
19. *Music composition* has been tried using neural networks. The network is trained to recognize patterns in the pitch and tempo of certain music, and then the network writes its own music.
20. *Photos and fingerprints* could be recognized by imposing a fine grid over the photo. Each square of the grid becomes an input to the neural network.
21. *Recipes and chemical formulations* could be optimized based on the predicted outcome of a formula change.
22. *Retail inventories* could be optimized by predicting demand based on past patterns.
23. *River water levels* could be predicted based on upstream reports, and time and location of each report.
24. *Scheduling of buses, airplanes and elevators* could be optimized by predicting demand.
25. *Staff scheduling* requirements for restaurants, retail stores, police stations, banks, etc., could be predicted based on the customer flow, day of week, paydays, holidays, weather, season, etc.
26. *Strategies* for games, business and war can be captured by analyzing the expert player's response to given stimuli. For example, a football coach must decide whether to kick, pass or run on the last down. The inputs for this decision include score, time, field location, yards to first down, etc.

27. *Traffic flows* could be predicted so that signal timing could be optimized. The neural network could recognize "a weekday morning rush hour during a school holiday" or "a typical winter Sunday morning."
28. *Voice recognition* could be obtained by analyzing the audio oscilloscope pattern, much like a stock market graph.
29. *Weather prediction* may be possible. Inputs would include weather reports from surrounding areas. Output(s) would be the future weather in specific areas based on the input information. Effects such as ocean currents and jet streams could be included.

Today, ANN represents a major extension to computation. Different types of neural networks are available for various applications. They perform operations akin to the human brain though to a limited extent. A rapid increase is expected in our understanding of the ANNs leading to the improved network paradigms and a host of application opportunities.

1.3 Fuzzy Logic

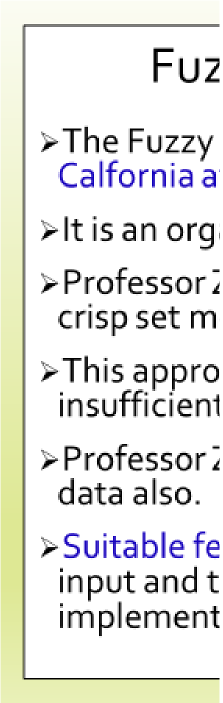
The concept of fuzzy logic (FL) was conceived by Lotfi Zadeh, a Professor at the University of California at Berkeley. An organized method for dealing with imprecise data is called fuzzy logic. The data are considered as fuzzy sets.

Professor Zadeh presented FL not as a control methodology but as a way of processing data by
allowing partial set membership rather than crisp set membership or nonmembership. This approach
to set theory was not applied to control systems until the 1970s due to insufficient computer capabil-
ity. Also, earlier the systems were designed only to accept precise and accurate data. However, in certain
systems it is not possible to get the accurate data. Therefore, Professor Zadeh reasoned that for process-
ing need not always require precise and numerical information input; processing can be performed even
with imprecise inputs. Suitable feedback controllers may be designed to accept noisy, imprecise input,
and they would be much more effective and perhaps easier to implement. The processing with impre-
cise inputs led to the growth of Zadeh's FL. Unfortunately, US manufacturers have not been so quick to
embrace this technology while the Europeans and Japanese have been aggressively building real products
around it.

Fuzzy logic is a superset of conventional (or Boolean) logic and contains similarities and differences with
Boolean logic. FL is similar to Boolean logic in that Boolean logic results are returned by FL operations
when all fuzzy memberships are restricted to 0 and 1. FL differs from Boolean logic in that it is permissive
of natural language queries and is more like human thinking; it is based on degrees of truth. For exam-
ple, traditional sets include or do not include an individual element; there is no other case than true or
false. However, fuzzy sets allow partial membership. FL is basically a multivalued logic that allows inter-
mediate values to be defined between conventional evaluations such as *yes/no, true/false, black/white,* etc.
Notions like rather warm or pretty cold can be formulated mathematically and processed with the com-
puter. In this way, an attempt is made to apply a more human-like way of thinking in the programming of
computers.

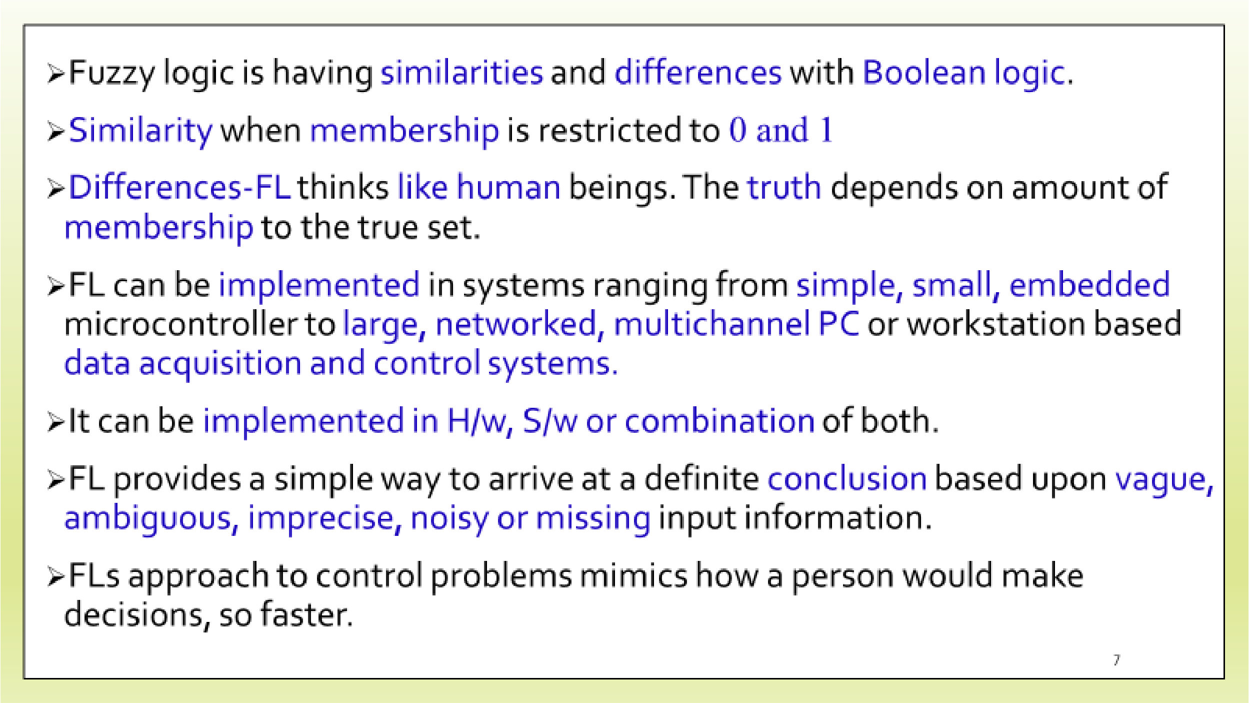
Fuzzy logic is a problem-solving control system methodology that lends itself to implementation in systems
ranging from simple, small, embedded microcontrollers to large, networked, multichannel PC or workstation-
based data acquisition and control systems. It can be implemented in hardware, software or a combination of
both. FL provides a simple way to arrive at a definite conclusion based upon vague, ambiguous, imprecise,
noisy, or missing input information. FL's approach to control problems mimics how a person would make
decisions, only much faster.

Fuzzy Logic CO2



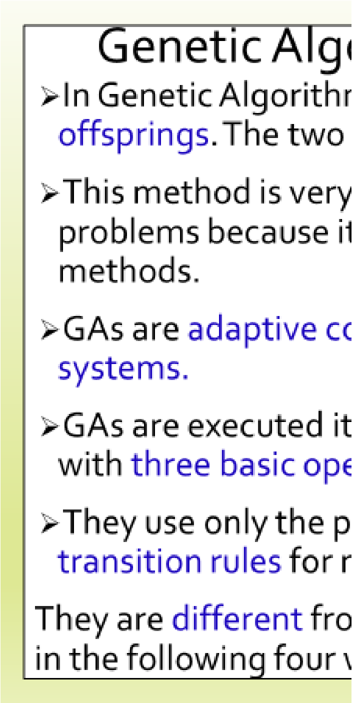
Fuz

- The Fuzzy Calfornia a
- It is an org
- Professor crisp set m
- This appro insufficient
- Professor data also.
- Suitable fe input and t implement



- Fuzzy logic is having similarities and differences with Boolean logic.
- Similarity when membership is restricted to 0 and 1
- Differences-FL thinks like human beings. The truth depends on amount of membership to the true set.
- FL can be implemented in systems ranging from simple, small, embedded microcontroller to large, networked, multichannel PC or workstation based data acquisition and control systems.
- It can be implemented in H/w, S/w or combination of both.
- FL provides a simple way to arrive at a definite conclusion based upon vague, ambiguous, imprecise, noisy or missing input information.
- FLs approach to control problems mimics how a person would make decisions, so faster.

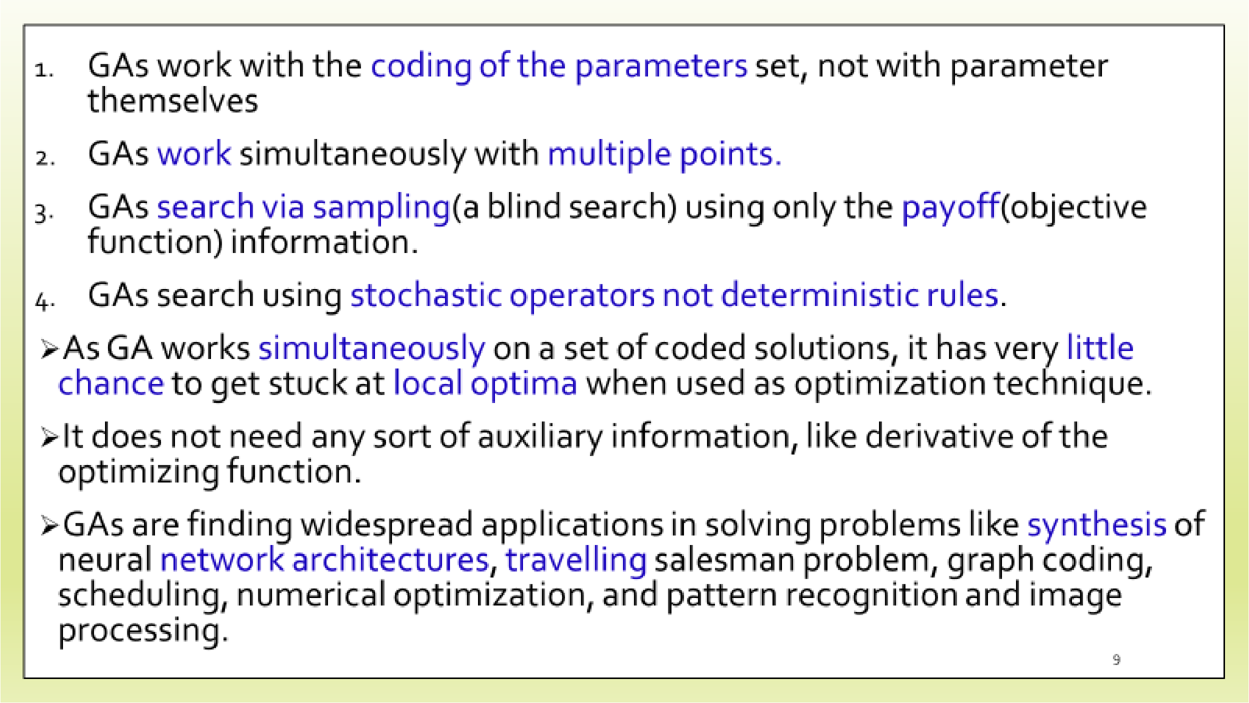
Genetic Algorithm CO2



Genetic Alg

- In Genetic Algorithr offsprings. The two
- This method is very problems because it methods.
- GAs are adaptive cc systems.
- GAs are executed it with three basic ope
- They use only the p transition rules for r

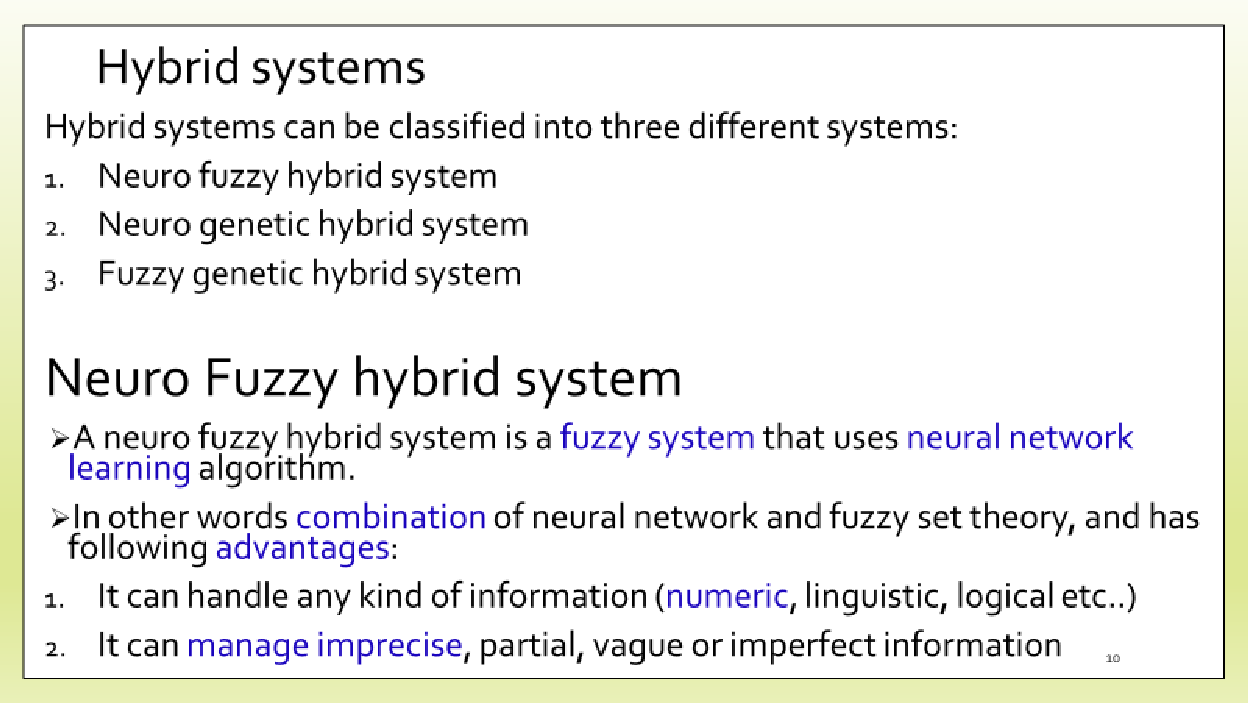
They are different fro in the following four v



1. GAs work with the coding of the parameters set, not with parameter themselves
2. GAs work simultaneously with multiple points.
3. GAs search via sampling(a blind search) using only the payoff(objective function) information.
4. GAs search using stochastic operators not deterministic rules.

- As GA works simultaneously on a set of coded solutions, it has very little chance to get stuck at local optima when used as optimization technique.
- It does not need any sort of auxiliary information, like derivative of the optimizing function.
- GAs are finding widespread applications in solving problems like synthesis of neural network architectures, travelling salesman problem, graph coding, scheduling, numerical optimization, and pattern recognition and image processing.

Hybrid Systems CO2



Hybrid systems

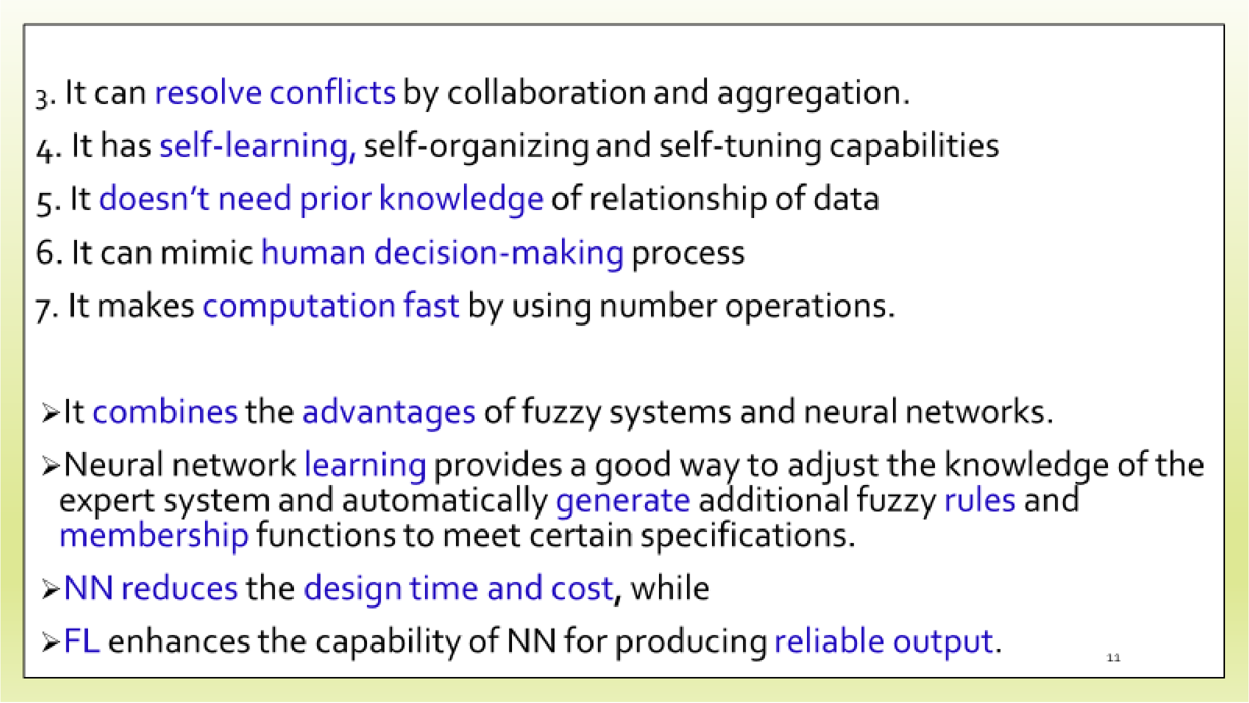
Hybrid systems can be classified into three different systems:

1. Neuro fuzzy hybrid system
2. Neuro genetic hybrid system
3. Fuzzy genetic hybrid system

Neuro Fuzzy hybrid system

- ➢A neuro fuzzy hybrid system is a fuzzy system that uses neural network learning algorithm.
- ➢In other words combination of neural network and fuzzy set theory, and has following advantages:

1. It can handle any kind of information (numeric, linguistic, logical etc..)
2. It can manage imprecise, partial, vague or imperfect information



3. It can resolve conflicts by collaboration and aggregation.
4. It has self-learning, self-organizing and self-tuning capabilities
5. It doesn’t need prior knowledge of relationship of data
6. It can mimic human decision-making process
7. It makes computation fast by using number operations.

- ➢It combines the advantages of fuzzy systems and neural networks.
- ➢Neural network learning provides a good way to adjust the knowledge of the expert system and automatically generate additional fuzzy rules and membership functions to meet certain specifications.
- ➢NN reduces the design time and cost, while
- ➢FL enhances the capability of NN for producing reliable output.

Neuro Genetic hybrid system

Genetic Algorithms have been applied in ANN design in several ways: like topology optimization, genetic training algorithms and control parameter optimization.

- ➢In topology optimization: GA is used to select a topology (number of hidden layers, number of hidden nodes, interconnections pattern) for ANN, which is trained using some training scheme, most commonly back propagation.
- ➢In genetic training algorithm: the learning of ANN is formulated as a weight optimization problem, usually using the inverse mean squared error as a fitness measure.
- ➢In control parameter optimization: Many of the control parameters such as learning rate, momentum rate, tolerance level etc..can also be optimized using GAs.

Fuzzy Genetic Hybrid systems

- ➢The optimization abilities of GA are used to develop the best set of rules to be used by a fuzzy inference engine, and to optimize the choice of membership functions.
- ➢A particular use of GA is in fuzzy classification, where an object is classified on the basis of the linguistic values of the object attributes.
- ➢The most difficult part is building the fuzzy rules for this classification. And this can be obtained by GA, which builds the set of rules based on the knowledge about the objects. Knowledge can be deduced from samples cases
- ➢The training data and randomly generated rules are combined to create initial population, giving better starting point for reproduction.

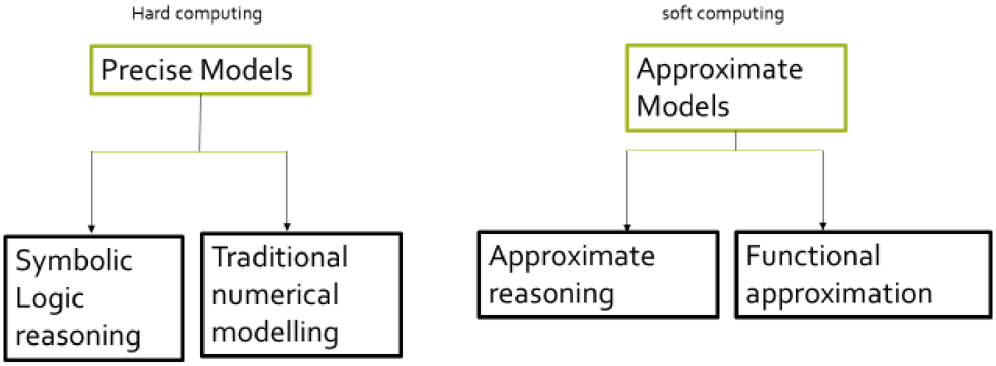
Soft computing

Two major problem-solving technologies are:

1. Hard computing
2. Soft computing

- Hard computing deals with precise models where accurate solutions are obtained quickly, while
- Soft computing deals with approximate models and gives solution to complex problems.
- Soft computing is relatively a new concept, introduced by Professor Lotfi Zadeh with the objective of exploiting the tolerance for imprecision, uncertainty and partial truth to achieve traceability, robustness, low solution cost and better rapport.
- The ultimate goal is to emulate human mind as closely as possible.
- Soft computing involves partnership of several fields, important being NNs, GA and FL

- Soft computing uses a combination of GAs, NNs and FL inheriting advantages of all these technologies, but won't have the features of single soft computing component.
- An important thing about soft computing is that, they are complementary and offers solutions to unsolvable problems.



Hard computing

Precise Models

Symbolic Logic reasoning

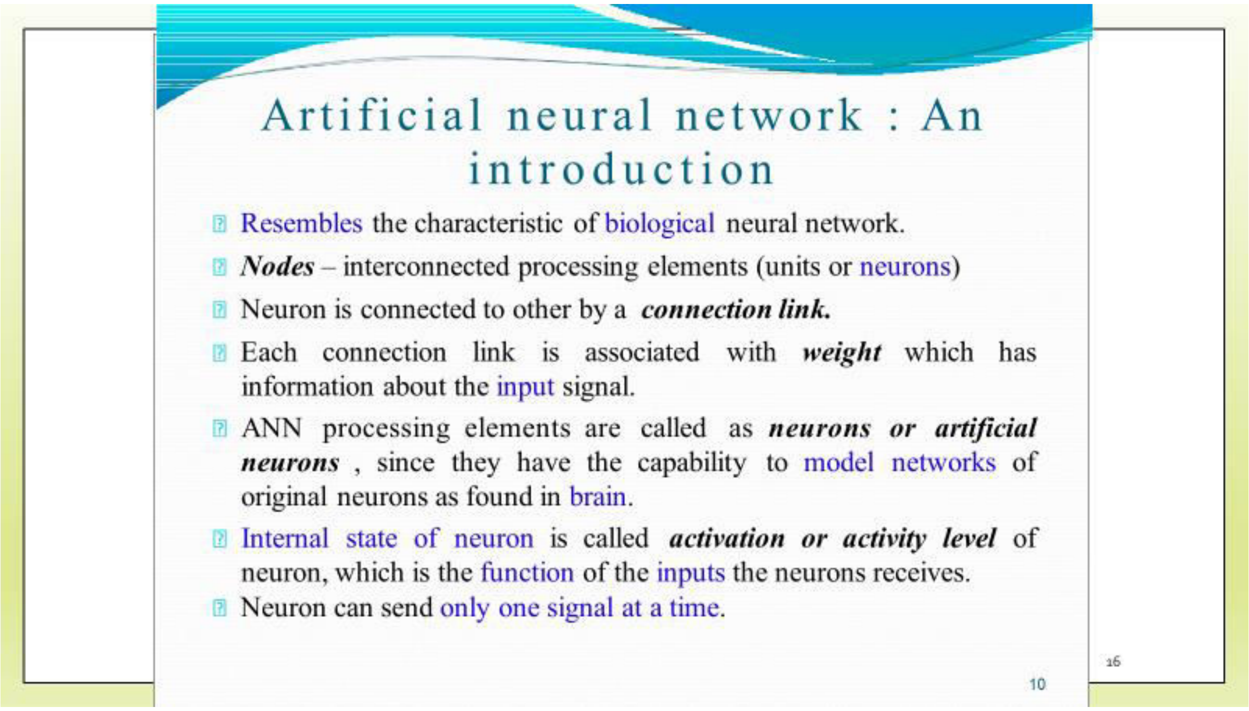
Traditional numerical modelling

soft computing

Approximate Models

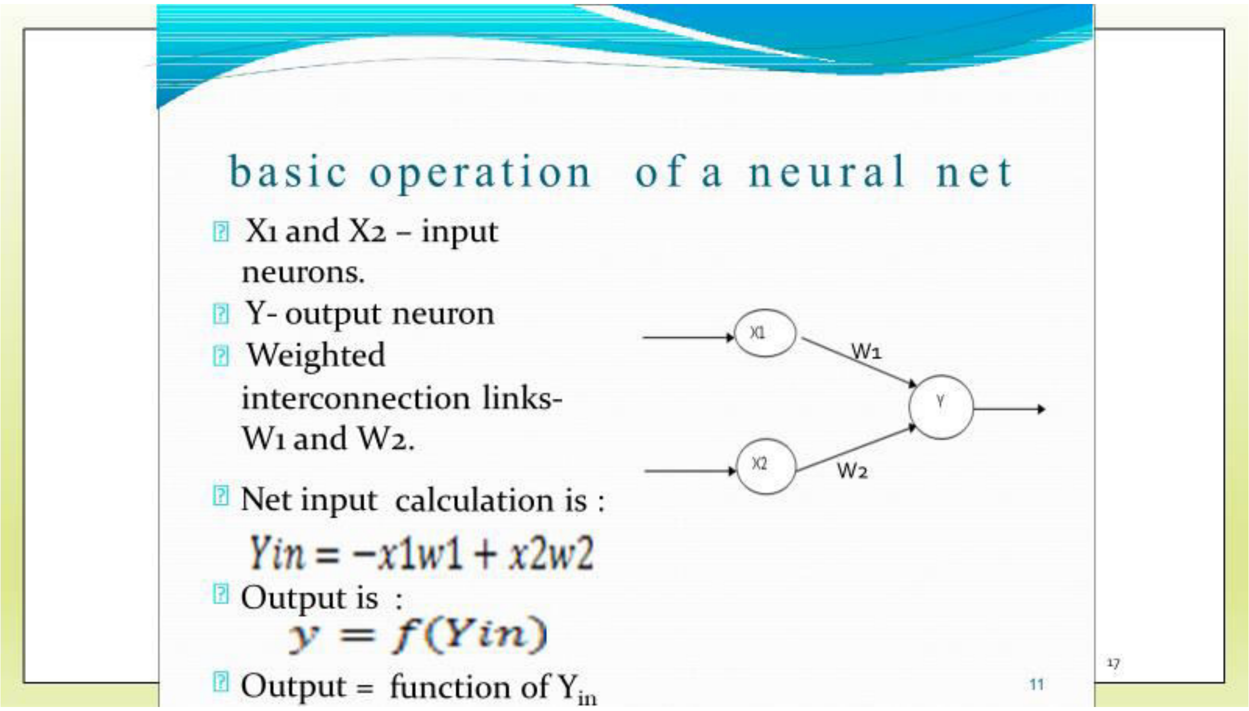
Approximate reasoning

Functional approximation



Artificial neural network : An introduction

- Resembles the characteristic of biological neural network.
- ***Nodes*** – interconnected processing elements (units or neurons)
- Neuron is connected to other by a ***connection link.***
- Each connection link is associated with ***weight*** which has information about the input signal.
- ANN processing elements are called as ***neurons or artificial neurons*** , since they have the capability to model networks of original neurons as found in brain.
- Internal state of neuron is called ***activation or activity level*** of neuron, which is the function of the inputs the neurons receives.
- Neuron can send only one signal at a time.



basic operation of a neural net

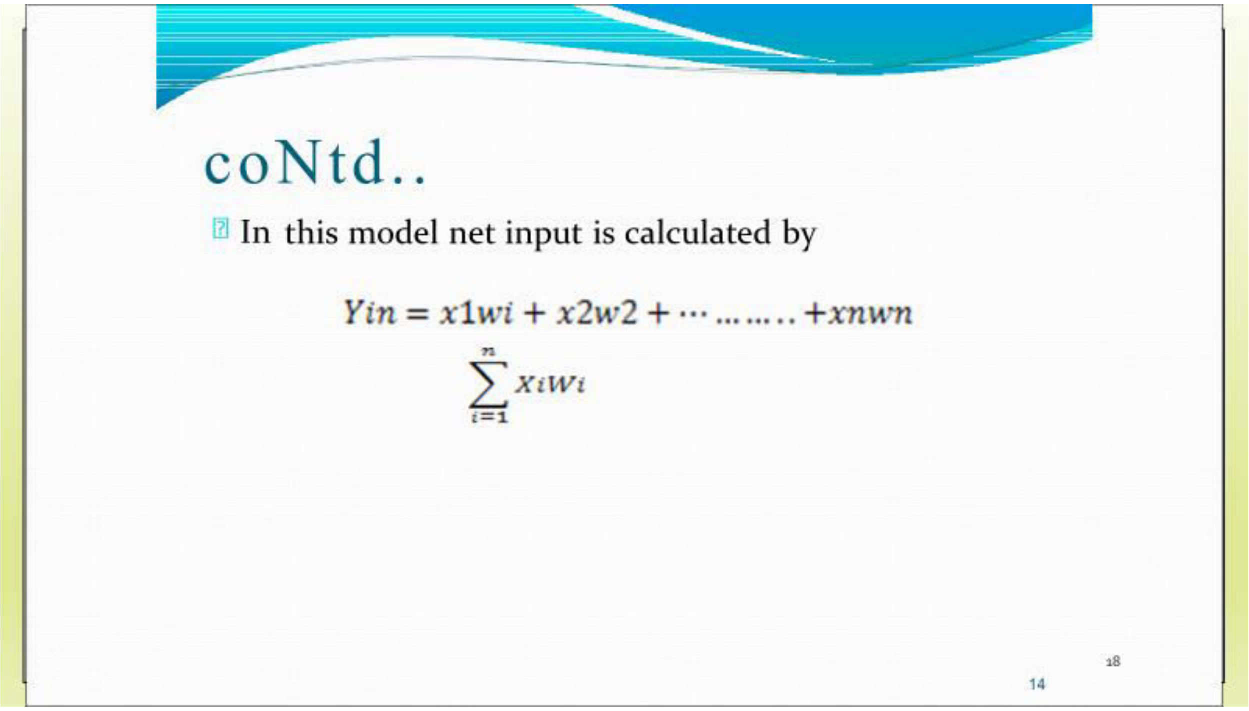
- X1 and X2 – input neurons.
- Y- output neuron
- Weighted interconnection links- W1 and W2.
- Net input calculation is :

$$Yin = -x1w1 + x2w2$$

- Output is :

$$y = f(Yin)$$

- Output = function of Y_{in}

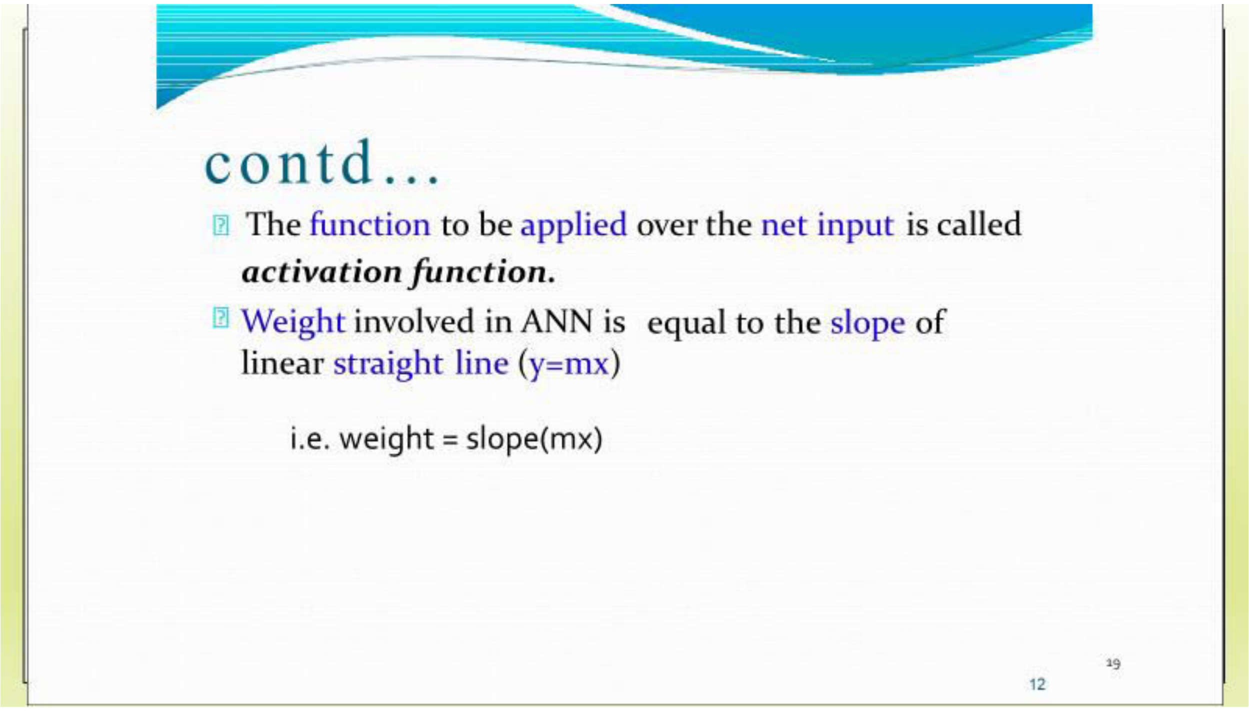


coNtd..

- In this model net input is calculated by

$$Yin = x1wi + x2w2 + \cdots \dots\dots\dots +xnwn$$

$$\sum_{i=1}^{n} XiWi$$

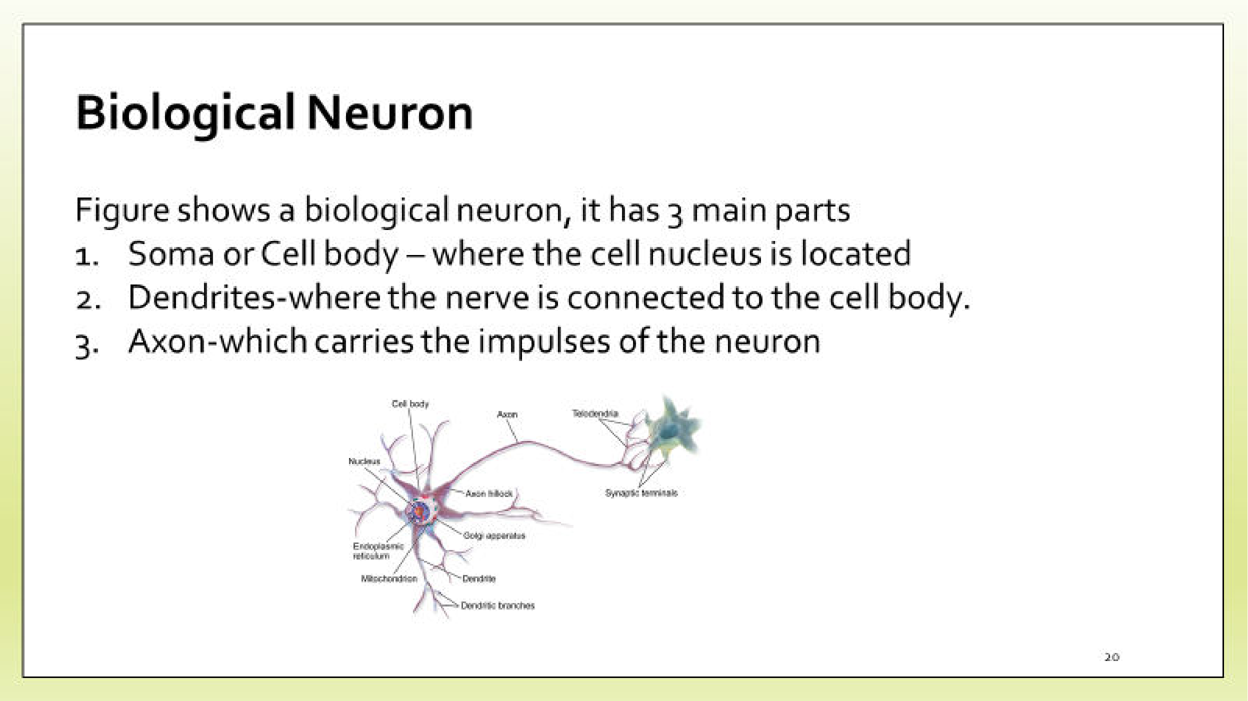


contd...

- The function to be applied over the net input is called ***activation function.***
- Weight involved in ANN is equal to the slope of linear straight line (y=mx)

i.e. weight = slope(mx)

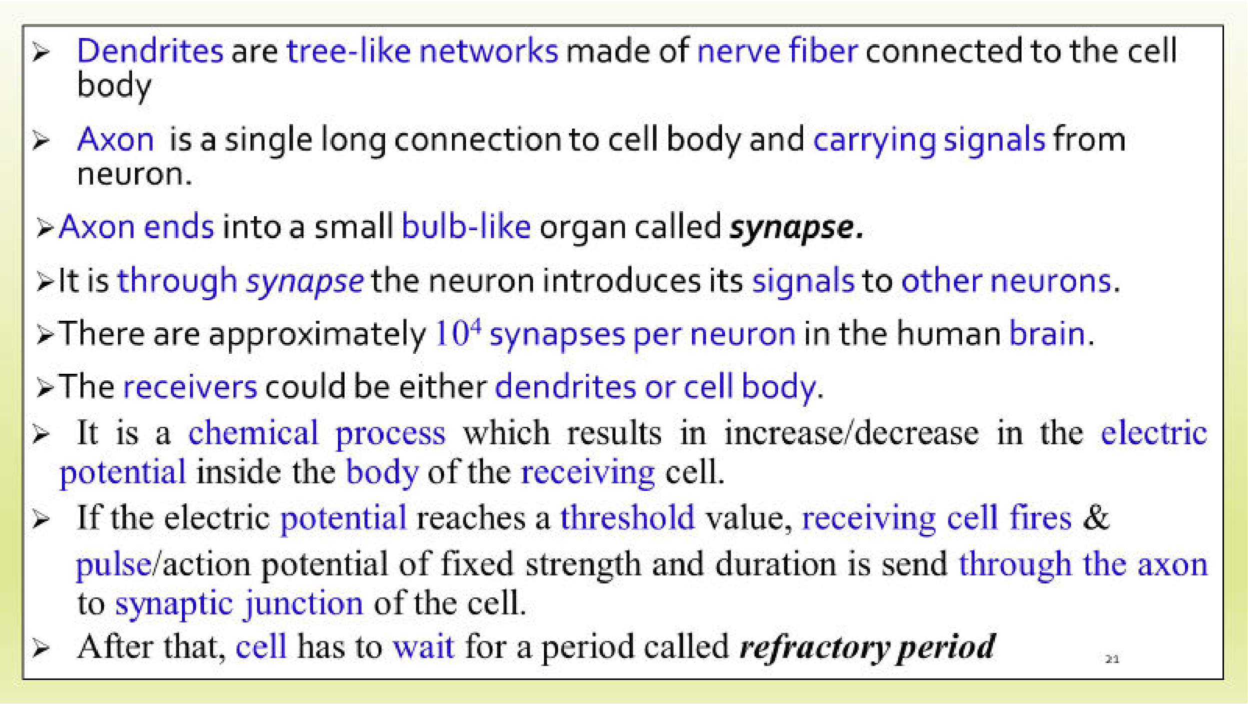
Biological Neuron CO2



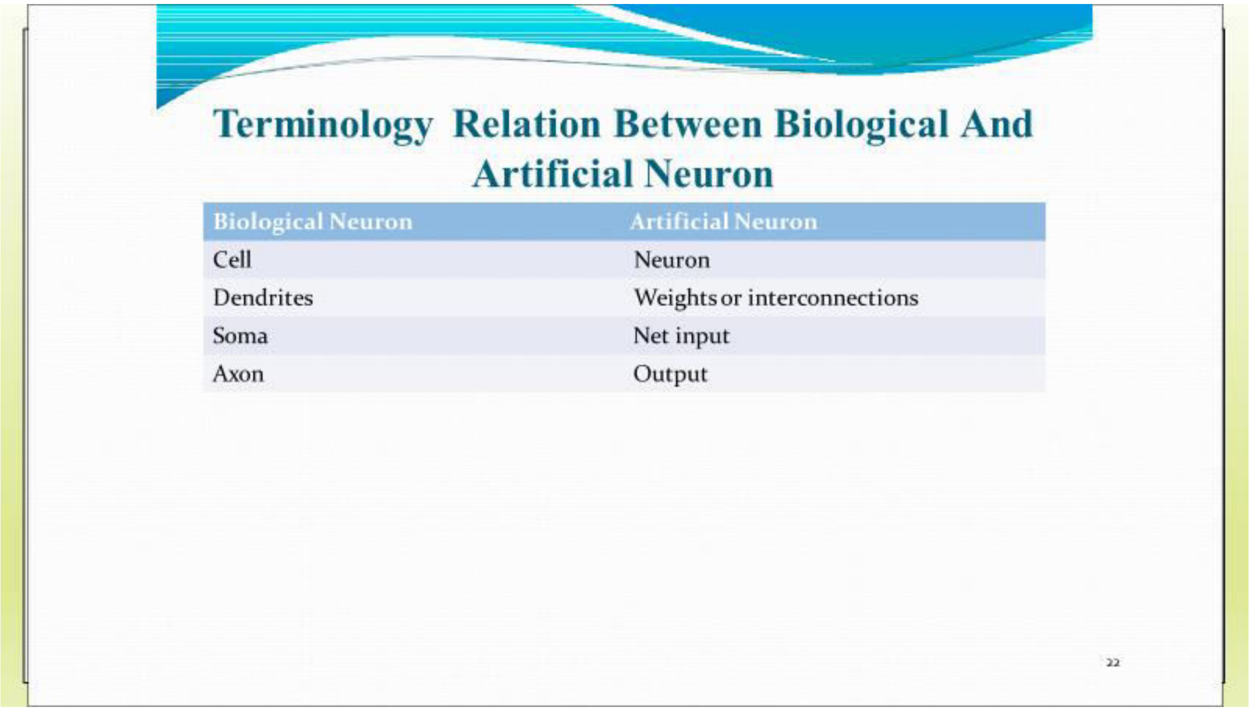
Biological Neuron

Figure shows a biological neuron, it has 3 main parts

1. Soma or Cell body – where the cell nucleus is located
2. Dendrites-where the nerve is connected to the cell body.
3. Axon-which carries the impulses of the neuron

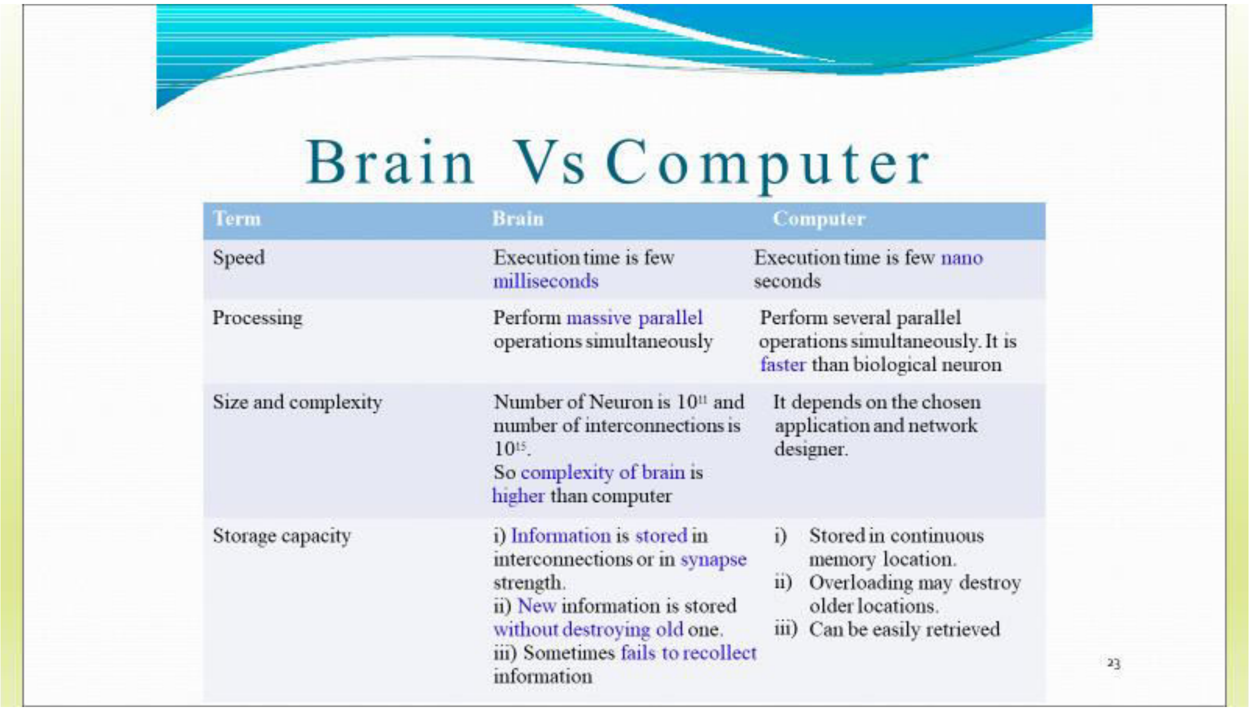


- Dendrites are tree-like networks made of nerve fiber connected to the cell body
- Axon is a single long connection to cell body and carrying signals from neuron.
- Axon ends into a small bulb-like organ called ***synapse.***
- It is through *synapse* the neuron introduces its signals to other neurons.
- There are approximately 10^4 synapses per neuron in the human brain.
- The receivers could be either dendrites or cell body.
- It is a chemical process which results in increase/decrease in the electric potential inside the body of the receiving cell.
- If the electric potential reaches a threshold value, receiving cell fires & pulse/action potential of fixed strength and duration is send through the axon to synaptic junction of the cell.
- After that, cell has to wait for a period called ***refractory period***



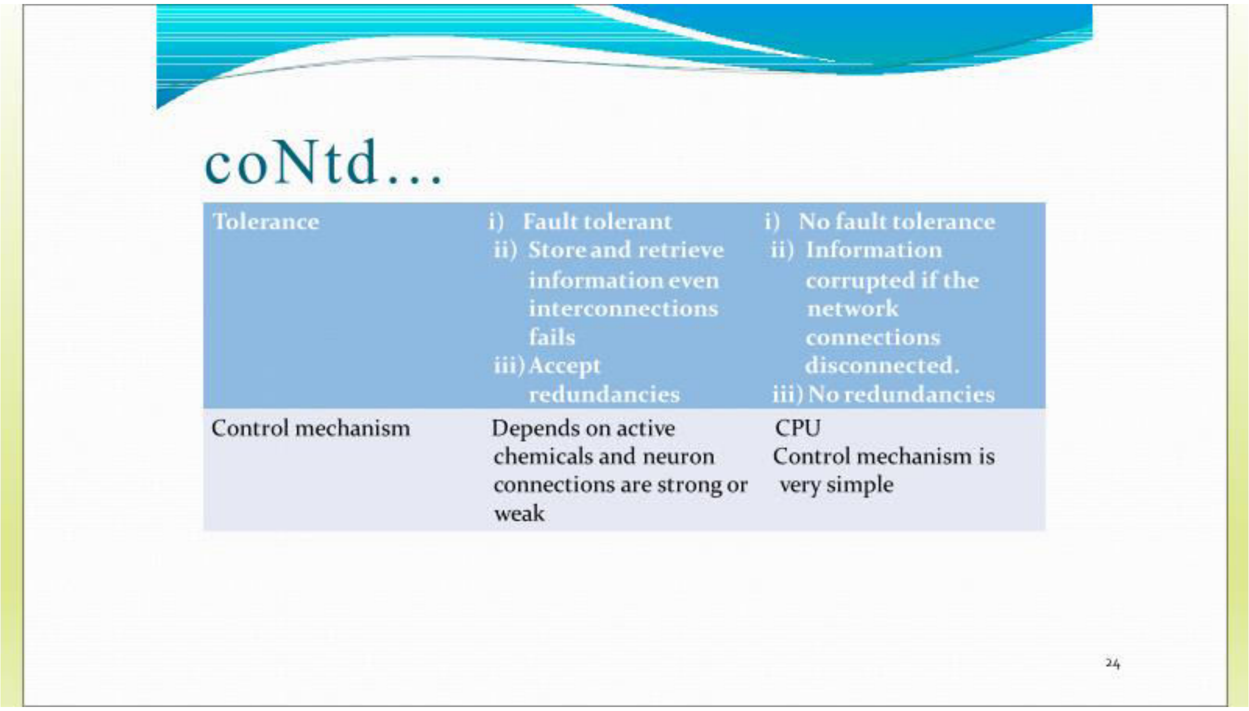
Terminology Relation Between Biological And Artificial Neuron

Biological Neuron	Artificial Neuron
Cell	Neuron
Dendrites	Weights or interconnections
Soma	Net input
Axon	Output



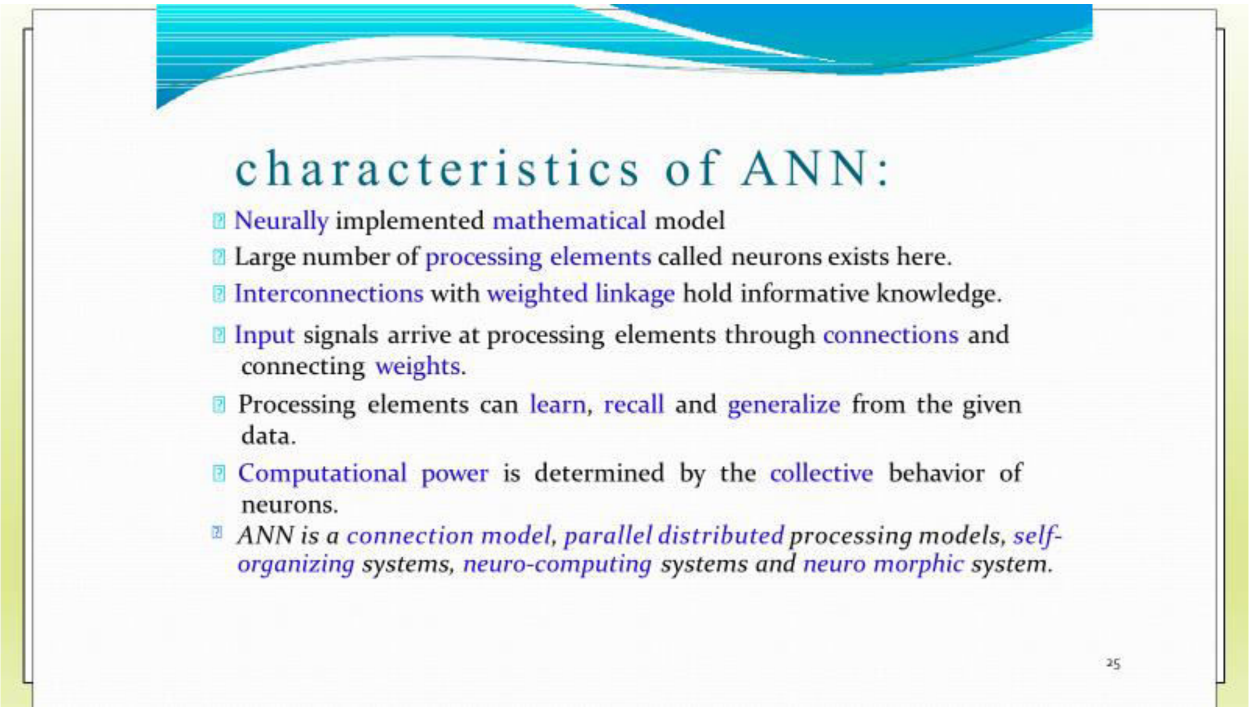
Brain Vs Computer

Term	Brain	Computer
Speed	Execution time is few milliseconds	Execution time is few nano seconds
Processing	Perform massive parallel operations simultaneously	Perform several parallel operations simultaneously. It is faster than biological neuron
Size and complexity	Number of Neuron is 10^{11} and number of interconnections is 10^{15}. So complexity of brain is higher than computer	It depends on the chosen application and network designer.
Storage capacity	i) Information is stored in interconnections or in synapse strength. ii) New information is stored without destroying old one. iii) Sometimes fails to recollect information	i) Stored in continuous memory location. ii) Overloading may destroy older locations. iii) Can be easily retrieved



coNtd…

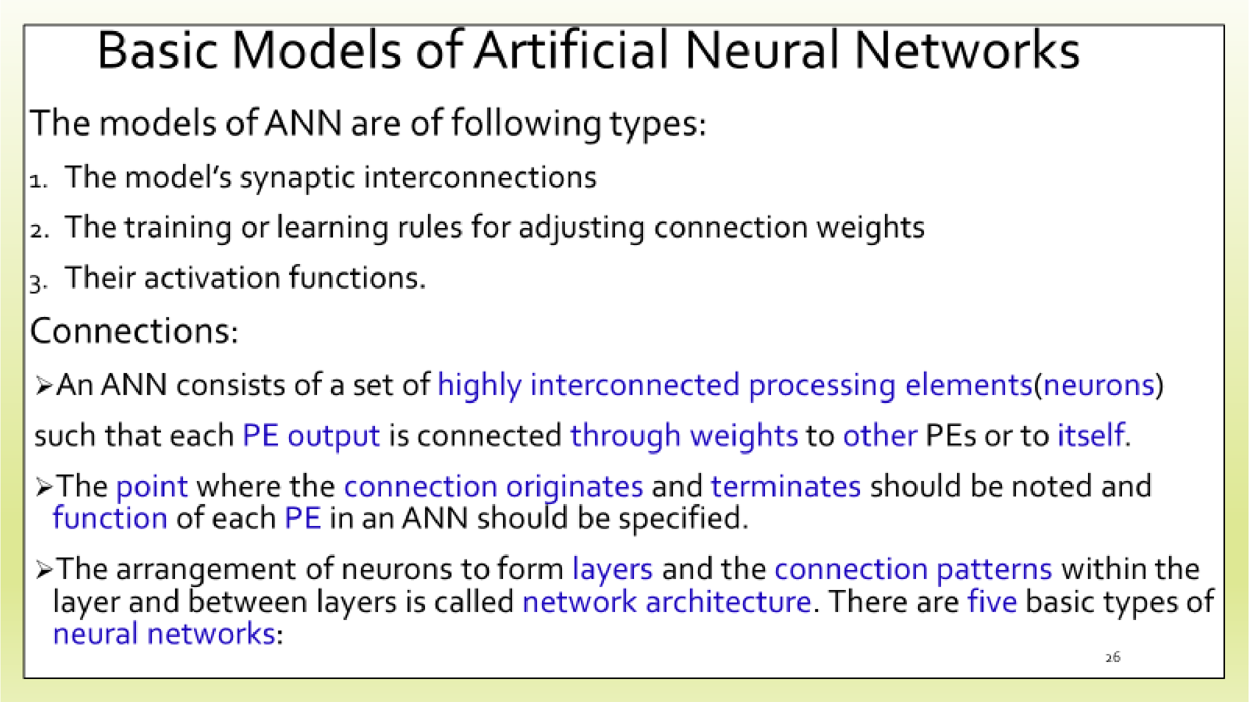
Tolerance	i) Fault tolerant ii) Store and retrieve information even interconnections fails iii) Accept redundancies	i) No fault tolerance ii) Information corrupted if the network connections disconnected. iii) No redundancies
Control mechanism	Depends on active chemicals and neuron connections are strong or weak	CPU Control mechanism is very simple



characteristics of ANN:

- Neurally implemented mathematical model
- Large number of processing elements called neurons exists here.
- Interconnections with weighted linkage hold informative knowledge.
- Input signals arrive at processing elements through connections and connecting weights.
- Processing elements can learn, recall and generalize from the given data.
- Computational power is determined by the collective behavior of neurons.
- *ANN is a connection model, parallel distributed processing models, self-organizing systems, neuro-computing systems and neuro morphic system.*

Basic Models of Neural Networks CO2



Basic Models of Artificial Neural Networks

The models of ANN are of following types:

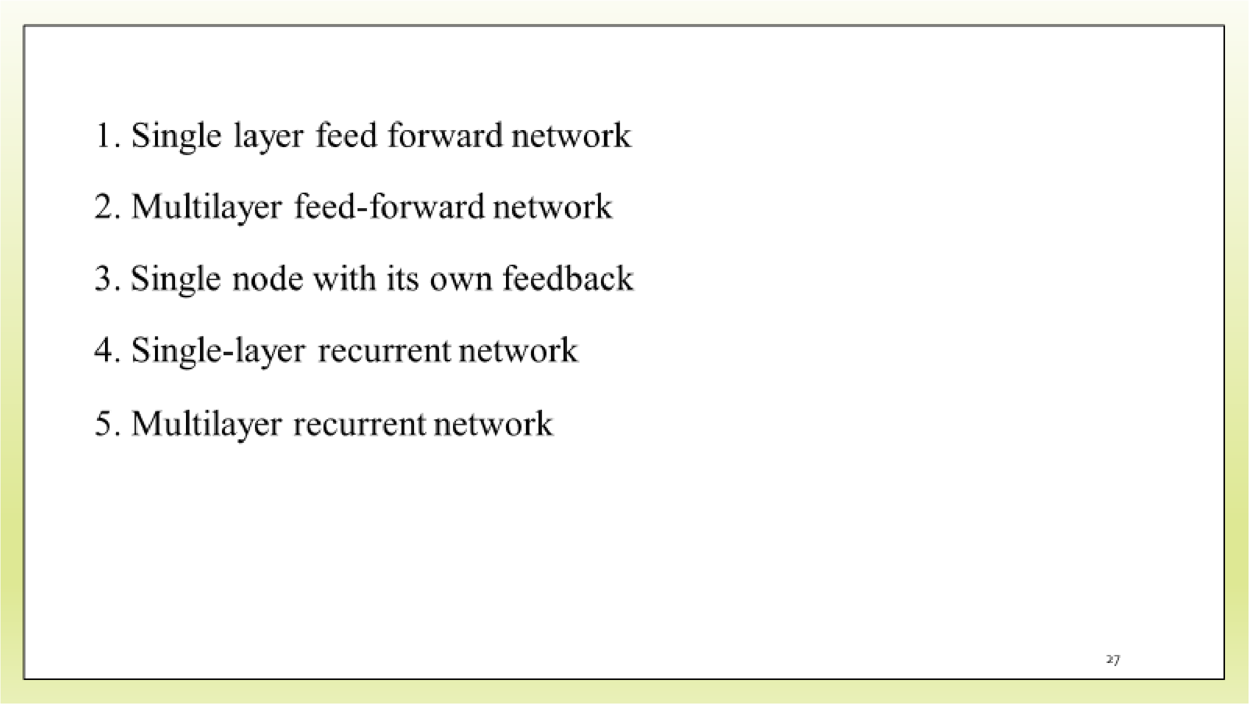
1. The model's synaptic interconnections
2. The training or learning rules for adjusting connection weights
3. Their activation functions.

Connections:

➢An ANN consists of a set of highly interconnected processing elements(neurons) such that each PE output is connected through weights to other PEs or to itself.

➢The point where the connection originates and terminates should be noted and function of each PE in an ANN should be specified.

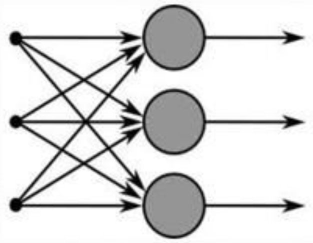
➢The arrangement of neurons to form layers and the connection patterns within the layer and between layers is called network architecture. There are five basic types of neural networks:



1. Single layer feed forward network
2. Multilayer feed-forward network
3. Single node with its own feedback
4. Single-layer recurrent network
5. Multilayer recurrent network

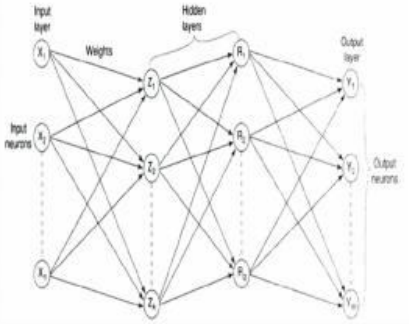
Single layer Feed- Forward Network

- Layer is formed by taking processing elements and combining them with other processing elements.
- Input and output are linked with each other
- Inputs are connected to the processing nodes with various weights, resulting in series of outputs one from each node.



Multilayer feed-forward network

- Formed by the interconnection of several layers.
- Input layer receives input and buffers input signal.
- Output layer generates output.
- Layer between input and output is called *hidden layer.*
- Hidden layer is internal to the network.
- Zero to several hidden layers in a network.
- More the hidden layer, more is the complexity of network, but efficient output is produced.



Single Node with its own Feedback network

- If output is not fed back as an input to a node in the same layer / preceding layer – ***feed forward network.***
- If outputs are directed back as input to the processing elements in the same layer/preceding layer –***feedback network.***
- If the outputs are directed back to the input of the same layer then it is ***lateral feedback.***
- ***Recurrent networks*** are networks with feedback networks with closed loop.
- Fig 2.8 (A) –simple recurrent neural network having a single neuron with feedback to itself.
- Fig 2.9 – single layer network with feedback from output can be directed to processing element itself or to other processing element/both.

Single-layer recurrent network

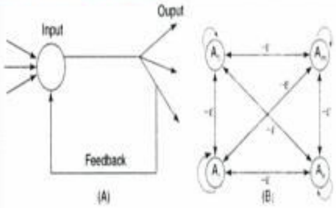


Figure 2-8 (A) Single node with own feedback. (B) Competitive nets.

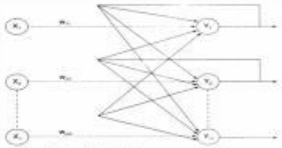


Figure 2.9 Single-layer recurrent network.

Maxnet –competitive interconnections having fixed weights.

- *On-center-off-surround/lateral inhibiton structure* – each processing neuron receives two different classes of inputs- "*excitatory*" input from nearby processing elements & "*inhibitory*" input from more distantly located precessing elements. This type of interconnection is shown below

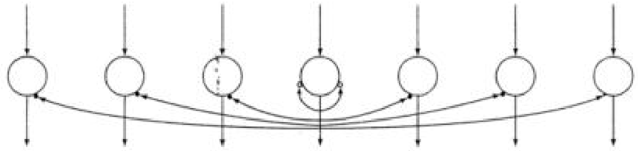
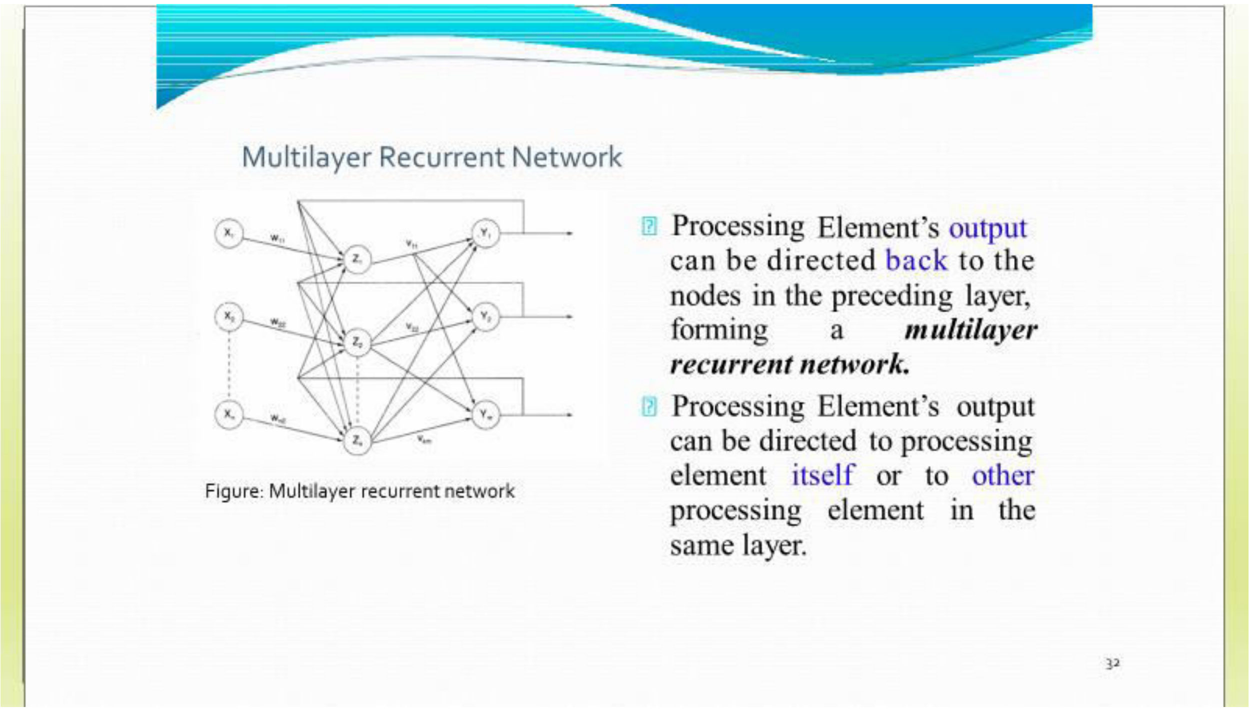


Figure A: Lateral inhibition structure

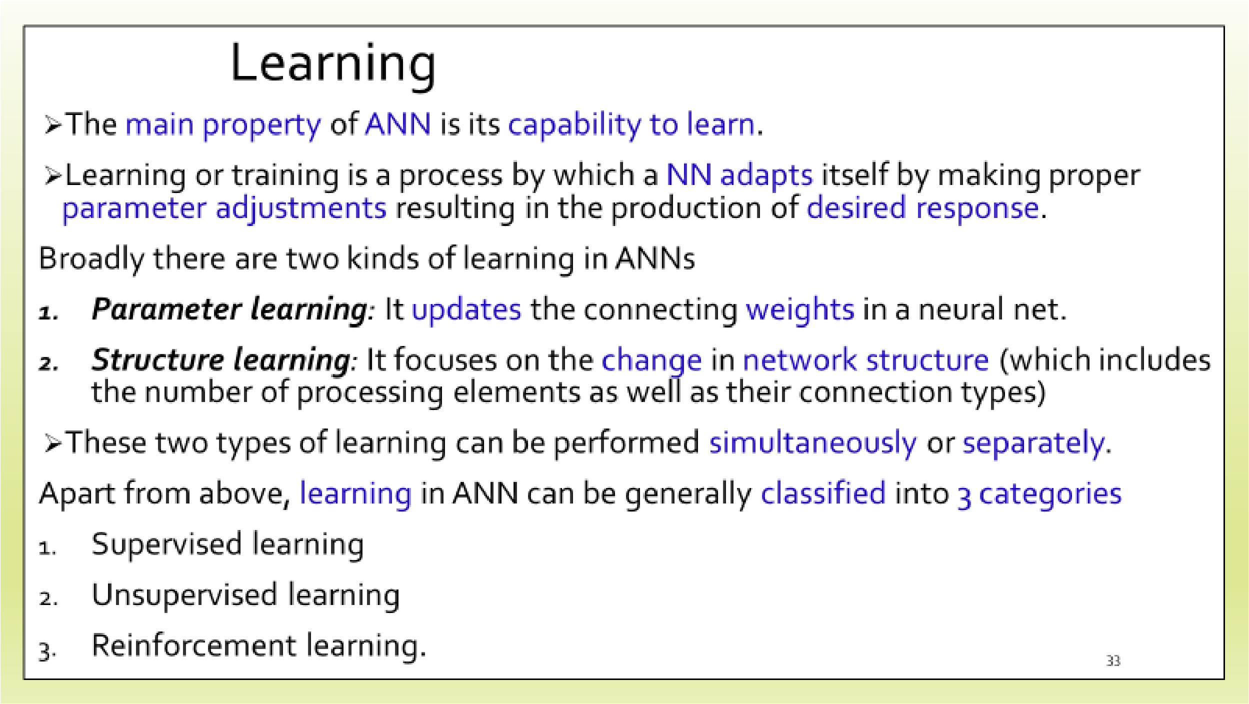


Multilayer Recurrent Network

Figure: Multilayer recurrent network

- Processing Element's output can be directed back to the nodes in the preceding layer, forming a ***multilayer recurrent network.***
- Processing Element's output can be directed to processing element itself or to other processing element in the same layer.

Types of Learning in Neural Network CO2



Learning

➢The main property of ANN is its capability to learn.

➢Learning or training is a process by which a NN adapts itself by making proper parameter adjustments resulting in the production of desired response.

Broadly there are two kinds of learning in ANNs

1. ***Parameter learning**:* It updates the connecting weights in a neural net.
2. ***Structure learning**:* It focuses on the change in network structure (which includes the number of processing elements as well as their connection types)

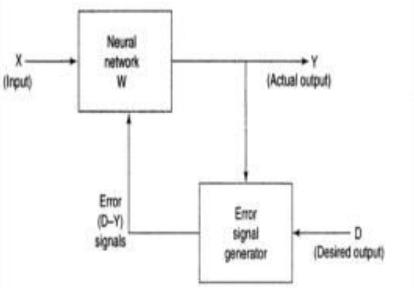
➢These two types of learning can be performed simultaneously or separately.

Apart from above, learning in ANN can be generally classified into 3 categories

1. Supervised learning
2. Unsupervised learning
3. Reinforcement learning.

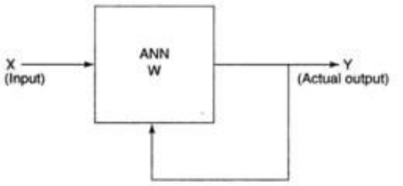
Supervised learning

- Learning with the help of a teacher.
- Example : learning process of a small child.
  - Child doesn't know read/write.
  - Their each & every action is supervised by a teacher
- In ANN, each input vector requires a corresponding target vector, which represents the desired output.
- The input vector along with target vector is called *training pair*.
- The input vector results in output vector.
- The actual output vector is compared with desired output vector.
- If there is a difference then an error signal is generated by the network.
- It is used for adjustment of weights until actual output matches with desired output.

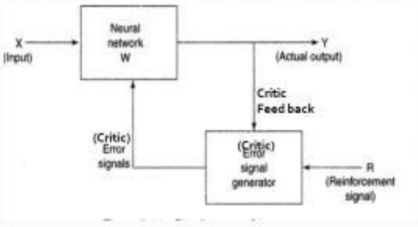


unsupervised learning

- Learning is performed without the help of a teacher.
- Example: tadpole – learn to swim by itself.
- In ANN, during training process, network receives input patterns and organize them to form clusters, based on similarities of input.
- From the Fig. it is observed that no feedback is applied from environment to inform what output should be or whether it is correct or not.
- The n/w itself should find out the output based on relations for the input data over the output. This is called *self-organizing* network



Reinforcement learning



- Similar to supervised learning.
- Learning based on *critic information* is called *reinforcement learning* the feedback sent is called *reinforcement signal.*
- The network receives some feedback from the environment.
- Feedback is only evaluative.
- The external reinforcement signals are processed in the critic signal generator, and the obtained critic Signal is sent to the ANN for adjustment of weights properly to get critic feedback in future.

Activation Function

- To make work more efficient and for exact output, some force or activation is given.
- Like that, activation function is applied over the net input to calculate the output of an ANN.
- Information processing of processing element has two major parts: input and output.
- An integration function (*f*) is associated with input of processing element.
- Several activation functions are there.

1. Identity function:

- it is a linear function which is defined as

  $f(x) = x$ *for all* x

- The output is same as the input.

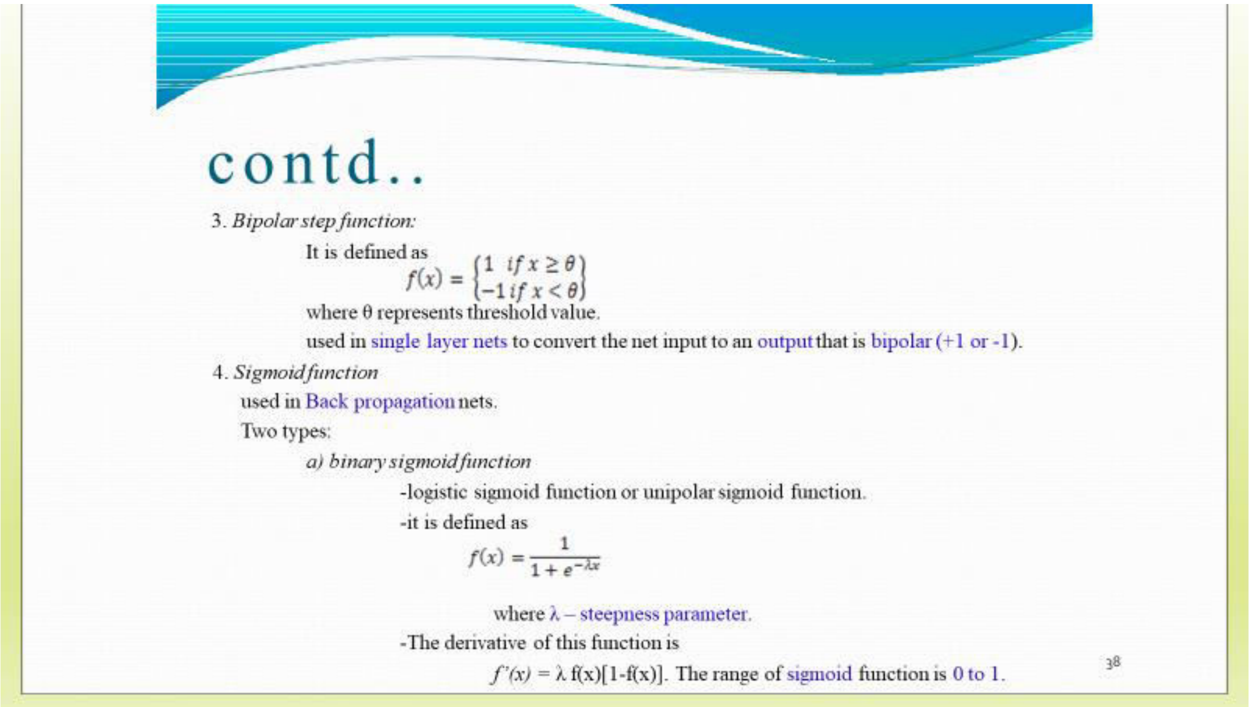
2. Binary step function

- it is defined as

$$f(x) = \begin{Bmatrix} 1 \ \ if\, x \geq \theta \\ 0 \ \ if\, x < \theta \end{Bmatrix}$$

where θ represents threshold value.

It is used in single layer nets to convert the net input to an output that is binary (0 or 1)



contd..

3. *Bipolar step function:*

It is defined as

$$f(x) = \begin{Bmatrix} 1 \ \ if\, x \geq \theta \\ -1\, if\, x < \theta \end{Bmatrix}$$

where θ represents threshold value.

used in single layer nets to convert the net input to an output that is bipolar (+1 or -1).

4. *Sigmoid function*

used in Back propagation nets.

Two types:

a) binary sigmoid function

-logistic sigmoid function or unipolar sigmoid function.

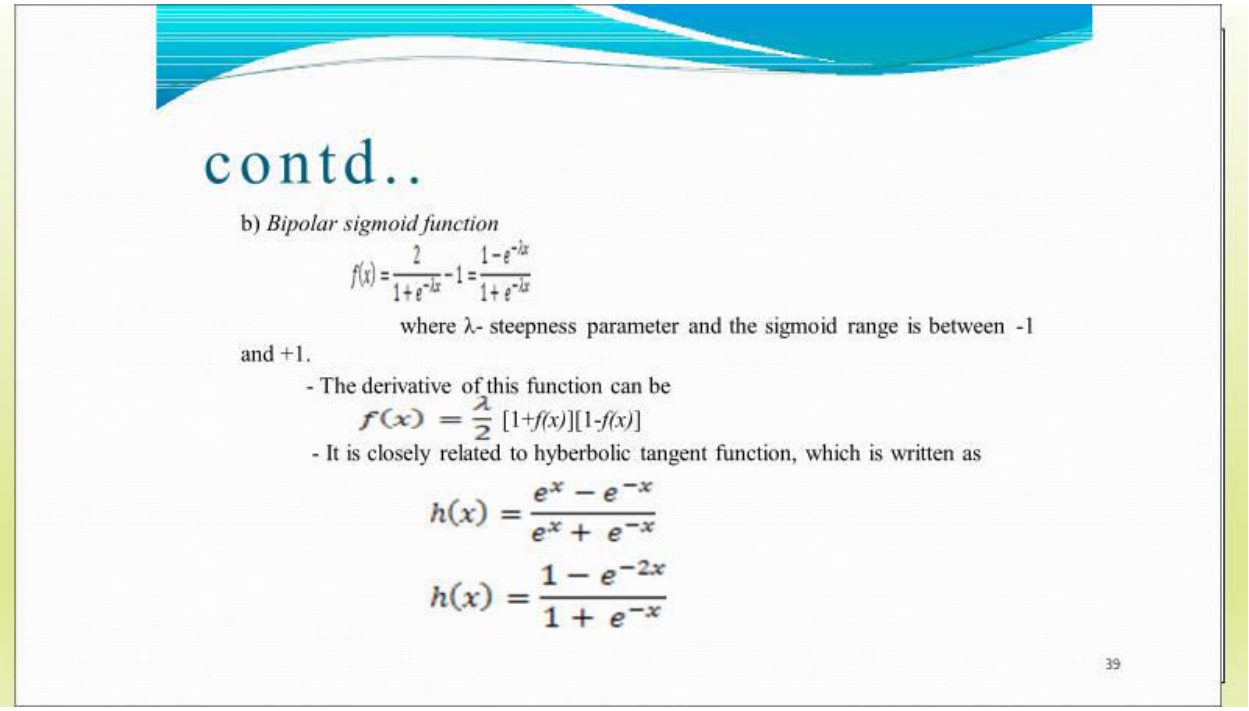
-it is defined as

$$f(x) = \frac{1}{1 + e^{-\lambda x}}$$

where λ – steepness parameter.

-The derivative of this function is

$f'(x) = \lambda$ f(x)[1-f(x)]. The range of sigmoid function is 0 to 1.



contd..

b) *Bipolar sigmoid function*

$$f(x) = \frac{2}{1+e^{-\lambda x}} - 1 = \frac{1-e^{-\lambda x}}{1+e^{-\lambda x}}$$

where λ- steepness parameter and the sigmoid range is between -1 and +1.

- The derivative of this function can be

$$f(x) = \frac{\lambda}{2}\,[1+f(x)][1-f(x)]$$

- It is closely related to hyberbolic tangent function, which is written as

$$h(x) = \frac{e^{x} - e^{-x}}{e^{x} + e^{-x}}$$

$$h(x) = \frac{1 - e^{-2x}}{1 + e^{-x}}$$

contd..

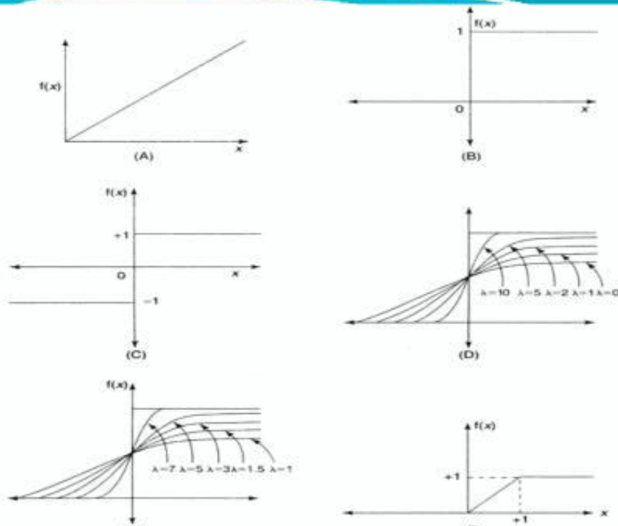
The derivative of the hyberbolic tangent function is

$$H'[1+h(x))][1-h(x)]$$

5. Ramp function

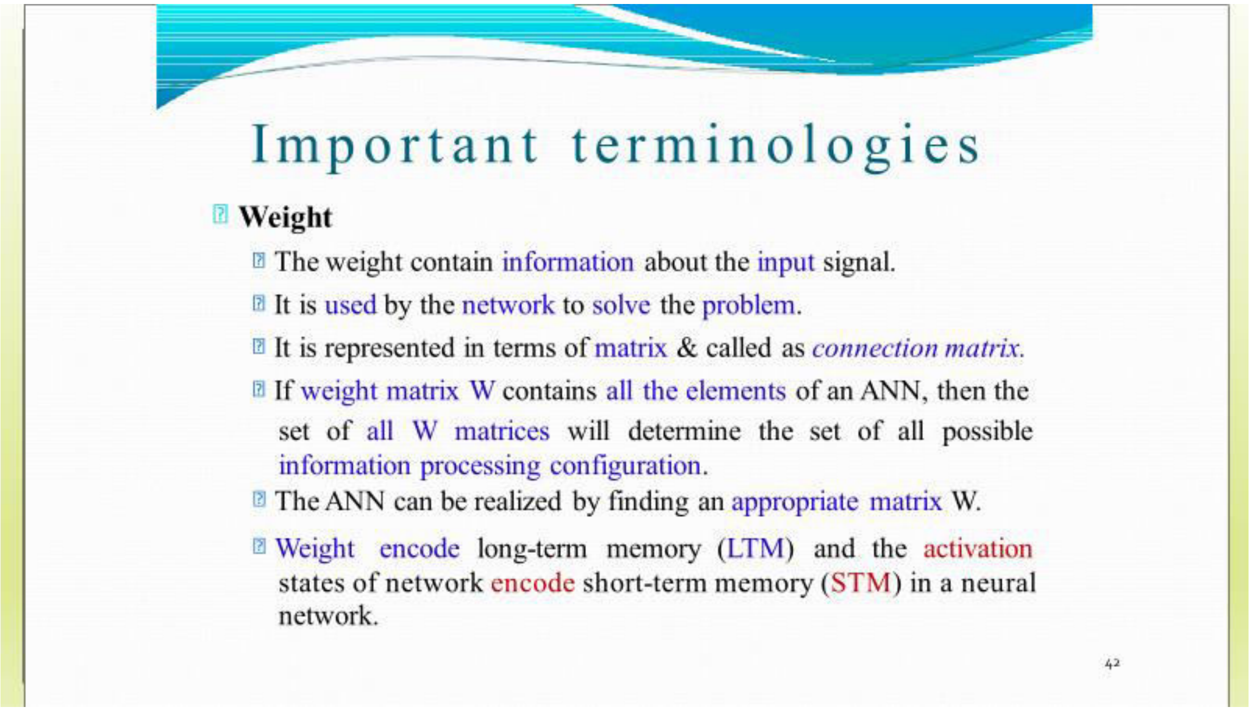
$$f(x) = \begin{Bmatrix} 1 \ \ if\, x > 1 \\ x \ \ if\, 0 \leq x \leq 1 \\ 0\, if\ x < 0 \end{Bmatrix}$$

The graphical representation of all these function is given in the upcoming Figure



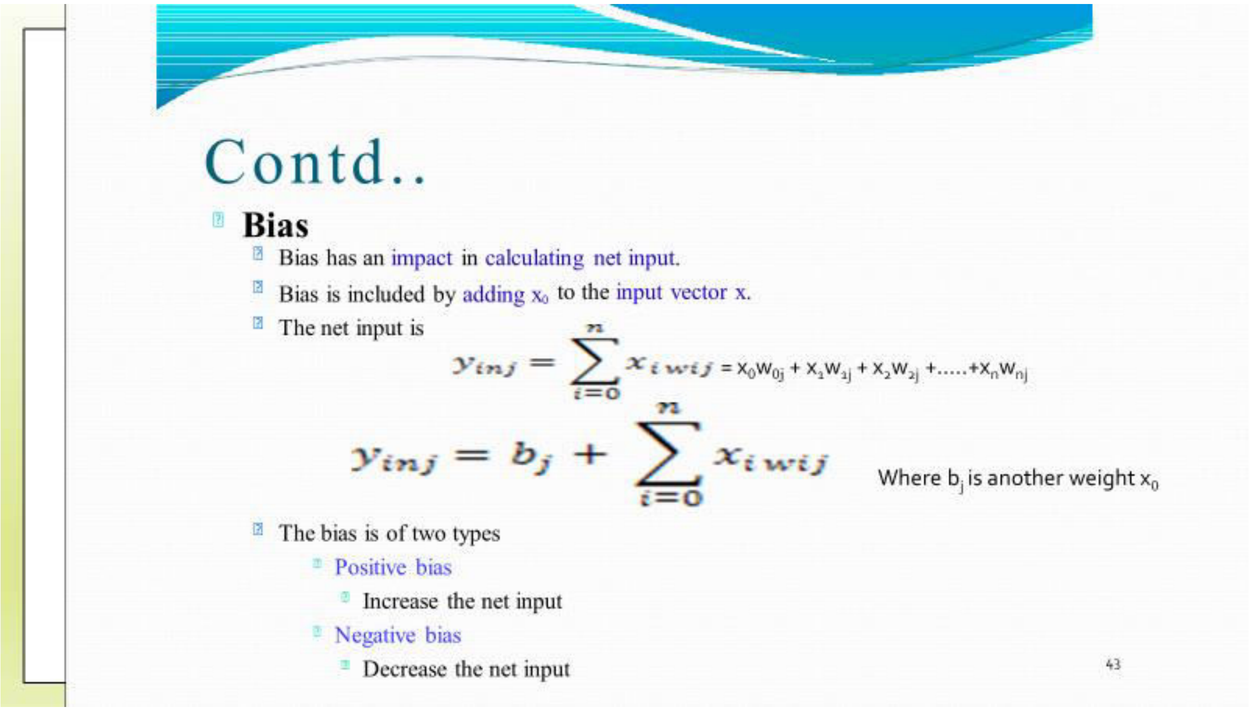
2-15 Depiction of activation functions: (A) identity function; (B) binary step function; (C) bipolar step function; (D) binary sigmoidal function; (E) bipolar sigmoidal function; (F) ramp function.

Terminologies used in Neural Networks CO2



Important terminologies

- **Weight**
  - The weight contain information about the input signal.
  - It is used by the network to solve the problem.
  - It is represented in terms of matrix & called as *connection matrix.*
  - If weight matrix W contains all the elements of an ANN, then the set of all W matrices will determine the set of all possible information processing configuration.
  - The ANN can be realized by finding an appropriate matrix W.
  - Weight encode long-term memory (LTM) and the activation states of network encode short-term memory (STM) in a neural network.



Contd..

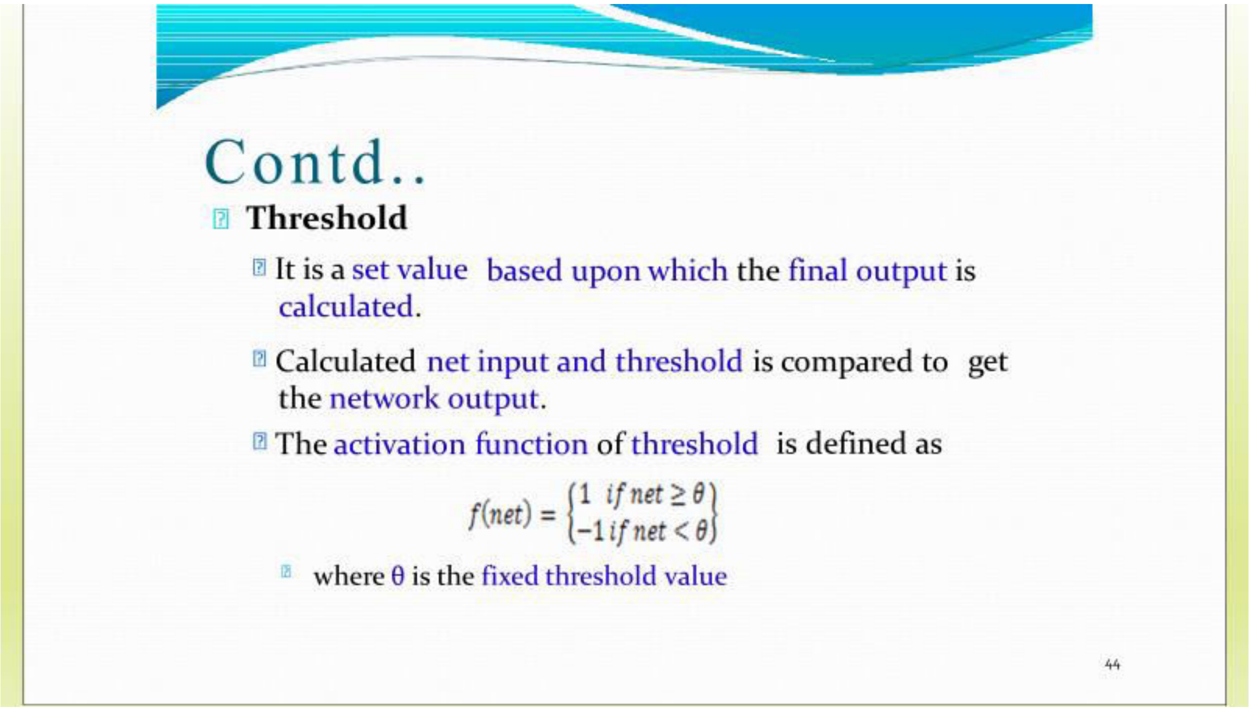
- **Bias**
  - Bias has an impact in calculating net input.
  - Bias is included by adding x_0 to the input vector x.
  - The net input is

$$y_{inj} = \sum_{i=0}^{n} x_{i}w_{ij} = x_0w_{0j} + x_1w_{1j} + x_2w_{2j} + \ldots\ldots + x_nw_{nj}$$

$$y_{inj} = b_j + \sum_{i=0}^{n} x_{i}w_{ij}$$

Where b_j is another weight x_0

  - The bias is of two types
    - Positive bias
      - Increase the net input
    - Negative bias
      - Decrease the net input

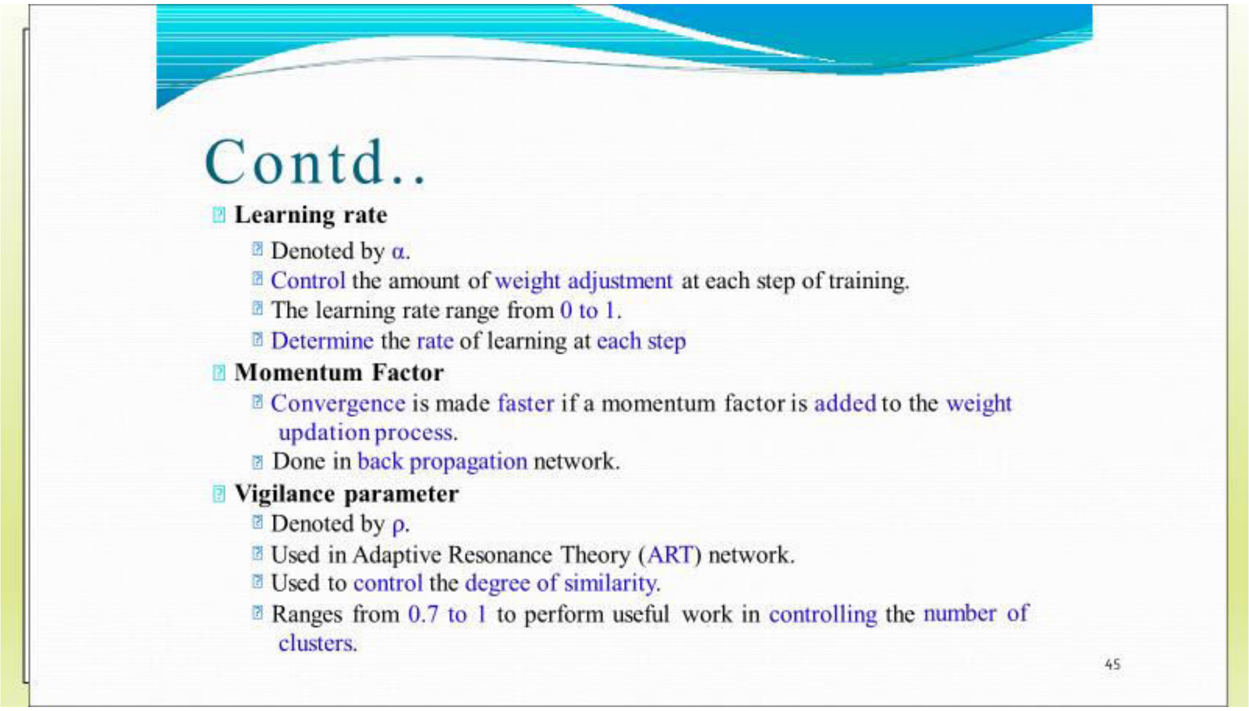


Contd..

- **Threshold**
  - It is a set value based upon which the final output is calculated.
  - Calculated net input and threshold is compared to get the network output.
  - The activation function of threshold is defined as

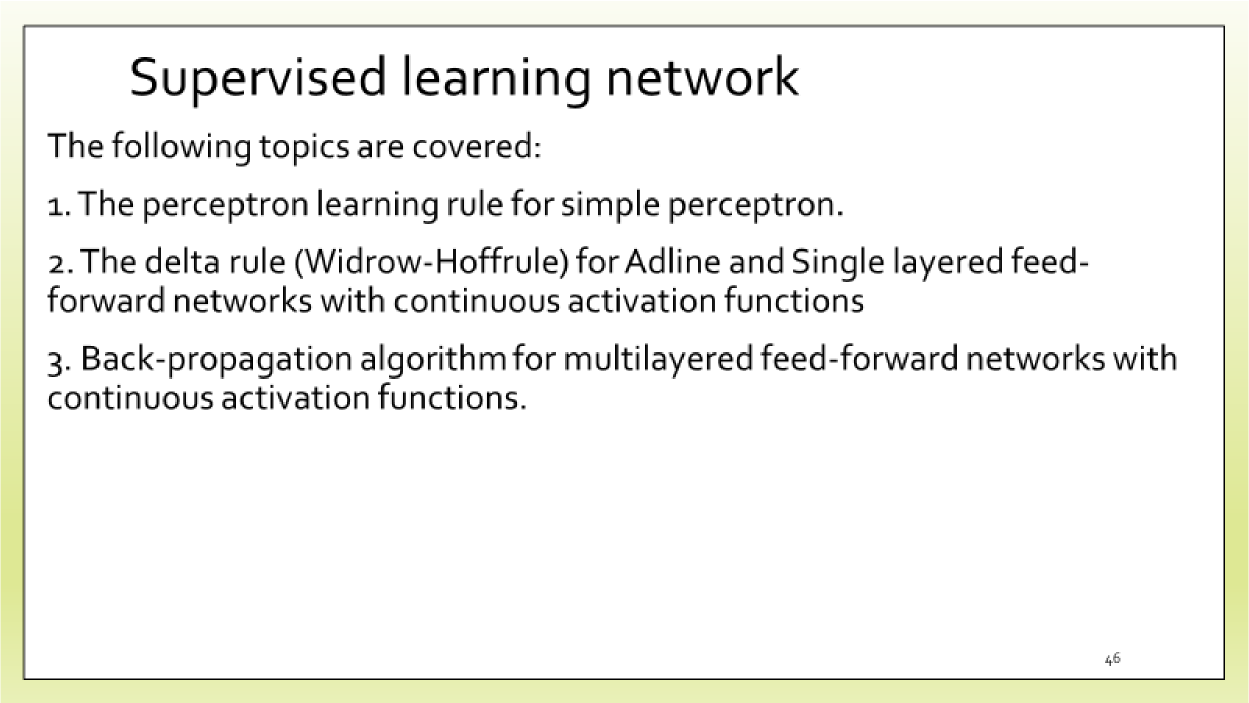
$$f(net) = \begin{Bmatrix} 1 \ \ if\, net \geq \theta \\ -1\, if\, net < \theta \end{Bmatrix}$$

  - where θ is the fixed threshold value



Contd..

- **Learning rate**
  - Denoted by α.
  - Control the amount of weight adjustment at each step of training.
  - The learning rate range from 0 to 1.
  - Determine the rate of learning at each step
- **Momentum Factor**
  - Convergence is made faster if a momentum factor is added to the weight updation process.
  - Done in back propagation network.
- **Vigilance parameter**
  - Denoted by ρ.
  - Used in Adaptive Resonance Theory (ART) network.
  - Used to control the degree of similarity.
  - Ranges from 0.7 to 1 to perform useful work in controlling the number of clusters.



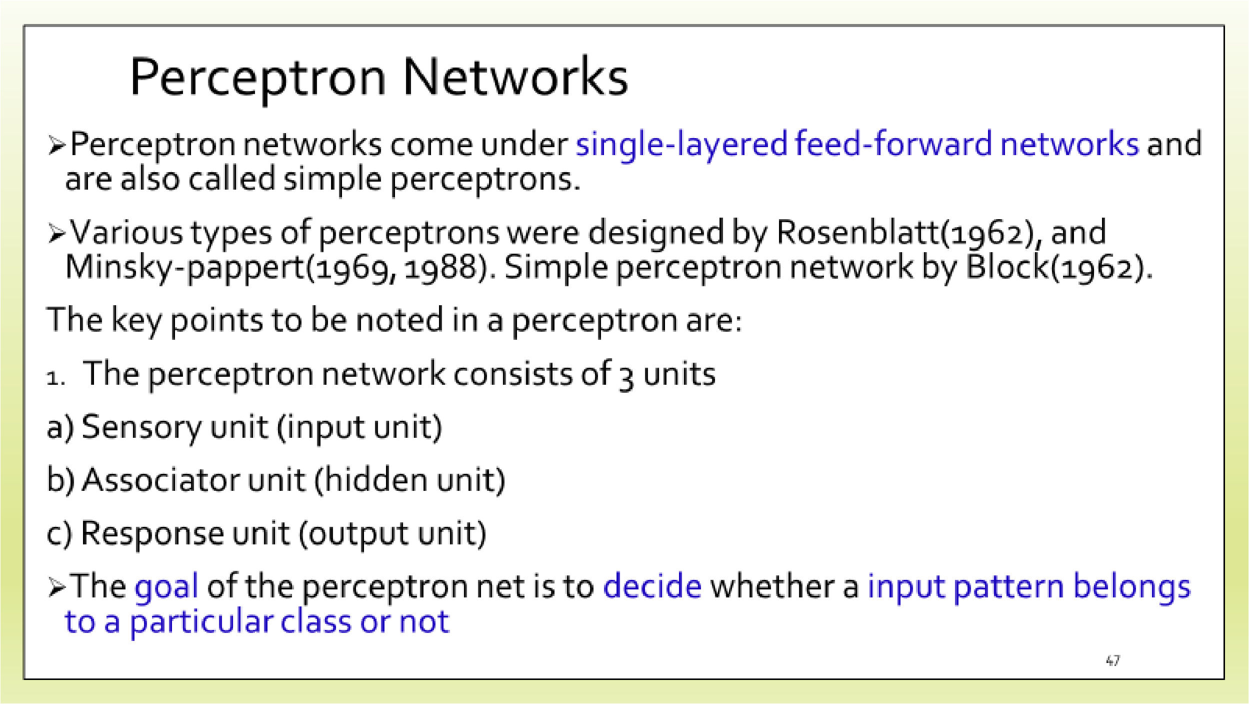
Supervised learning network

The following topics are covered:

1. The perceptron learning rule for simple perceptron.

2. The delta rule (Widrow-Hoffrule) for Adline and Single layered feed-forward networks with continuous activation functions

3. Back-propagation algorithm for multilayered feed-forward networks with continuous activation functions.



Perceptron Networks

- Perceptron networks come under single-layered feed-forward networks and are also called simple perceptrons.
- Various types of perceptrons were designed by Rosenblatt(1962), and Minsky-pappert(1969, 1988). Simple perceptron network by Block(1962).

The key points to be noted in a perceptron are:

1. The perceptron network consists of 3 units

a) Sensory unit (input unit)

b) Associator unit (hidden unit)

c) Response unit (output unit)

- The goal of the perceptron net is to decide whether a input pattern belongs to a particular class or not

2. The sensory units are connected to associator units with fixed weights having values 1, 0 or −1, which are assigned at random.
3. The binary activation function is used in sensory unit and associator unit.
4. The response unit has an activation of 1, 0 or −1. The binary step with fixed threshold θ is used as activation for associator. The output signals that are sent from the associator unit to the response unit are only binary.
5. The output of the perceptron network is given by

$$y = f(y_{in})$$

where $f(y_{in})$ is activation function and is defined as

$$f(y_{in}) = \begin{cases} 1 & \text{if } y_{in} > \theta \\ 0 & \text{if } -\theta \le y_{in} \le \theta \\ -1 & \text{if } y_{in} < -\theta \end{cases}$$

6. The perceptron learning rule is used in the weight updation between the associator unit and the response unit. For each training input, the net will calculate the response and it will determine whether or not an error has occurred.
7. The error calculation is based on the comparison of the values of targets with those of the calculated outputs.
8. The weights on the connections from the units that send the nonzero signal will get adjusted suitably.
9. The weights will be adjusted on the basis of the learning rule if an error has occurred for a particular training pattern, i.e.,

$$w_i(\text{new}) = w_i(\text{old}) + \alpha t x_i$$
$$b(\text{new}) = b(\text{old}) + \alpha t$$

If no error occurs, there is no weight updation and hence the training process may be stopped. In the above equations, the target value "t" is +1 or −1 and α is the learning rate. In general, these learning rules begin with an initial guess at the weight values and then successive adjustments are made on the basis of the evaluation of an objective function. Eventually, the learning rules reach a near-optimal or optimal solution in a finite number of steps.

A perceptron network with its three units is shown in Figure 3-1. As shown in Figure 3-1, a sensory unit can be a two-dimensional matrix of 400 photodetectors upon which a lighted picture with geometric black and white pattern impinges. These detectors provide a binary (0) electrical signal if the input signal is found to exceed a certain value of threshold. Also, these detectors are connected randomly with the associator unit. The associator unit is found to consist of a set of subcircuits called *feature predicates*. The feature predicates are hard-wired to detect the specific feature of a pattern and are equivalent to the *feature detectors*. For a particular feature, each predicate is examined with a few or all of the responses of the sensory unit. It can be found that the results from the predicate units are also binary (0 or 1). The last unit, i.e. response unit, contains the pattern-recognizers or perceptrons. The weights present in the input layers are all fixed, while the weights on the response unit are trainable.

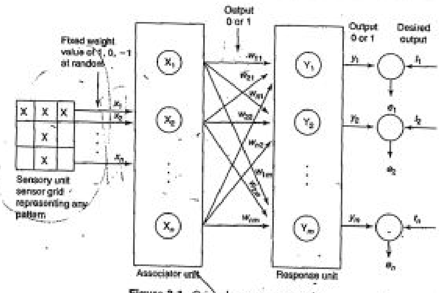


Figure 3-1 Original perceptron network.

binary step with θ is used as activation.

3.2.2 Perceptron Learning Rule

In case of the perceptron learning rule, the learning signal is the difference between the desired and actual response of a neuron. The perceptron learning rule is explained as follows: (learning sig = DE-AR)

Consider a finite "n" number of input training vectors, with their associated target (desired) values $x(n)$ and $t(n)$, where "n" ranges from 1 to N. The target is either +1 or −1. The output "y" is obtained on the basis of the net input calculated and activation function being applied over the net input.

$$y = f(y_{in}) = \begin{cases} 1 & \text{if } y_{in} > \theta \\ 0 & \text{if } -\theta \le y_{in} \le \theta \\ -1 & \text{if } y_{in} < -\theta \end{cases}$$

f(x(n)W) t(n)

The weight updation in case of perceptron learning is as shown.

If $y \ne t$, then

$$w(\text{new}) = w(\text{old}) + \alpha t x \quad (\alpha - \text{learning rate})$$

else, we have

$$w(\text{new}) = w(\text{old})$$

The weights can be initialized at any values in this method. The perceptron rule convergence theorem states that "If there is a weight vector W such that $f(x(n)W) = t(n)$, for all n, then for any starting vector w_1, the perceptron learning rule will converge to a weight vector that gives the correct response for all training patterns, and this learning takes place within a finite number of steps provided that the solution exists."

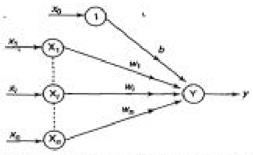


Figure 3-2 Single classification perceptron network.

3.2.3 Architecture

In the original perceptron network, the output obtained from the associator unit is a binary vector, and hence that output can be taken as input signal to the response unit, and classification can be performed. Here only the weights between the associator unit and the output unit can be adjusted, and the weights between the sensory and associator units are fixed. As a result, the discussion of the network is limited to a single portion. Thus, the associator unit behaves like the input unit. A simple perceptron network architecture is shown in Figure 3-2.

In Figure 3-2, there are n input neurons, 1 output neuron and a bias. The input-layer and output-layer neurons are connected through a directed communication link, which is associated with weights. The goal of the perceptron net is to classify the input pattern as a member or not a member to a particular class.

Goal:=> classify input pattern as a member or not

3.2.4 Flowchart for Training Process

The flowchart for the perceptron network training is shown in Figure 3-3. The network has to be suitably trained to obtain the response. The flowchart depicted here presents the flow of the training process.

As depicted in the flowchart, first the basic initialization required for the training process is performed. The entire loop of the training process continues until the training input pair is presented to the network. The training (weight updation) is done on the basis of the comparison between the calculated and desired output. The loop is terminated if there is no change in weight.

3.2.5 Perceptron Training Algorithm for Single Output Classes

The perceptron algorithm can be used for either binary or bipolar input vectors, having bipolar targets, threshold being fixed and variable bias. The algorithm discussed in this section is not particularly sensitive to the initial values of the weights or the value of the learning rate. In the algorithm discussed below, initially the inputs are assigned. Then the net input is calculated. The output of the network is obtained by applying the activation function over the calculated net input. On performing comparison over the calculated and

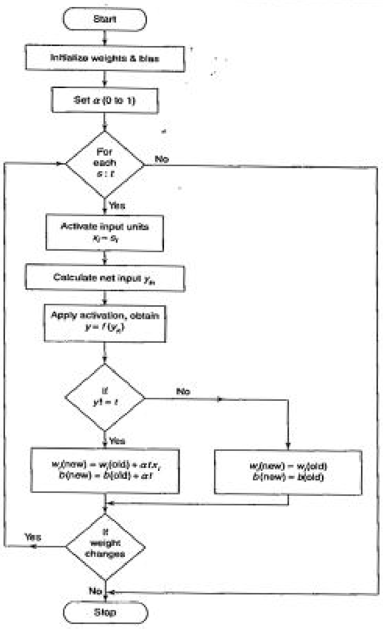

Start
Initialize weights & bias
Set α (0 to 1)
For each s : t
No
Yes
Activate input units $x_i = s_i$
Calculate net input y_{in}
Apply activation, obtain $y = f(y_{in})$
If y ! = t
No
Yes
$w_i(\text{new}) = w_i(\text{old}) + \alpha t x_i$
$b(\text{new}) = b(\text{old}) + \alpha t$
$w_i(\text{new}) = w_i(\text{old})$
$b(\text{new}) = b(\text{old})$
If weight changes
Yes
No
Stop

Figure 3-3 Flowchart for perceptron network with single output.

the desired output, the weight updation process is carried out. The entire network is trained based on the mentioned stopping criterion. The algorithm of a perceptron network is as follows:

Step 0: Initialize the weights and the bias (for easy calculation they can be set to zero). Also initialize the learning rate $\alpha(0 < \alpha \leq 1)$. For simplicity α is set to 1.

Step 1: Perform Steps 2–6 until the final stopping condition is false.

Step 2: Perform Steps 3–5 for each training pair indicated by $s:t$.

Step 3: The input layer containing input units is applied with identity activation functions:

$$x_i = s_i$$

Step 4: Calculate the output of the network. To do so, first obtain the net input:

$$y_{in} = b + \sum_{i=1}^{n} x_i w_i$$

where "n" is the number of input neurons in the input layer. Then apply activations over the net input calculated to obtain the output:

$$y = f(y_{in}) = \begin{cases} 1 & \text{if } y_{in} > \theta \\ 0 & \text{if } -\theta \leq y_{in} \leq \theta \\ -1 & \text{if } y_{in} < -\theta \end{cases}$$

Step 5: *Weight and bias adjustment:* Compare the value of the actual (calculated) output and desired (target) output.

If $y \neq t$, then

$$w_i(\text{new}) = w_i(\text{old}) + \alpha t x_i$$
$$b(\text{new}) = b(\text{old}) + \alpha t$$

else, we have

$$w_i(\text{new}) = w_i(\text{old})$$
$$b(\text{new}) = b(\text{old})$$

Step 6: Train the network until there is no weight change. This is the stopping condition for the network. If this condition is not met, then start again from Step 2.

The algorithm discussed above is not sensitive to the initial values of the weights or the value of the learning rate.

3.2.6 Perceptron Training Algorithm for Multiple Output Classes

For multiple output classes, the perceptron training algorithm is as follows:

Step 0: Initialize the weights, biases and learning rate suitably.

Step 1: Check for stopping condition; if it is false, perform Steps 2–6.

Step 2: Perform Steps 3–5 for each bipolar or binary training vector pair $s:t$.

Step 3: Set activation (identity) of each input unit $i = 1$ to n:

$$x_i = s_i$$

Step 4: Calculate output response of each output unit $j = 1$ to m: First, the net input is calculated as

$$y_{inj} = b_j + \sum_{i=1}^{n} x_i w_{ij}$$

each input pattern response [illegible] (continues)

Then activations are applied over the net input to calculate the output response:

$$y_j = f(y_{inj}) = \begin{cases} 1 & \text{if } y_{inj} > \theta \\ 0 & \text{if } -\theta \leq y_{inj} \leq \theta \\ -1 & \text{if } y_{inj} < -\theta \end{cases}$$

Step 5: Make adjustment in weights and bias for $j = 1$ to m and $i = 1$ to n.

If $t_j \neq y_j$, then

$$w_{ij}(\text{new}) = w_{ij}(\text{old}) + \alpha t_j x_i$$
$$b_j(\text{new}) = b_j(\text{old}) + \alpha t_j$$

else, we have

$$w_{ij}(\text{new}) = w_{ij}(\text{old})$$
$$b_j(\text{new}) = b_j(\text{old})$$

Step 6: Test for the stopping condition, i.e., if there is no change in weights then stop the training process, else start again from Step 2.

It can be noticed that after training, the net classifies each of the training vectors. The above algorithm is suited for the architecture shown in Figure 3-4.

3.2.7 Perceptron Network Testing Algorithm

It is best to test the network performance once the training process is complete. For efficient performance of the network, it should be trained with more data. The testing algorithm (application procedure) is as follows:

Step 0: The initial weights to be used here are taken from the training algorithms (the final weights obtained during training).

Step 1: For each input vector X to be classified, perform Steps 2–3.

Step 2: Set activations of the input unit.

50

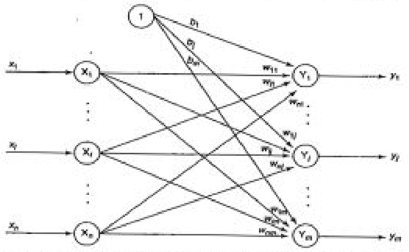


Figure 3-4 Network architecture for perceptron network for several output classes.

Step 3: Obtain the response of output unit.

$$y_{in} = \sum_{i=1}^{n} x_i w_i$$

$$y = f(y_{in}) = \begin{cases} 1 & \text{if } y_{in} > \theta \\ 0 & \text{if } -\theta \leq y_{in} \leq \theta \\ -1 & \text{if } y_{in} < -\theta \end{cases}$$

Thus, the testing algorithm tests the performance of network.

Note: In the case of perceptron network, it can be used for linear separability concept. Here the separating line may be based on the value of threshold, i.e., the threshold used in activation function must be a non-negative value.

The condition for separating the response from region of positive to region of zero is

$$w_1x_1 + w_2x_2 + b > \theta$$

The condition for separating the response from region of zero to region of negative is

$$w_1x_1 + w_2x_2 + b < -\theta$$

The conditions above are stated for a single-layer perceptron network with two input neurons and one output neuron and one bias.

3.3 Adaptive Linear Neuron (Adaline)

3.3.1 Theory

The units with linear activation function are called linear units. A network with a single linear unit is called an *Adaline* (adaptive linear neuron). That is, in an Adaline, the input–output relationship is linear. Adaline uses bipolar activation for its input signals and its target output. The weights between the input and the output are adjustable. The bias in Adaline acts like an adjustable weight, whose connection is from a unit with activations being always 1. Adaline is a net which has only one output unit. The Adaline network may be trained using delta rule. The delta rule may also be called as *least mean square* (LMS) *rule* or Widrow–Hoff rule. This learning rule is found to minimize the mean-squared error between the activation and the target value.

3.3.2 Delta Rule for Single Output Unit

The Widrow-Hoff rule is very similar to perceptron learning rule. However, their origins are different. The perceptron learning rule originates from the Hebbian assumption while the delta rule is derived from the gradient-descent method (it can be generalized to more than one layer). Also, the perceptron learning rule stops after a finite number of learning steps, but the gradient-descent approach continues forever, converging only asymptotically to the solution. The delta rule updates the weights between the connections so as to minimize the difference between the net input to the output unit and the target value. The major aim is to minimize the error over all training patterns. This is done by reducing the error for each pattern, one at a time.

The delta rule for adjusting the weight of ith pattern ($i = 1$ to n) is

$$\Delta w_i = \alpha(t - y_{in})x_i$$

where Δw_i is the weight change; α the learning rate; x the vector of activation of input unit; y_{in} the net input to output unit, i.e., $Y = \sum_{i=1}^{n} x_i w_i$; t the target output. The delta rule in case of several output units for adjusting the weight from ith input unit to the jth output unit (for each pattern) is

$$\Delta w_{ij} = \alpha(t_j - y_{inj})x_i$$

3.3.3 Architecture

As already stated, Adaline is a single-unit neuron, which receives input from several units and also from one unit called bias. An Adaline model is shown in Figure 3-5. The basic Adaline model consists of trainable weights. Inputs are either of the two values (+1 or −1) and the weights have signs (positive or negative). Initially, random weights are assigned. The net input calculated is applied to a quantizer transfer function (possibly activation function) that restores the output to +1 or −1. The Adaline model compares the actual output with the target output and on the basis of the training algorithm, the weights are adjusted.

3.3.4 Flowchart for Training Process

The flowchart for the training process is shown in Figure 3-6. This gives a pictorial representation of the network training. The conditions necessary for weight adjustments have to be checked carefully. The weights and other required parameters are initialized. Then the net input is calculated, output is obtained and compared with the desired output for calculation of error. On the basis of the error factor, weights are adjusted.

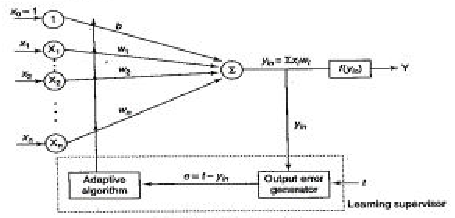


Figure 3-5 Adaline model.

3.3.5 Training Algorithm

The Adaline network training algorithm is as follows:

Step 0: Weights and bias are set to some random values but not zero. Set the learning rate parameter α.

Step 1: Perform Steps 2–6 when stopping condition is false.

Step 2: Perform Steps 3–5 for each bipolar training pair $s:t$.

Step 3: Set activations for input units $i = 1$ to n.

$$x_i = s_i$$

Step 4: Calculate the net input to the output unit.

$$y_{in} = b + \sum_{i=1}^{n} x_i w_i$$

Step 5: Update the weights and bias for $i = 1$ to n:

$$w_i(\text{new}) = w_i(\text{old}) + \alpha\,(t - y_{in})\,x_i$$
$$b(\text{new}) = b\,(\text{old}) + \alpha\,(t - y_{in})$$

Step 6: If the highest weight change that occurred during training is smaller than a specified tolerance then stop the training process, else continue. This is the test for stopping condition of a network.

The range of learning rate can be between 0.1 and 1.0.

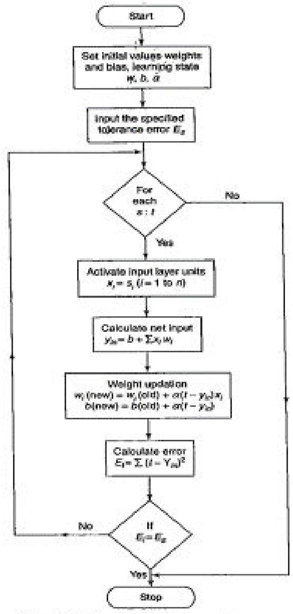

Start
Set initial values weights and bias, learning state w, b, α
Input the specified tolerance error E_s
For each $s : t$
No
Yes
Activate input layer units $x_i = s_i$ $(i = 1$ to $n)$
Calculate net input $y_{in} = b + \Sigma x_i w_i$
Weight updation $w_i(\text{new}) = w_i(\text{old}) + \alpha(t - y_{in})x_i$ $b(\text{new}) = b(\text{old}) + \alpha(t - y_{in})$
Calculate error $E_i = \Sigma\,(t - Y_{in})^2$
No
If $E_i = E_s$
Yes
Stop

Figure 3-6 Flowchart for Adaline training process.

Step 5: Calculate output of each hidden unit:

$$z_j = f(z_{inj})$$

Step 6: Find the output of the net:

$$y_{in} = b_0 + \sum_{j=1}^{m} z_j v_j$$
$$y = f(y_{in})$$

Step 7: Calculate the error and update the weights.

1. If $t = y$, no weight updation is required.
2. If $t \neq y$ and $t = +1$, update weights on z_j, where net input is closest to 0 (zero):

$$b_j(\text{new}) = b_j(\text{old}) + \alpha\,(1 - z_{inj})$$
$$w_{ij}(\text{new}) = w_{ij}(\text{old}) + \alpha\,(1 - z_{inj})x_i$$

3. If $t \neq y$ and $t = -1$, update weights on units z_k whose net input is positive:

$$w_{ik}(\text{new}) = w_{ik}(\text{old}) + \alpha\,(-1 - z_{ink})\,x_i$$
$$b_k(\text{new}) = b_k(\text{old}) + \alpha\,(-1 - z_{ink})$$

Step 8: Test for the stopping condition. (If there is no weight change or weight reaches a satisfactory level, or if a specified maximum number of iterations of weight updation have been performed then stop, or else continue).

Madalines can be formed with the weights on the output unit set to perform some logic functions. If there are only two hidden units present, or if there are more than two hidden units, then the "majority vote rule" function may be used.

3.5 Back-Propagation Network

3.5.1 Theory

The back-propagation learning algorithm is one of the most important developments in neural networks (Bryson and Ho, 1969; Werbos, 1974; Lecun, 1985; Parker, 1985; Rumelhart, 1986). This network has reawakened the scientific and engineering community to the modeling and processing of numerous quantitative phenomena using neural networks. This learning algorithm is applied to multilayer feed-forward networks consisting of processing elements with continuous differentiable activation functions. The networks associated with back-propagation learning algorithm are also called *back-propagation networks* (BPNs). For a given set of training input-output pair, this algorithm provides a procedure for changing the weights in a BPN to classify the given input patterns correctly. The basic concept for this weight update algorithm is simply the gradient-descent method as used in the case of simple perceptron networks with differentiable units. This is a method where the error is propagated back to the hidden unit. The aim of the neural network is to train the net to achieve a balance between the net's ability to respond (memorization) and its ability to give reasonable responses to the input that is similar but not identical to the one that is used in training (generalization).

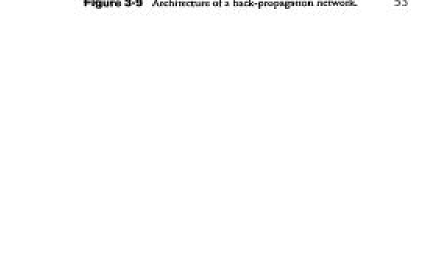
The back-propagation algorithm is different from other networks in respect to the process by which the weights are calculated during the learning period of the network. The general difficulty with the multilayer perceptrons is calculating the weights of the hidden layers in an efficient way that would result in a very small or zero output error. When the hidden layers are increased the network training becomes more complex. To update weights, the error must be calculated. The error, which is the difference between the actual (calculated) and the desired (target) output, is easily measured at the output layer. It should be noted that at the hidden layers, there is no direct information of the error. Therefore, other techniques should be used to calculate an error at the hidden layer, which will cause minimization of the output error, and this is the ultimate goal.

The training of the BPN is done in three stages – the feed-forward of the input training pattern, the calculation and back-propagation of the error, and updation of weights. The testing of the BPN involves the computation of feed-forward phase only. There can be more than one hidden layer (more beneficial) but one hidden layer is sufficient. Even though the training is very slow, once the network is trained it can produce its outputs very rapidly.

3.5.2 Architecture

A back-propagation neural network is a multilayer, feed-forward neural network consisting of an input layer, a hidden layer and an output layer. The neurons present in the hidden and output layers have biases, which are the connections from the units whose activation is always 1. The bias terms also acts as weights. Figure 3-9 shows the architecture of a BPN, depicting only the direction of information flow for the feed-forward phase. During the back-propagation phase of learning, signals are sent in the reverse direction.

The inputs are sent to the BPN and the output obtained from the net could be either binary (0, 1) or bipolar (−1, +1). The activation function could be any function which increases monotonically and is also differentiable.



→ feedforward phase

Figure 3-9 Architecture of a back-propagation network.

3.5.3 Flowchart for Training Process

The flowchart for the training process using a BPN is shown in Figure 3-10. The terminologies used in the flowchart and in the training algorithm are as follows:

x = input training vector $(x_1, \ldots, x_i, \ldots, x_n)$
t = target output vector $(t_1, \ldots, t_k, \ldots, t_m)$
α = learning rate parameter
x_i = input unit i. (Since the input layer uses identity activation function, the input and output signals here are same.)
v_{0j} = bias on jth hidden unit
w_{0k} = bias on kth output unit
z_j = hidden unit j. The net input to z_j is

$$z_{inj} = v_{0j} + \sum_{i=1}^{n} x_i v_{ij}$$

and the output is

$$z_j = f(z_{inj})$$

y_k = output unit k. The net input to y_k is

$$y_{ink} = w_{0k} + \sum_{j=1}^{p} z_j w_{jk}$$

and the output is

$$y_k = f(y_{ink})$$

δ_k = error correction weight adjustment for w_{jk} that is due to an error at output unit y_k, which is back-propagated to the hidden units that feed into unit y_k
δ_j = error correction weight adjustment for v_{ij} that is due to the back-propagation of error to the hidden unit z_j. [to input unit that feed i/p z_j]

Also, it should be noted that the commonly used activation functions are binary sigmoidal and bipolar sigmoidal activation functions (discussed in Section 2.3.3). These functions are used in the BPN because of the following characteristics: (i) continuity; (ii) differentiability; (iii) nondecreasing monotony.

The range of binary sigmoid is from 0 to 1, and for bipolar sigmoid it is from −1 to +1.

3.5.4 Training Algorithm

The error back-propagation learning algorithm can be outlined in the following algorithm:

Step 0: Initialize weights and learning rate (take some small random values).
Step 1: Perform Steps 2–9 when stopping condition is false.
Step 2: Perform Steps 3–8 for each training pair.

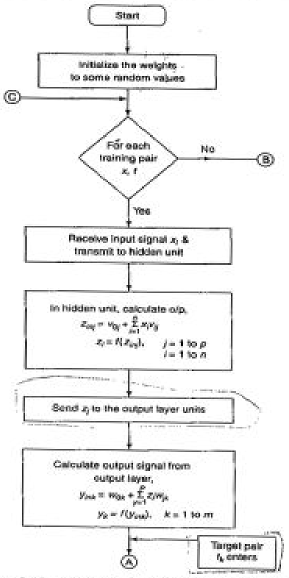

Start
Initialize the weights to some random values
C
For each training pair x, t
No
B
Yes
Receive input signal x_i & transmit to hidden unit
In hidden unit, calculate o/p,
$z_{inj} = v_{0j} + \sum_{i=1}^{n} x_i v_{ij}$
$z_j = f(z_{inj})$, $j = 1$ to p
$i = 1$ to n
Send z_j to the output layer units
Calculate output signal from output layer,
$y_{ink} = w_{0k} + \sum_{j=1}^{p} z_j w_{jk}$
$y_k = f(y_{ink})$, $k = 1$ to m
Target pair t_k enters
A

Figure 3-10 Flowchart for back-propagation network training.

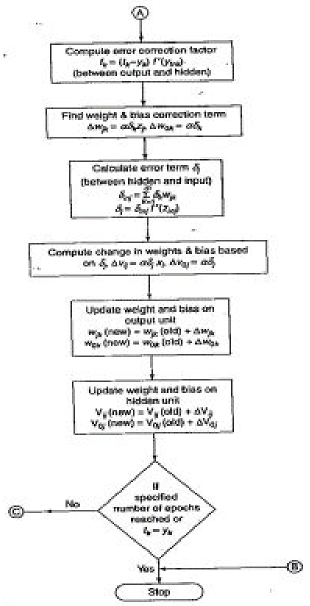

A
Compute error correction factor
$\delta_k = (t_k - y_k) f'(y_{ink})$
(between output and hidden)
Find weight & bias correction term
$\Delta w_{jk} = \alpha \delta_k z_j$, $\Delta w_{0k} = \alpha \delta_k$
Calculate error term δ_j
(between hidden and input)
$\delta_{inj} = \sum_{k=1}^{m} \delta_k w_{jk}$
$\delta_j = \delta_{inj} f'(z_{inj})$
Compute change in weights & bias based on δ_j, $\Delta v_{ij} = \alpha \delta_j x_i$, $\Delta v_{0j} = \alpha \delta_j$
Update weight and bias on output unit
w_{jk} (new) = w_{jk} (old) + Δw_{jk}
w_{0k} (new) = w_{0k} (old) + Δw_{0k}
Update weight and bias on hidden unit
V_{ij} (new) = V_{ij} (old) + ΔV_{ij}
V_{0j} (new) = V_{0j} (old) + ΔV_{0j}
If specified number of epochs reached or $t_k = y_k$
No
C
Yes
B
Stop

Figure 3-10 (*Continued*).

Feed-forward phase (Phase I)

Step 3: Each input unit receives input signal x_i and sends it to the hidden unit ($i = 1$ to n).

Step 4: Each hidden unit z_j($j = 1$ to p) sums its weighted input signals to calculate net input:

$$z_{inj} = v_{0j} + \sum_{i=1}^{n} x_i v_{ij}$$

Calculate output of the hidden unit by applying its activation functions over z_{inj} (binary or bipolar sigmoidal activation function):

$$z_j = f(z_{inj})$$

and send the output signal from the hidden unit to the input of output layer units.

Step 5: For each output unit y_k ($k = 1$ to m), calculate the net input:

$$y_{ink} = w_{0k} + \sum_{j=1}^{p} z_j w_{jk}$$

and apply the activation function to compute output signal

$$y_k = f(y_{ink})$$

Back-propagation of error (Phase II)

Step 6: Each output unit y_k($k = 1$ to m) receives a target pattern corresponding to the input training pattern and computes the error correction term:

$$\delta_k = (t_k - y_k) f'(y_{ink})$$

The derivative $f'(y_{ink})$ can be calculated as in Section 2.3.3. On the basis of the calculated error correction term, update the change in weights and bias:

$$\Delta w_{jk} = \alpha \delta_k z_j; \quad \Delta w_{0k} = \alpha \delta_k$$

Also, send δ_k to the hidden layer backwards.

Step 7: Each hidden unit (z_j, $j = 1$ to p) sums its delta inputs from the output units:

$$\delta_{inj} = \sum_{k=1}^{m} \delta_k w_{jk}$$

The term δ_{inj} gets multiplied with the derivative of $f(z_{inj})$ to calculate the error term:

$$\boxed{\delta_j = \delta_{inj} f'(z_{inj})}$$

The derivative $f'(z_{inj})$ can be calculated as discussed in Section 2.3.3 depending on whether binary or bipolar sigmoidal function is used. On the basis of the calculated δ_j, update the change in weights and bias:

$$\Delta v_{ij} = \alpha \delta_j x_i; \quad \Delta v_{0j} = \alpha \delta_j$$

Weight and bias updation (Phase III):

Step 8: Each output unit (y_k, $k = 1$ to m) updates the bias and weights:

$$w_{jk}(\text{new}) = w_{jk}(\text{old}) + \Delta w_{jk}$$
$$w_{0k}(\text{new}) = w_{0k}(\text{old}) + \Delta w_{0k}$$

Each hidden unit (z_j, $j = 1$ to p) updates its bias and weights:

$$v_{ij}(\text{new}) = v_{ij}(\text{old}) + \Delta v_{ij}$$
$$v_{0j}(\text{new}) = v_{0j}(\text{old}) + \Delta v_{0j}$$

Step 9: Check for the stopping condition. The stopping condition may be certain number of epochs reached or when the actual output equals the target output.

The above algorithm uses the incremental approach for updation of weights, i.e., the weights are being changed immediately after a training pattern is presented. There is another way of training called *batch-mode training*, where the weights are changed only after all the training patterns are presented. The effectiveness of two approaches depends on the problem, but batch-mode training requires additional local storage for each connection to maintain the immediate weight changes. When a BPN is used as a classifier, it is equivalent to the optimal Bayesian discriminant function for asymptotically large sets of statistically independent training patterns.

The problem in this case is whether the back-propagation learning algorithm can always converge and find proper weights for network even after enough learning. It will converge since it implements a gradient-descent on the error surface in the weight space, and this will roll down the error surface to the nearest minimum error and will stop. This becomes true only when the relation existing between the input and the output training patterns is deterministic and the error surface is deterministic. This is not the case in real world because the produced square-error surfaces are always at random. This is the stochastic nature of the back-propagation algorithm, which is purely based on the stochastic gradient-descent method. The BPN is a special case of stochastic approximation.

If the BPN algorithm converges at all, then it may get stuck with local minima and may be unable to find satisfactory solutions. The randomness of the algorithm helps it to get out of local minima. The error functions may have large number of global minima because of permutations of weights that keep the network input–output function unchanged. This causes the error surfaces to have numerous troughs.

3.5.5 Learning Factors of Back-Propagation Network

The training of a BPN is based on the choice of various parameters. Also, the convergence of the BPN is based on some important learning factors such as the initial weights, the learning rate, the updation rule, the size and nature of the training set, and the architecture (number of layers and number of neurons per layer).

3.5.5.1 *Initial Weights*

The ultimate solution may be affected by the initial weights of a multilayer feed-forward network. They are initialized at small random values. The choice of the initial weight determines how fast the network converges. The initial weights cannot be very high because the sigmoidal activation functions used here may get saturated

from the beginning itself and the system may be stuck at a local minima or at a very flat plateau at the starting point itself. One method of choosing the weight w_{ij} is choosing it in the range

$$w_{ij} = \left[\frac{-3}{\sqrt{o_i}}, \frac{3}{\sqrt{o_i}}\right]$$

where o_i is the number of processing elements j that feed-forward to processing element i. The initialization can also be done by a method called Nyugen–Widrow initialization. This type of initialization leads to faster convergence of network. The concept here is based on the geometric analysis of the response of hidden neurons to a single input. The method is used for improving the learning ability of the hidden units. The random initialization of weights connecting input neurons to the hidden neurons is obtained by the equation

$$v_{ij}(\text{new}) = \gamma \frac{v_{ij}(\text{old})}{\left\|\bar{v}_j(\text{old})\right\|}$$

where $\bar{v}_j$ is the average weight calculated for all values of i, and the scale factor $\gamma = 0.7(P)^{1/n}$ ("n" is the number of input neurons and "P" is the number of hidden neurons).

3.5.5.2 *Learning Rate α*

The learning rate (α) affects the convergence of the BPN. A larger value of α may speed up the convergence but might result in overshooting, while a smaller value of α has vice-versa effect. The range of α from 10^{-3} to 10 has been used successfully for several back-propagation algorithmic experiments. Thus, a large learning rate leads to rapid learning but there is oscillation of weights, while the lower learning rate leads to slower learning.

3.5.5.3 *Momentum Factor*

The gradient descent is very slow if the learning rate α is small and oscillates widely if α is too large. One very efficient and commonly used method that allows a larger learning rate without oscillations is by adding a momentum factor to the normal gradient-descent method.

The momentum factor is denoted by $\eta \in [0, 1]$ and the value of 0.9 is often used for the momentum factor. Also, this approach is more useful when some training data are very different from the majority of data. A momentum factor can be used with either pattern by pattern updating or batch-mode updating. In case of batch mode, it has the effect of complete averaging over the patterns. Even though the averaging is only partial in the pattern-by-pattern mode, it leaves some useful information for weight updation.

The weight updation formulas used here are

$$w_{jk}(t+1) = w_{jk}(t) + \underbrace{\alpha\delta_k z_j + \eta\left[w_{jk}(t) - w_{jk}(t-1)\right]}_{\Delta w_{jk}(t+1)}$$

and

$$v_{ij}(t+1) = v_{ij}(t) + \underbrace{\alpha\delta_j x_i + \eta\left[v_{ij}(t) - v_{ij}(t-1)\right]}_{\Delta v_{ij}(t+1)}$$

The momentum factor also helps in faster convergence.

3.5.5.4 *Generalization*

The best network for generalization is BPN. A network is said to be generalized when it sensibly interpolates with input networks that are new to the network. When there are many trainable parameters for the given amount of training data, the network learns well but does not generalize well. This is usually called *overfitting* or *overtraining*. One solution to this problem is to monitor the error on the test set and terminate the training when the error increases. With small number of trainable parameters, the network fails to learn the training data and performs very poorly on the test data. For improving the ability of the network to generalize from a training data set to a test data set, it is desirable to make small changes in the input space of a pattern, without changing the output components. This is achieved by introducing variations in the input space of training patterns as part of the training set. However, computationally, this method is very expensive. Also, a net with large number of nodes is capable of membrizing the training set at the cost of generalization. As a result, smaller nets are preferred than larger ones.

3.5.5.5 *Number of Training Data*

The training data should be sufficient and proper. There exists a rule of thumb, which states that the training data should cover the entire expected input space, and while training, training-vector pairs should be selected randomly from the set. Assume that the input space as being linearly separable into "L" disjoint regions with their boundaries being part of hyper planes. Let "T" be the lower bound on the number of training patterns. Then, choosing T such that $T/L \gg 1$ will allow the network to discriminate pattern classes using fine piecewise hyperplane partitioning. Also in some cases, scaling or normalization has to be done to help learning.

3.5.5.6 *Number of Hidden Layer Nodes*

If there exists more than one hidden layer in a BPN, then the calculations performed for a single layer are repeated for all the layers and are summed up at the end. In case of all multilayer feed-forward networks, the size of a hidden layer is very important. The number of hidden units required for an application needs to be determined separately. The size of a hidden layer is usually determined experimentally. For a network of a reasonable size, the size of hidden nodes has to be only a relatively small fraction of the input layer. For example, if the network does not converge to a solution, it may need more hidden nodes. On the other hand, if the network converges, the user may try a very few hidden nodes and then settle finally on a size based on overall system performance.

3.5.6 Testing Algorithm of Back-Propagation Network

The testing procedure of the BPN is as follows:

Step 0: Initialize the weights. The weights are taken from the training algorithm.

Step 1: Perform Steps 2–4 for each input vector.

Step 2: Set the activation of input unit for x_i ($i = 1$ to n).

Step 3: Calculate the net input to hidden unit x and its output. For $j = 1$ to p,

$$z_{inj} = v_{0j} + \sum_{i=1}^{n} x_i v_{ij}$$
$$z_j = f(z_{inj})$$

Step 4: Now compute the output of the output layer unit. For $k = 1$ to m,

$$y_{ink} = w_{0k} + \sum_{j=1}^{p} z_j w_{jk}$$
$$y_k = f(y_{ink})$$

Use sigmoidal activation functions for calculating the output.

3.6 Radial Basis Function Network

3.6.1 Theory

The radial basis function (RBF) is a classification and functional approximation neural network developed by M.J.D. Powell. The network uses the most common nonlinearities such as sigmoidal and Gaussian kernel functions. The Gaussian functions are also used in regularization networks. The response of such a function is positive for all values of y; the response decreases to 0 as $|y| \to 0$. The Gaussian function is generally defined as

$$f(y) = e^{-y^2}$$

The derivative of this function is given by

$$f'(y) = -2y\,e^{-y^2} = -2y\,f(y)$$

The graphical representation of this Gaussian function is shown in Figure 3-11 below.

When the Gaussian potential functions are being used, each node is found to produce an identical output for inputs existing within the fixed radial distance from the center of the kernel, they are found to be radically symmetric, and hence the name radial basis function network. The entire network forms a linear combination of the nonlinear basis function.

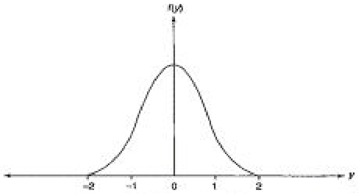
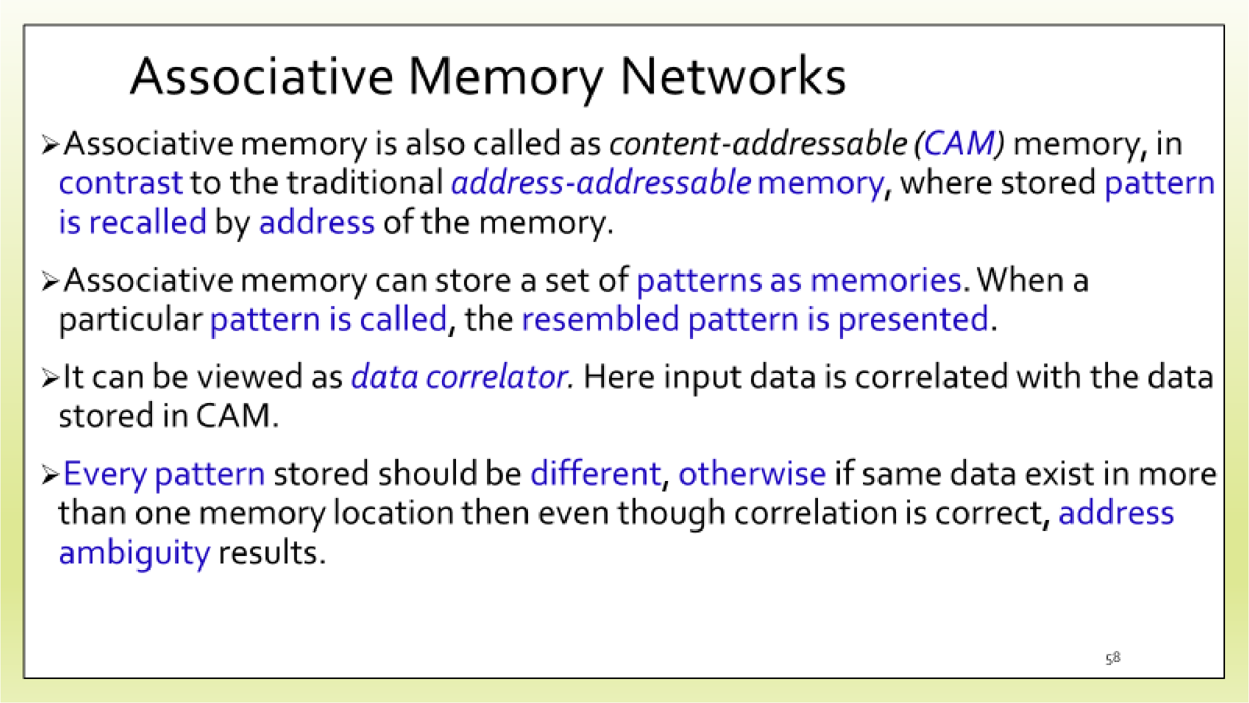


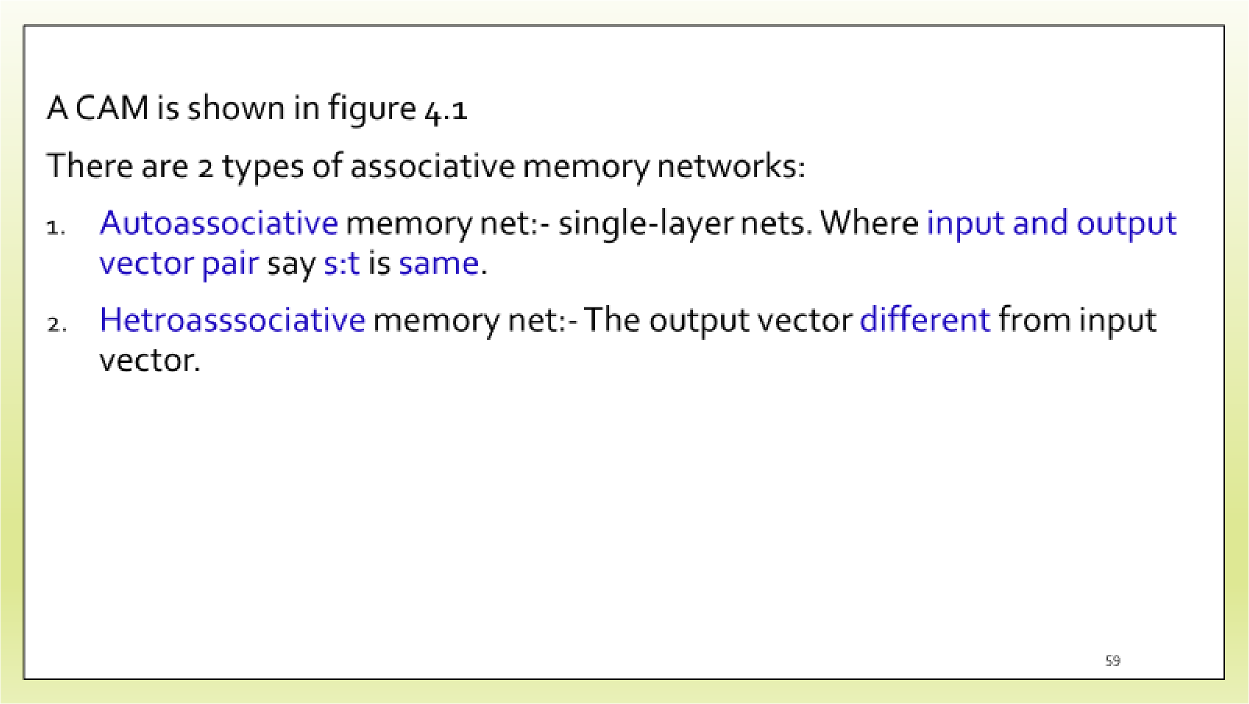
Figure 3-11 Gaussian kernel function.

Associative Memory Networks   CO2



Associative Memory Networks

- ➢Associative memory is also called as *content-addressable (CAM)* memory, in contrast to the traditional *address-addressable* memory, where stored pattern is recalled by address of the memory.
- ➢Associative memory can store a set of patterns as memories. When a particular pattern is called, the resembled pattern is presented.
- ➢It can be viewed as *data correlator.* Here input data is correlated with the data stored in CAM.
- ➢Every pattern stored should be different, otherwise if same data exist in more than one memory location then even though correlation is correct, address ambiguity results.



A CAM is shown in figure 4.1

There are 2 types of associative memory networks:

1. Autoassociative memory net:- single-layer nets. Where input and output vector pair say s:t is same.
2. Hetroasssociative memory net:- The output vector different from input vector.

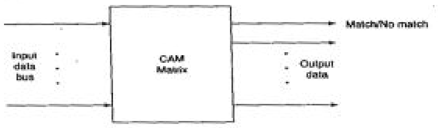


Figure 4-1 CAM architecture.

be autoassociative memory net. On the other hand, if the output vectors are different from the input vectors then the net is said to be heteroassociative memory net.

If there exist vectors, say, $x = (x_1, x_2, \ldots, x_n)^T$ and $x' = (x_1', x_2', \ldots, x_n')^T$, then the hamming distance (HD) is defined as the number of mismatched components of x and x' vectors, i.e.,

$$\text{HD}\,(x, x') = \begin{cases} \sum_{i=1}^{n} |x_i - x_i'| \text{ if } x_i, x_i' \in [0, 1] \\ \frac{1}{2} \sum_{i=1}^{n} |x_i - x_i'| \text{ if } x_i, x_i' \in [-1, 1] \end{cases}$$

The architecture of an associative net may be either feed-forward or iterative (recurrent). As is already known, in a feed-forward net the information flows from the input units to the output units; on the other hand, in a recurrent neural net, there are connections among the units to form a closed-loop structure. In the forthcoming sections, we will discuss the training algorithms used for pattern association and various types of association nets in detail.

4.2 Training Algorithms for Pattern Association

There are two algorithms developed for training of pattern association nets. These are discussed below.

4.2.1 Hebb Rule

The Hebb rule is widely used for finding the weights of an associative memory neural net. The training vector pairs here are denoted as $s{:}t$. The flowchart for the training algorithm of pattern association is as shown in Figure 4-2. The weights are updated until there is no weight change. The algorithmic steps followed are given below:

Step 0: Set all the initial weights to zero, i.e.,

$$w_{ij} = 0 \quad (i = 1 \text{ to } n, j = 1 \text{ to } m)$$

Step 1: For each training target input output vector pairs $s{:}t$, perform Steps 2–4.

Step 2: Activate the input layer units to current training input,

$$x_i = s_i \quad (\text{for } i = 1 \text{ to } n)$$

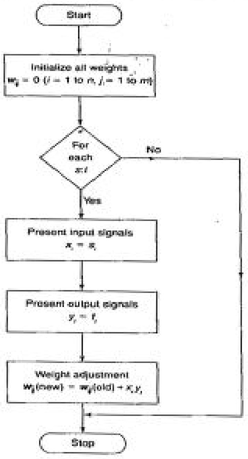


Figure 4-2 Flowchart for Hebb rule.

Step 3: Activate the output layer units to current target output,

$$y_j = t_j \quad (\text{for } j = 1 \text{ to } m)$$

Step 4: Start the weight adjustment

$$w_{ij}(\text{new}) = w_{ij}(\text{old}) + x_i y_j \quad (\text{for } i = 1 \text{ to } n, j = 1 \text{ to } m)$$

This algorithm is used for the calculation of the weights of the associative nets. Also, it can be u patterns that are being represented as either binary or bipolar vectors.

60

4.2.2 Outer Products Rule

Outer products rule is an alternative method for finding weights of an associative net. This is depicted as follows:

$$\text{Input} \Rightarrow s = (s_1, \ldots, s_i, \ldots, s_n)$$
$$\text{Output} \Rightarrow t = (t_1, \ldots, t_j, \ldots, t_m)$$

The outer product of the two vectors is the product of the matrices $S = s^T$ and $T = t$, i.e., between $[n \times 1]$ matrix and $[1 \times m]$ matrix. The transpose is to be taken for the input matrix given.

The matrix multiplication is done as follows:

$$ST = s^T t$$

$$= \begin{bmatrix} s_1 \\ \vdots \\ s_i \\ \vdots \\ s_n \end{bmatrix}_{n \times 1} [t_1 \ldots t_j \ldots t_m]_{1 \times m}$$

$$W = \begin{bmatrix} s_1 t_1 & \ldots & s_1 t_j & \ldots & s_1 t_m \\ \vdots & & & & \vdots \\ s_i t_1 & \ldots & s_i t_j & \ldots & s_i t_m \\ \vdots & & & & \vdots \\ s_n t_1 & \ldots & s_n t_j & \ldots & s_n t_m \end{bmatrix}_{n \times m}$$

This weight matrix is same as the weight matrix obtained by Hebb rule to store the pattern association $s{:}t$. For storing a set of associations, $s(p){:}t(p)$, $p = 1$ to P, wherein,

$$s(p) = (s_1(p), \ldots, s_i(p), \ldots, s_n(p))$$
$$t(p) = (t_1(p), \ldots, t_j(p), \ldots, t_m(p))$$

the weight matrix $W = \{w_{ij}\}$ can be given as

$$\boxed{w_{ij} = \sum_{p=1}^{P} s_i^T(p)\, t_j(p)}$$

This can also be rewritten as

$$\boxed{W = \sum_{p=1}^{P} s^T(p)\, t(p)}$$

for finding the weights of the net using Hebbian learning. Similar to the Hebb rule, even the delta rule discussed in Chapter 2 can be used for storing weights of pattern association nets.

4.3 Autoassociative Memory Network

4.3.1 Theory

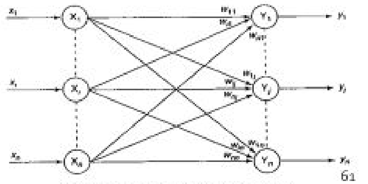
In the case of an autoassociative neural net, the training input and the target output vectors are the same. The determination of weights of the association net is called storing of vectors. This type of memory net needs suppression of the output noise at the memory output. The vectors that have been stored can be retrieved from distorted (noisy) input if the input is sufficiently similar to it. The net's performance is based on its ability to reproduce a stored pattern from a noisy input. It should be noted, that in the case of autoassociative net, the weights on the diagonal can be set to zero. This can be called as auto associative net with no self-connection. The main reason behind setting the weights to zero is that it improves the net's ability to generalize or increase the biological plausibility of the net. This may be more suited for iterative nets and when delta rule is being used.

4.3.2 Architecture

The architecture of an autoassociative neural net is shown in Figure 4-3. It shows that for an autoassociative net, the training input and target output vectors are the same. The input layer consists of n input units and the output layer also consists of n output units. The input and output layers are connected through weighted interconnections. The input and output vectors are perfectly correlated with each other component by component.

4.3.3 Flowchart for Training Process

The flowchart here is the same as discussed in Section 4.2.1, but it may be noted that the number of input units and output units are the same. The flowchart is shown in Figure 4-4.



61

Figure 4-3 Architecture of autoassociative net.

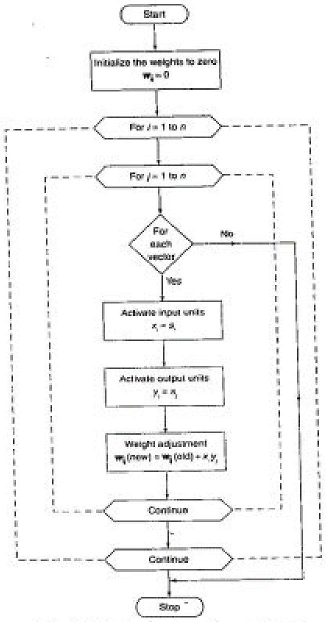


Figure 4-4 Flowchart for training of autoassociative net.

4.3.4 Training Algorithm

The training algorithm discussed here is similar to that discussed in Section 4.2.1 but there are same numbers of output units as that of the input units.

```
Step 0: Initialize all the weights to zero,
            $w_{ij} = 0\ (i = 1 \text{ to } n,\ j = 1 \text{ to } n)$
Step 1: For each of the vector that has to be stored perform Steps 2–4.
Step 2: Activate each of the input unit.
            $x_i = s_i\ (i = 1 \text{ to } n)$
Step 3: Activate each of the output unit.
            $y_j = s_j\ (j = 1 \text{ to } n)$
Step 4: Adjust the weights,
            $w_{ij}(\text{new}) = w_{ij}(\text{old}) + x_i y_j$
    The weights can also be determined by the formula
            $W = \sum_{p=1}^{P} s^T(p)s(p)$
```

→ the matched pattern [illegible] with the output

4.3.5 Testing Algorithm

An autoassociative memory neural network can be used to determine whether the given input vector is a "known" vector or an "unknown" vector. The net is said to recognize a "known" vector if the net produces a pattern of activation on the output units which is same as one of the vectors stored in it. The testing procedure of an autoassociative neural net is as follows:

```
Step 0: Set the weights obtained for Hebb's rule or outer products.
Step 1: For each of the testing input vector presented perform Steps 2–4.
Step 2: Set the activations of the input units equal to that of input vector.
Step 3: Calculate the net input to each output unit $j = 1$ to $n$:
            $y_{inj} = \sum_{i=1}^{n} x_i w_{ij}$
Step 4: Calculate the output by applying the activation over the net input:
            $y_j = f(y_{inj}) = \begin{cases} +1 & \text{if } y_{inj} > 0 \\ -1 & \text{if } y_{inj} \le 0 \end{cases}$
```

62

This type of network can be used in speech processing, image processing, pattern classification, etc.

4.4 Heteroassociative Memory Network

4.4.1 Theory

In case of a heteroassociative neural net, the training input and the target output vectors are different. The weights are determined in a way that the net can store a set of pattern associations. The association here is a pair of training input target output vector pairs $(s(p), t(p))$, with $p = 1, \ldots, P$. Each vector $s(p)$ has n components and each vector $t(p)$ has m components. The determination of weights is done either by using Hebb rule or delta rule. The net finds an appropriate output vector, which corresponds to an in~~put vector~~ x, that ~~may be eith~~er one of the stored patterns or a new pattern.

4.4.2 Architecture

The architecture of a heteroassociative net is shown in Figure 4-5. From the figure, it can be noticed that for a heteroassociative net, the training input and target output vectors are different. The input layer consists of n number of input units and the output layer consists of m number of output units. There exist weighted interconnections between the input and output layers. The input and output layer units are not correlated with each other. The flowchart of the training process and the ~~training algorithm~~ are the same as discussed in Section 4.2.1.

4.4.3 Testing Algorithm

The testing algorithm used for testing the heteroassociative net with either noisy input or with known input is as follows:

```
Step 0: Initialize the weights from the training algorithm.
Step 1: Perform Steps 2–4 for each input vector presented.
Step 2: Set the activation for input layer units equal to that of the current input vector given, $x_i$.
```

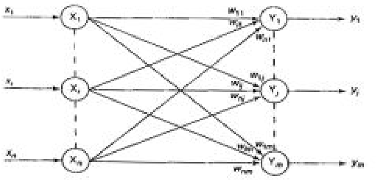


Figure 4-5 Architecture of heteroassociative net.

```
Step 3: Calculate the net input to the output units:
            $y_{inj} = \sum_{i=1}^{n} x_i w_{ij}\quad (j = 1 \text{ to } m)$
Step 4: Determine the activations of the output units over the calculated net input:
            $y_j = \begin{cases} 1 & \text{if } y_{inj} > 0 \\ 0 & \text{if } y_{inj} = 0 \\ -1 & \text{if } y_{inj} < 0 \end{cases}$
```

binary output

& i=1 to n

Thus, the output vector y obtained gives the pattern associated with the input vector x.

Note: Heteroassociative memory is not an iterative memory network. If the responses of the net are binary, a activation function to be used is

$$y_j = \begin{cases} 1 & \text{if } y_{inj} \ge 0 \\ 0 & \text{if } y_{inj} < 0 \end{cases}$$

→ feed back [illegible]

4.5 Bidirectional Associative Memory (BAM)

4.5.1 Theory

The BAM was developed by Kosko in the year 1988. The BAM network performs forward and ba associative searches for stored stimulus responses. The BAM is a recurrent heteroassociative pattern-m network that encodes binary or bipolar patterns using Hebbian learning rule. It associates patterns, s set A to patterns from set B and vice versa is also performed. BAM neural nets can respond to inp either layers (input layer and output layer). There exist two types of BAM, called *discrete* and *continuou* These two types of BAM are discussed in the following sections.

4.5.2 Architecture

The architecture of BAM network is shown in Figure 4-6. It consists of two layers of neurons which a nected by directed weighted path interconnections. The network dynamics involve two layers of inte The BAM network iterates by sending the signals back and forth between the two layers until all the reach equilibrium. The weights associated with the network are bidirectional. Thus, BAM can resp the inputs in either layer. Figure 4-6 shows a single layer BAM network consisting of n units in X la m units in Y layer. The layers can be connected in both directions (bidirectional) with the result the matrix sent from the X layer to the Y layer is W and the weight matrix for signals sent from the Y laye X layer is W^T. Thus, the weight matrix is calculated in both directions.

4.5.3 Discrete Bidirectional Associative Memory

The structure of discrete BAM is same as shown in Figure 4-6. When the memory neurons are being a by putting an initial vector at the input of a layer, the network evolves a two-pattern stable state wi pattern at the output of one layer. Thus, the network involves two layers of interaction between eac

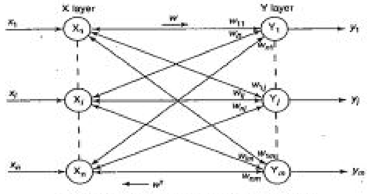


Figure 4-6 Bidirectional associative memory net.

The two bivalent forms of BAM are found to be related with each other, i.e., binary and bipolar. The weights in both the cases are found as the sum of ~~the outer~~ products of the bipolar form of the given training vector pairs. In case of BAM, a definite nonzero threshold is assigned. Thus, the activation function is a step function, with the defined nonzero threshold. When compared, to the binary vectors, bipolar vectors improve the performance of ~~the net to a large~~ extent.

4.5.3.1 Determination of Weights

Let the input vectors be denoted by $s(p)$ and target vectors by $t(p)$, $p = 1, \ldots, P$. Then the weight matrix to store a set of input and target vectors, where

$$s(p) = (s_1(p), \ldots, s_i(p), \ldots, s_n(p))$$
$$t(p) = (t_1(p), \ldots, t_j(p), \ldots, t_m(p))$$

can be determined by Hebb rule training algorithm discussed in Section 4.2.1. In case of input vectors being binary, the weight matrix $W = \{w_{ij}\}$ is given by

$$\boxed{w_{ij} = \sum_{p=1}^{P} [2s_i(p) - 1][2t_j(p) - 1]}$$

On the other hand, when the input vectors are bipolar, the weight matrix $W = \{w_{ij}\}$ can be defined as

$$w_{ij} = \sum_{p=1}^{P} s_i(p)t_j(p)$$

The weights matrix in both the cases is going to be in bipolar form neither the input vectors are in binary or not. The formulas mentioned above can be directly applied to the determination of weights of a BAM.

4.5.3.2 Activation Functions for BAM

The step activation function with a nonzero threshold is used as the activation function for discrete BAM network. The activation function is based on whether the input target vector pairs used are binary or bipolar. The activation function for the Y layer

1. with binary input vectors is

$$y_j = \begin{cases} 1 & \text{if } y_{inj} > 0 \\ y_j & \text{if } y_{inj} = 0 \\ 0 & \text{if } y_{inj} < 0 \end{cases}$$

2. with bipolar input vectors is

$$y_j = \begin{cases} 1 & \text{if } y_{inj} > \theta_j \\ y_j & \text{if } y_{inj} = \theta_j \\ -1 & \text{if } y_{inj} < \theta_j \end{cases}$$

The activation function for the X layer

1. with binary input vectors is

$$x_i = \begin{cases} 1 & \text{if } x_{ini} > 0 \\ x_i & \text{if } x_{ini} = 0 \\ 0 & \text{if } x_{ini} < 0 \end{cases}$$

2. with bipolar input vectors is

$$x_i = \begin{cases} 1 & \text{if } x_{ini} > \theta_i \\ x_i & \text{if } x_{ini} = \theta_i \\ -1 & \text{if } x_{ini} < \theta_i \end{cases}$$

It may be noted that if the threshold value is equal to that of the net input calculated, then the previous output value calculated is left as the activation of that unit. At a particular time instant, signals are sent only from one layer to the other and not in both the directions.

4.5.3.3 Testing Algorithm for Discrete BAM

The testing algorithm is used to test the noisy patterns entering into the network. Based on the training algorithm, weights are determined, by means of which net input is calculated for the given test pattern and activations is applied over it, to recognize the test patterns. The testing algorithm for the net is as follows:

Step 0: Initialize the weights to store p vectors. Also initialize all the activations to zero.

Step 1: Perform Steps 2–6 for each testing input.

Step 2: Set the activations of X layer to current input pattern, i.e., presenting the input pattern x to X layer and similarly presenting the input pattern y to Y layer. Even though, it is bidirectional memory, at one time step, signals can be sent from only one layer. So, either of the input patterns may be the zero vector.

Step 3: Perform Steps 4–6 when the activations are not converged.

Step 4: Update the activations of units in Y layer. Calculate the net input,

$$y_{inj} = \sum_{i=1}^{n} x_i w_{ij}$$

Applying the activations (as in Section 4.5.3.2), we obtain

$$y_j = f(y_{inj})$$

Send this signal to the X layer.

Step 5: Update the activations of units in X layer. Calculate the net input,

$$x_{ini} = \sum_{j=1}^{m} y_j w_{ij}$$

Apply the activations over the net input,

$$x_i = f(x_{ini})$$

Send this signal to the Y layer.

Step 6: Test for convergence of the net. The convergence occurs if the activation vectors x and y reach equilibrium. If this occurs then stop, otherwise, continue.

4.5.4 Continuous BAM

A continuous BAM transforms the input smoothly and continuously in the range 0–1 using logistic sigmoid functions as the activation functions for all units. The logistic sigmoidal ~~function may be either binary~~ sigmoidal function or bipolar ~~sigmoidal function~~. When a bipolar sigmoidal function with a high gain is chosen, then the continuous BAM might converge to a state of vectors which will approach vertices of the cube. When that state of the vector approaches it acts like a discrete BAM.

If the input vectors are binary, $(s(p), t(p))$, $p = 1$ to P, the weights are determined using the formula

[illegible]

$$w_{ij} = \sum_{p=1}^{P} [2s_i(p) - 1][2t_j(p) - 1]$$

i.e., even though the input vectors are binary, the weight matrix is bipolar. The activation function used here is the logistic sigmoidal function. If it is binary logistic function, then the activation function is

$$f(y_{inj}) = \frac{1}{1 + e^{-y_{inj}}}$$

If the activation function used is a bipolar logistic function, then the function is defined as

$$f(y_{inj}) = \frac{2}{1 + e^{-y_{inj}}} - 1 = \frac{1 - e^{-y_{inj}}}{1 + e^{-y_{inj}}}$$

These activation functions are applied over the net input to calculate the output. The net input can be calculated with a bias included, i.e.,

$$y_{inj} = b_j + \sum_{i=1}^{n} x_i w_{ij}$$

$x_{ini} = b_i + \sum_j y_i w_{ij}$

and all these formulas apply for the units in X layer also.

4.5.5 Analysis of Hamming Distance, Energy Function and Storage Capacity

The hamming distance is defined as the number of mismatched components of two given bipolar or binary vectors. It can also be defined as the number of different bits in two binary or bipolar vectors X and X'. It is denoted as $H[X, X']$. The average hamming distance between the vectors is $(1/n)H[X, X']$, where "n" is the number of components in each vector. Consider the vectors,

$$X = [1\ 0\ 1\ 0\ 1\ 1\ 0] \quad \text{and} \quad X' = [1\ 1\ 1\ 1\ 0\ 0\ 1]$$

The hamming distance between these two given vectors is equal to 5. The average hamming distance between the corresponding vectors is 5/7.

The stability analysis of a BAM is based on the definition of Lyapunov function (energy function). Consider that there are p vector association pairs to be stored in a BAM:

$$\{(x^1, y^1), (x^2, y^2), \ldots, (x^p, y^p)\}$$

where $x^k = (x_1^k, x_2^k, \ldots, x_m^k)^T$ and $y^k = (y_1^k, y_2^k, \ldots, y_n^k)^T$ are either binary or bipolar vectors. A Lyapunov function must be always bounded and decreasing. A BAM can be said to be bidirectionally stable if the state converges to a stable point, i.e., $y^k \to y^{k+1} \to y^{k+2}$ and $y^{k+2} = y^k$. This gives the minimum of the energy function. The energy function or Lynapunov function of a BAM is defined as

$$E_f(x, y) = \frac{-1}{2} x^T W^T y - \frac{1}{2} y^T Wx = -y^T Wx$$

The change in energy due to the single bit changes in both vectors y and x given as Δy_i and Δx_j can be found as

$$\Delta E_f(y_i) = \nabla_y E \Delta y_i = -Wx\Delta y_i = -\left(\sum_{j=1}^{m} x_j w_{ij}\right) \times \Delta y_i, \quad i = 1 \text{ to } n$$

$$\Delta E_f(x_j) = \nabla_x E \Delta x_j = -W^T y \Delta x_j = -\left(\sum_{i=1}^{n} y_i w_{ij}\right) \times \Delta x_j, \quad j = 1 \text{ to } m$$

where Δy_i and Δx_j are given as

$$\Delta x_j = \begin{cases} 2 & \text{if } \sum_{i=1}^{n} y_i w_{ij} > 0 \\ 0 & \text{if } \sum_{i=1}^{n} y_i w_{ij} = 0 \\ -2 & \text{if } \sum_{i=1}^{n} y_i w_{ij} < 0 \end{cases} \quad \text{and} \quad \Delta y_i = \begin{cases} 2 & \text{if } \sum_{j=1}^{m} x_j w_{ij} > 0 \\ 0 & \text{if } \sum_{j=1}^{m} x_j w_{ij} = 0 \\ -2 & \text{if } \sum_{j=1}^{m} x_j w_{ij} < 0 \end{cases}$$

65

Here the energy function is bounded below by

$$E_f(x,y) \geq -\sum_{i=1}^{n}\sum_{j=1}^{m}|w_{ij}|$$

so the discrete BAM will converge to a stable state.

The memory capacity or the storage capacity of BAM may be given as

$$\min(m,n)$$

where "n" is the number of units in X layer and "m" is the number of units in Y layer. Also a more conservative capacity is estimated as follows:

$$\sqrt{\min(m,n)}$$

4.6 Hopfield Networks

John J. Hopfield developed a model in the year 1982 conforming to the asynchronous nature of biological neurons. The networks proposed by Hopfield are known as Hopfield networks and it is his work that promoted construction of the first analog VLSI neural chip. This network has found many useful applications in associative memory and various optimization problems. In this section, two types of network are discussed: *discrete* and *continuous Hopfield networks*.

4.6.1 Discrete Hopfield Network

The Hopfield network is an autoassociative fully interconnected single-layer feedback network. It is also a symmetrically weighted network. When this is operated in discrete line fashion it is called as *discrete Hopfield network* and its architecture as a single-layer feedback network can be called as *recurrent*. The network takes two-valued inputs: binary (0, 1) or bipolar (+1, −1); the use of bipolar inputs makes the analysis easier. The network has symmetrical weights with no self-connections, i.e.,

$$w_{ij} = w_{ji};\quad w_{ii} = 0$$

The key points to be noted in Hopfield net are: only one unit updates its activation at a time; also each unit is found to continuously receive an external signal along with the signals it receives from the other units in the net. When a single-layer recurrent network is performing a sequential updating process, an input pattern is first applied to the network and the network's output is found to be initialized accordingly. Afterwards, the initializing pattern is removed, and the output that is initialized becomes the new updated input through the feedback connections. The first updated input forces the first updated output, which in turn acts as the second updated input through the feedback interconnections and results in second updated output. This transition process continues until no new, updated responses are produced and the network reaches its equilibrium.

The asynchronous updation of the units allows a function, called as energy functions or Lyapunov function, for the net. The existence of this function enables us to prove that the net will converge to a stable set of activations. The usefulness of content addressable memory is realized by the discrete Hopfield net.

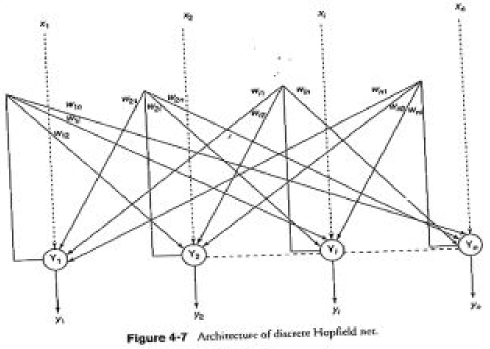


Figure 4-7 Architecture of discrete Hopfield net.

4.6.1.1 Architecture of Discrete Hopfield Net

The architecture of discrete Hopfield net is shown in Figure 4-7. The Hopfield's model consists of processing elements with two outputs, one inverting and the other non-inverting. The outputs from each processing element are fed back to the input of other processing elements but not to itself. The connections are found to be resistive and the connection strength over it is represented as w_{ij}. Here, as such there are no negative resistors, hence excitatory connections use positive outputs and inhibitory connections use inverted outputs. Connections are excitatory if the output of a processing element is found to be same as the input, and they are inhibitory if the inputs differ from the output of the processing element. A connection between the processing elements i and j is found to be associated with a connection strength w_{ij}. This weight is positive if units i and j are both on. On the other hand, if the connection strength is negative, it represents the situation of unit i being on and j being off. Also, the weights are symmetric, i.e., the weights w_{ij} are same as w_{ji}.

4.6.1.2 Training Algorithm of Discrete Hopfield Net

There exist several versions of the discrete Hopfield net. It should be noted that Hopfield's first description used binary input vectors and only later on bipolar input vectors used.

For storing a set of binary patterns $s(p)$, $p = 1$ to P, where $s(p) = (s_1(p), \ldots, s_i(p), \ldots, s_n(p))$, the weight matrix W is given as

$$w_{ij} = \sum_{p=1}^{P}[2s_i(p)-1][2s_j(p)-1],\quad \text{for } i \neq j$$ 66

For storing a set of bipolar input patterns, $s(p)$ (as defined above), the weight matrix W is given as

$$w_{ij} = \sum_{p=1}^{P} s_i(p)s_j(p),\quad \text{for } i \neq j$$

and the weights here have no self-connection, i.e., $w_{ij} = 0$.

4.6.1.3 Testing Algorithm of Discrete Hopfield Net

In the case of testing, the update rule is formed and the initial weights are those obtained from the training algorithm. The testing algorithm for the discrete Hopfield net is as follows:

Step 0: Initialize the weights to store patterns, i.e., weights obtained from training algorithm using Hebb rule.

Step 1: When the activations of the net are not converged, then perform Steps 2–8.

Step 2: Perform Steps 3–7 for each input vector X.

Step 3: Make the initial activations of the net equal to the external input vector X:

$$y_i = x_i\ (i = 1 \text{ to } n)$$

Step 4: Perform Steps 5–7 for each unit Y_i. (Here, the units are updated in random order.)

Step 5: Calculate the net input of the network:

$$y_{ini} = x_i + \sum_j y_j w_{ji}$$

Step 6: Apply the activations over the net input to calculate the output:

$$y_i = \begin{cases} 1 & \text{if } y_{ini} > \theta_i \\ y_i & \text{if } y_{ini} = \theta_i \\ 0 & \text{if } y_{ini} < \theta_i \end{cases}$$

where θ_i is the threshold and is normally taken as zero.

Step 7: Now feed back (transmit) the obtained output y_i to all other units. Thus, the activation vectors are updated.

Step 8: Finally, test the network for convergence.

The updation here is carried out at random, but it should be noted that each unit may be updated at the same average rate. The asynchronous fashion of updation is carried out here. This means that for a given time only a single neural unit is allowed to update its output. The next update can be carried out on a randomly chosen node which uses the already updated output. It can also be said that under asynchronous operation of the network, each output node unit is updated separately by taking into account the most recent values that have already been updated. This type of updation is referred to as an *asynchronous stochastic recursion* of the discrete Hopfield network. By performing the analysis of the Lyapunov function, i.e., the energy function for the Hopfield net, it can be shown that the main feature for the convergence of this net is the asynchronous updation of weights and the weights with no self-connection, i.e., the zeros exist on the diagonals of the weight matrix.

A Hopfield network with binary input vectors is used to determine whether an input vector is a "known" vector or an "unknown" vector. The net has the capacity to recognize a known vector by producing a pattern of activations on the units of the net that is same as the vector stored in the net. For example, if the input vector is an unknown vector, the activation vectors resulted during iteration will converge to an activation vector which is not one of the stored patterns; such a pattern is called as *spurious stable state*.

4.6.1.4 Analysis of Energy Function and Storage Capacity on Discrete Hopfield Net

An energy function generally is defined as a function that is bounded and is a nonincreasing function of the state of the system. The energy function, also called as Lyapunov function, determines the stability property of a discrete Hopfield network. The state of a system for a neural network is the vector of activations of the units. Hence, if it is possible to find an energy function for an iterative neural net, the net will converge to a stable set of activations. An energy function E_f of a discrete Hopfield network is characterized as

$$E_f = -\frac{1}{2}\sum_{i=1}^{n}\sum_{\substack{j=1\\ j\neq i}}^{n} y_i y_j w_{ij} - \sum_{i=1}^{n} x_i y_i + \sum_{i=1}^{n}\theta_i y_i$$

If the network is stable, then the above energy function decreases whenever the state of any node changes. Assuming that node i has changed its state from $y_i^{(k)}$ to $y_i^{(k+1)}$, i.e., the output has changed from +1 to −1 or from −1 to +1, the energy change ΔE_f is then given by

$$\begin{aligned}\Delta E_f &= E_f\left(y_i^{(k+1)}\right) - E_f\left(y_i^{(k)}\right)\\ &= -\left(\sum_{\substack{j=1\\ j\neq i}}^{n} y_j^{(k)} w_{ij} + x_i - \theta_i\right)\left(y_i^{(k+1)} - y_i^{(k)}\right)\\ &= -(\text{net}_i)\,\Delta y_i\end{aligned}$$

where $\Delta y_i = y_i^{(k+1)} - y^{(k)}$. The change in energy is dependent on the fact that only one unit can update its activation at a time. The change in energy equation ΔE_f exploits the fact that $y_j^{(k+1)} = y_j^{(k)}$ for $j \neq i$ and $w_{ij} = w_{ji}$ and $w_{ii} = 0$ (symmetric weight property).

There exist two cases in which a change Δy_i will occur in the activation of neuron Y_i. If y_i is positive, then it will change to zero if

$$\left[x_i + \sum_{j=1}^{n} y_j w_{ji}\right] < \theta_i$$

This results in a negative change for y_i and $\Delta E_f < 0$. On the other hand, if y_i is zero, then it will change to positive if

$$\left[x_i + \sum_{j=1}^{n} y_j w_{ji}\right] > \theta_i$$

This results in a positive change for y_i and $\Delta E_f < 0$. Hence Δy_i is positive only if net input is positive and Δy_i is negative only if net input is negative. Therefore, the energy cannot increase in any manner. As a result,

67

because the energy is bounded, the net must reach a stable state equilibrium, such that the energy does not change with further iteration. From this it can be concluded that the energy change depends mainly on the change in activation of one unit and on the symmetry of weight matrix with zeros existing on the diagonal.

A Hopfield network always converges to a stable state in a finite number of node-updating steps, where every stable state is found to be at the local minima of the energy function E_f. Also, the proving process uses the well-known Lyapunov stability theorem, which is generally used to prove the stability of dynamic system defined with arbitrarily many interlocked differential equations. A positive-definite (energy) function $E_f(y)$ can be found such that:

1. $E_f(y)$ is continuous with respect to all the components y_i for $i = 1$ to n;
2. $d\,E_f[y(t)]/dt < 0$, which indicates that the energy function is decreasing with time and hence the origin of the state space is asymptotically stable.

Hence, a positive-definite (energy) function $E_f(y)$ satisfying the above requirements can be Lyapunov function for any given system; this function is not unique. If, at least one such function can be found for a system, then the system is asymptotically stable. According to the Lyapunov theorem, the energy function that is associated with a Hopfield network is a Lyapunov function and thus the discrete Hopfield network is asymptotically stable.

The storage capacity is another important factor. It can be found that the number of binary patterns that can be stored and recalled in a network with a reasonable accuracy is given approximately as

$$\text{Storage capacity } C \simeq 0.15n$$

where n is the number of neurons in the net. It can also be given as

$$C \cong \frac{n}{2\log_2 n}$$

4.6.2 Continuous Hopfield Network

A discrete Hopfield net can be modified to a continuous model, in which time is assumed to be a continuous variable, and can be used for associative memory problems or optimization problems like traveling salesman problem. The nodes of this network have a continuous, graded output rather than a two-state binary output. Thus, the energy of the network decreases continuously with time. The continuous Hopfield networks can be realized as an electronic circuit, which uses non-linear amplifiers and resistors. This helps building the Hopfield network using analog VLSI technology.

4.6.2.1 Hardware Model of Continuous Hopfield Network

The continuous network build up of electrical components is shown in Figure 4-8.

The model consists of n amplifiers, mapping its input voltage u_i into an output voltage y_i over an activation function $a(u_i)$. The activation function used can be a sigmoid function, say,

$$a(\lambda u_i) = \frac{1}{1 + e^{-\lambda u_i}}$$

where λ is called the gain parameter.

The continuous model becomes a discrete one when $\lambda \to \alpha$. Each of the amplifiers consists of an input capacitance c_i and an input conductance g_{ri}. The external signals entering into the circuit are x_i. The external

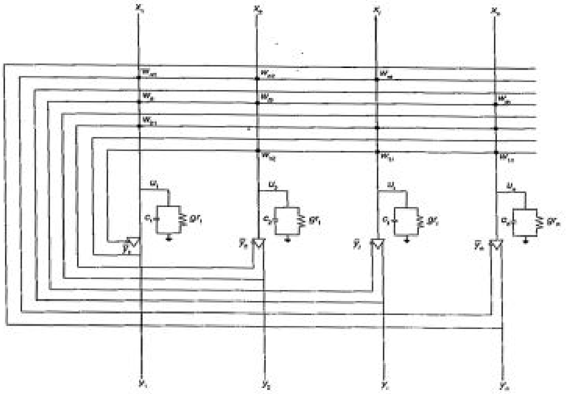


Figure 4-8 Model of Hopfield network using electrical components.

signals supply constant current to each amplifier for an actual circuit. The output of the jth node is connected to the input of the ith node through conductance w_{ij}. Since all real resistor values are positive, the inverted node outputs $\bar{y}_j$ are used to simulate the inhibitory signals. The connection is made with the signal from the noninverted output if the output of a particular node excites some other node. If the connection is inhibitory, then the connection is made with the signal from the inverted output. Here also, the important symmetric weight requirement for Hopfield network is imposed, i.e., $w_{ij} = w_{ji}$ and $w_{ii} = 0$.

The rule of each node in a continuous Hopfield network can be derived as shown in Figure 4-9. Consider the input of a single node as in Figure 4-9. Applying Kirchoff's current law (KCL), which states that the total current entering a junction is equal to that leaving the same function, we get

68

$$C_i\frac{du_i}{dt} = \sum_{\substack{j=1\\j\neq i}}^{n} w_{ij}(y_j - u_i) - g_{ri}u_i + x_i = \sum_{\substack{j=1\\j\neq i}}^{n} w_{ij}y_j - G_i u_i + x_i$$

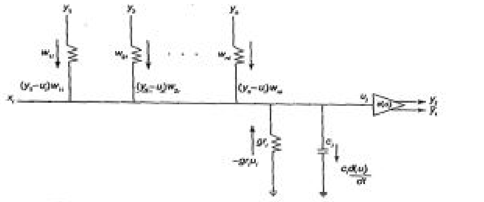


Figure 4-9 Input of a single node of continuous Hopfield network.

where

$$G_i = \sum_{\substack{j=i\\j\neq i}}^{n} w_{ij} + g_{ri}$$

The equation obtained using KCL describes the time evolution of the system completely. If each single node is given an initial value say, $u_i(0)$, then the value $u_i(t)$ and thus the amplifier output, $y_i(t) = a(u_i(t))$ at time t, can be known by solving the differential equation obtained using KCL.

4.6.2.2 Analysis of Energy Function of Continuous Hopfield Network

For evaluating the stability property of continuous Hopfield network, a continuous energy function is defined such that the evolution of the system is in the negative gradient of the energy function and finally converges to one of the table minima in the state space. The corresponding Lyapunov energy function for the model shown in Figure 4-8 is

$$E_f = -\frac{1}{2}\sum_{i=1}^{n}\sum_{\substack{j=1\\j\neq i}}^{n} w_{ij}y_iy_j - \sum_{i=1}^{n} x_iy_i + \frac{1}{\lambda}\sum_{i=1}^{n} G_i\int_0^{y_i} a^{-1}(y)dy$$

where $a^{-1}(y) = \lambda u$ is the inverse of the function $y = a(\lambda u)$. The inverse of the function $a^{-1}(y)$ is shown in Figure 4-10(A) and the integral of it in Figure 4-10(B).

To prove that E_f obtained is the Lyapunov function for the network, its time derivative is taken with weights w_{ij} symmetric:

$$\frac{dE_f}{dt} = \sum_{i=1}^{n}\frac{dE}{dy_i}\frac{dy_i}{dt} = \sum_{i=1}^{n}\left(-\sum_{\substack{j=1\\j\neq i}}^{n} y_jw_{ij} + G_iu_i - x_i\right)\frac{dy_i}{dt} = -\sum_{i=1}^{n} c_i\frac{dy_i}{dt}\frac{du_i}{dt}$$

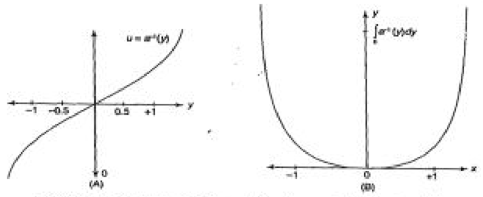


Figure 4-10 (A) Inverse and (B) integral of nonlinear activation function $a^{-1}(y)$.

As

$$u_i = \left(\frac{1}{\lambda}\right)a^{-1}(y_i)$$

we get

$$\frac{du_i}{dt} = \frac{1}{\lambda}\frac{da^{-1}(y_i)}{dy_i}\frac{dy_i}{dt} = \frac{1}{\lambda}a^{-1'}(y_i)\frac{dy_i}{dt}$$

where the derivative of $a^{-1}(y)$ is $a^{-1'}(y)$. So, the derivative of energy function equation becomes

$$\frac{dE_f}{dt} = -\sum_{i=1}^{n}\frac{1}{\lambda}c_ia^{-1'}(y_i)\left(\frac{dy_i}{dt}\right)^2$$

From Figure 4-10(A), we know that $a^{-1}(y_i)$ is a monotonically increasing function of y_i and hence its derivative is positive, all over. This shows that dE_f/dt is negative, and thus the energy function E_f must decrease as the system evolves. Therefore, if E_f is bounded, the system will eventually reach a stable state, where

$$\frac{dE_f}{dt} = \frac{dy_i}{dt} = 0$$

When the values of threshold are zero, the continuous energy function becomes equal to the discrete energy function, except for the term,

$$\frac{1}{\lambda}\sum_{i=1}^{n} G_i\int_0^{y_i} a^{-1}(y)dy$$

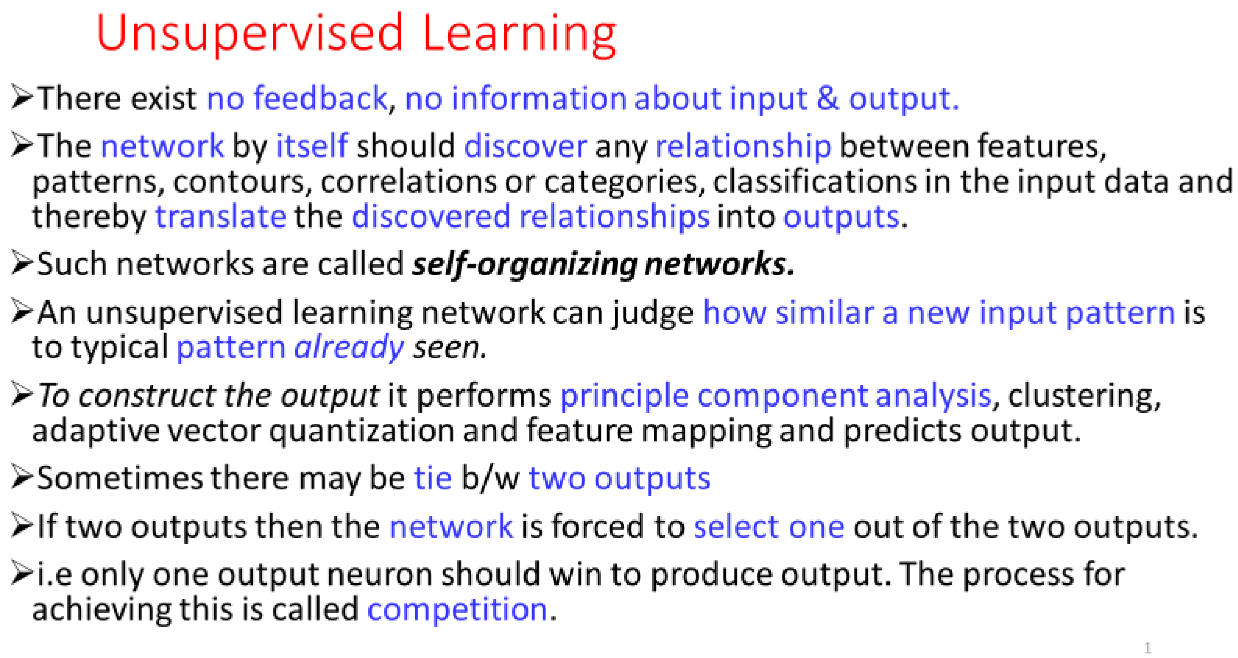
From Figure 4-10(B), the integral of $a^{-1}(y)$ is zero when y_i is zero and positive for all other values of y_i. The integral becomes very large as y approaches $+1$ or -1. Hence, the energy function E_f is bounded from below and is a Lyapunov function. The continuous Hopfield nets are best suited for the constrained optimization problems.

CVR COLLEGE OF ENGINEERING

An UGC Autonomous Institution - Affiliated to JNTUH

Handout – 3
Unit - 3
Year and Semester: IVyr &II Sem
Subject**: Soft Computing**
Branch: **CSE**
Faculty: **Dr.Md.Yusuf Mulge**, Professor (CSE)

Unsupervised Learning   CO3

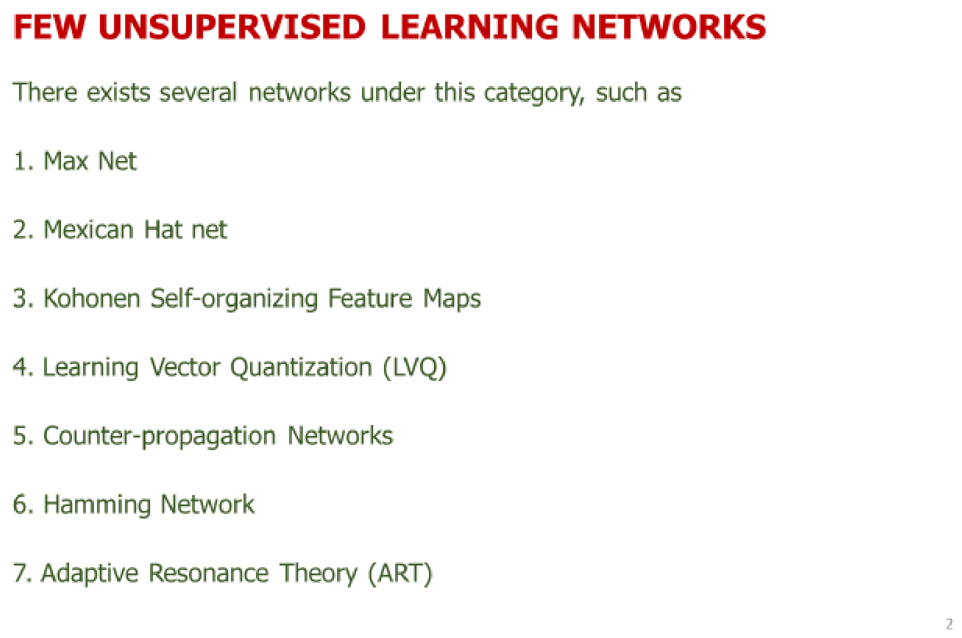


Unsupervised Learning

- There exist no feedback, no information about input & output.
- The network by itself should discover any relationship between features, patterns, contours, correlations or categories, classifications in the input data and thereby translate the discovered relationships into outputs.
- Such networks are called ***self-organizing networks.***
- An unsupervised learning network can judge how similar a new input pattern is to typical pattern *already seen.*
- *To construct the output* it performs principle component analysis, clustering, adaptive vector quantization and feature mapping and predicts output.
- Sometimes there may be tie b/w two outputs
- If two outputs then the network is forced to select one out of the two outputs.
- i.e only one output neuron should win to produce output. The process for achieving this is called competition.

1

Unsupervised Learning Networks CO3



FEW UNSUPERVISED LEARNING NETWORKS

There exists several networks under this category, such as

1. Max Net

2. Mexican Hat net

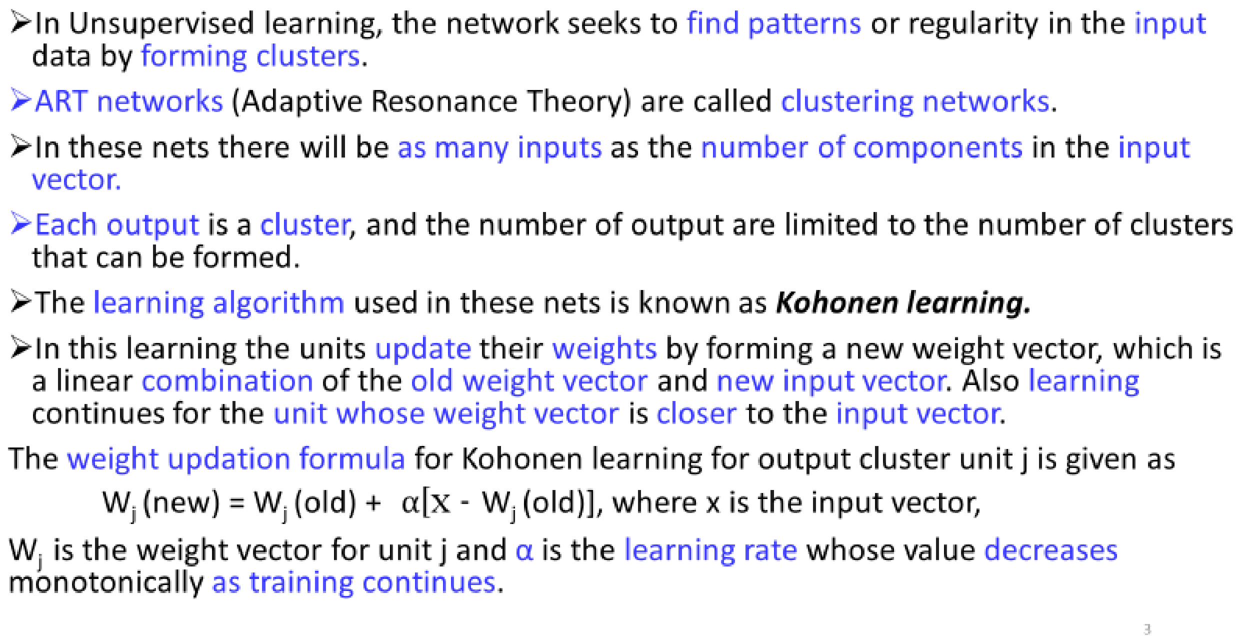
3. Kohonen Self-organizing Feature Maps

4. Learning Vector Quantization (LVQ)

5. Counter-propagation Networks

6. Hamming Network

7. Adaptive Resonance Theory (ART)



- In Unsupervised learning, the network seeks to find patterns or regularity in the input data by forming clusters.
- ART networks (Adaptive Resonance Theory) are called clustering networks.
- In these nets there will be as many inputs as the number of components in the input vector.
- Each output is a cluster, and the number of output are limited to the number of clusters that can be formed.
- The learning algorithm used in these nets is known as ***Kohonen learning.***
- In this learning the units update their weights by forming a new weight vector, which is a linear combination of the old weight vector and new input vector. Also learning continues for the unit whose weight vector is closer to the input vector.

The weight updation formula for Kohonen learning for output cluster unit j is given as

$$W_j\,(\text{new}) = W_j\,(\text{old}) + \alpha[x - W_j\,(\text{old})],$$ where x is the input vector,

W_j is the weight vector for unit j and α is the learning rate whose value decreases monotonically as training continues.

- There are two methods to determine the winner during competition:
- One method use Square of the Euclidean distance between the input vector and weight vector, and the unit whose weight vector is at the smallest Euclidean distance from input vector is chosen as the winner.
- In the second method, the dot product of input vector and weight vector is used, and the unit with largest dot product is the winner.

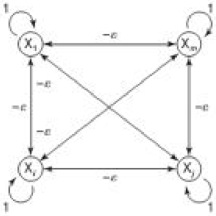
Fixed weight competitive Nets

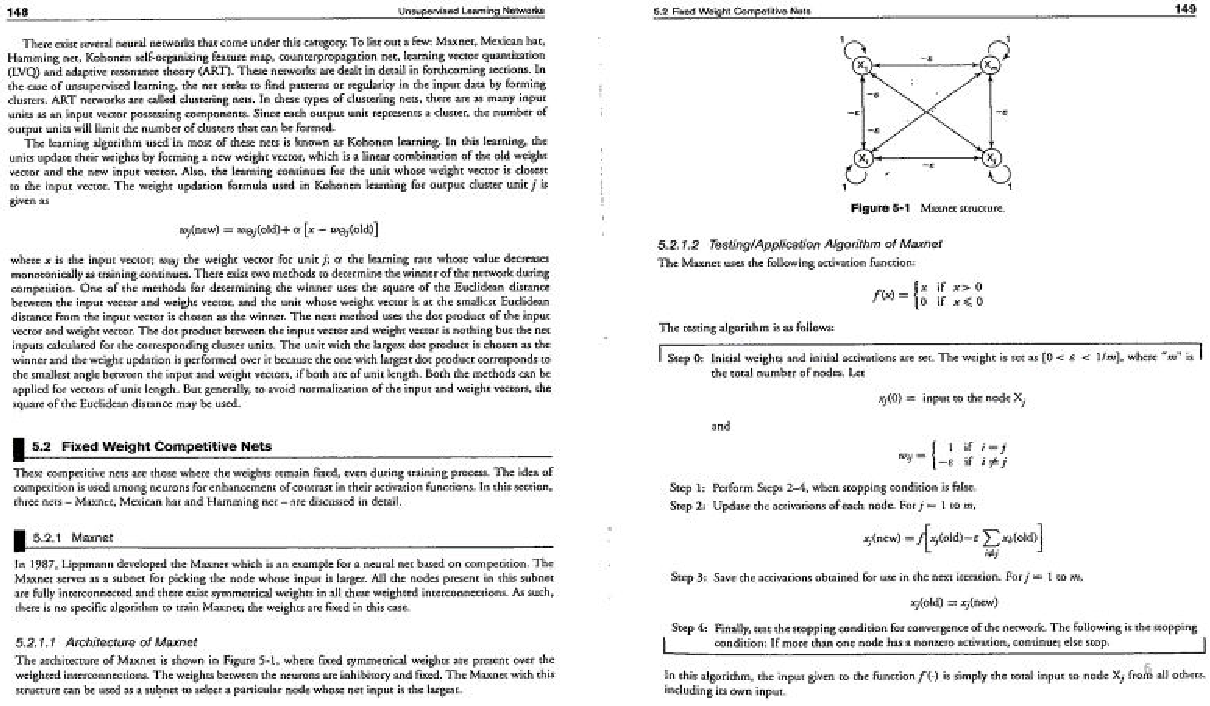
Competitive nets are those where the weights remain fixed. The competition is in the activation functions of neurons. In the following section 3 nets are discussed

Maxnet, Mexican hat net and Hamming net.

MAX NET

- Max Net is a fixed weight competitive net.
- Max Net serves as a subnet for picking the node whose input is larger. All the nodes present in this subnet are fully interconnected and there exist symmetrical weights in all these weighted interconnections.
- The weights between the neurons are inhibitory and fixed.
- The architecture of this net is as shown in the figure





There exist several neural networks that come under this category. To list out a few: Maxnet, Mexican hat, Hamming net, Kohonen self-organizing feature map, counterpropagation net, learning vector quantization (LVQ) and adaptive resonance theory (ART). These networks are dealt in detail in forthcoming sections. In the case of unsupervised learning, the net seeks to find patterns or regularity in the input data by forming clusters. ART networks are called clustering nets. In these types of clustering nets, there are as many input units as an input vector possessing components. Since each output unit represents a cluster, the number of output units will limit the number of clusters that can be formed.

The learning algorithm used in most of these nets is known as Kohonen learning. In this learning, the units update their weights by forming a new weight vector, which is a linear combination of the old weight vector and the new input vector. Also, the learning continues for the unit whose weight vector is closest to the input vector. The weight updation formula used in Kohonen learning for output cluster unit j is given as

$$w_j(\text{new}) = w_{0j}(\text{old}) + \alpha\left[x - w_{0j}(\text{old})\right]$$

where x is the input vector; w_{0j} the weight vector for unit j; α the learning rate whose value decreases monotonically as training continues. There exist two methods to determine the winner of the network during competition. One of the methods for determining the winner uses the square of the Euclidean distance between the input vector and weight vector, and the unit whose weight vector is at the smallest Euclidean distance from the input vector is chosen as the winner. The next method uses the dot product of the input vector and weight vector. The dot product between the input vector and weight vector is nothing but the net inputs calculated for the corresponding cluster units. The unit with the largest dot product is chosen as the winner and the weight updation is performed over it because the one with largest dot product corresponds to the smallest angle between the input and weight vectors, if both are of unit length. Both the methods can be applied for vectors of unit length. But generally, to avoid normalization of the input and weight vectors, the square of the Euclidean distance may be used.

5.2 Fixed Weight Competitive Nets

These competitive nets are those where the weights remain fixed, even during training process. The idea of competition is used among neurons for enhancement of contrast in their activation functions. In this section, three nets – Maxnet, Mexican hat and Hamming net – are discussed in detail.

5.2.1 Maxnet

In 1987, Lippmann developed the Maxnet which is an example for a neural net based on competition. The Maxnet serves as a subnet for picking the node whose input is larger. All the nodes present in this subnet are fully interconnected and there exist symmetrical weights in all these weighted interconnections. As such, there is no specific algorithm to train Maxnet; the weights are fixed in this case.

5.2.1.1 Architecture of Maxnet

The architecture of Maxnet is shown in Figure 5-1, where fixed symmetrical weights are present over the weighted interconnections. The weights between the neurons are inhibitory and fixed. The Maxnet with this structure can be used as a subnet to select a particular node whose net input is the largest.

Figure 5-1 Maxnet structure.

5.2.1.2 Testing/Application Algorithm of Maxnet

The Maxnet uses the following activation function:

$$f(x) = \begin{cases} x & \text{if } x > 0 \\ 0 & \text{if } x \leqslant 0 \end{cases}$$

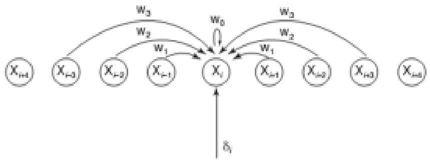
The testing algorithm is as follows:

```
Step 0: Initial weights and initial activations are set. The weight is set as [0 < ε < 1/m], where "m" is
        the total number of nodes. Let
                         $x_j(0) =$ input to the node $X_j$
        and
                         $w_{ij} = \begin{cases} 1 & \text{if } i = j \\ -\varepsilon & \text{if } i \neq j \end{cases}$
Step 1: Perform Steps 2–4, when stopping condition is false.
Step 2: Update the activations of each node. For $j = 1$ to $m$,
                         $x_j(\text{new}) = f\left[x_j(\text{old}) - \varepsilon \sum_{i \neq j} x_k(\text{old})\right]$
Step 3: Save the activations obtained for use in the next iteration. For $j = 1$ to $m$,
                         $x_j(\text{old}) = x_j(\text{new})$
Step 4: Finally, test the stopping condition for convergence of the network. The following is the stopping
        condition: If more than one node has a nonzero activation, continue; else stop.
```

In this algorithm, the input given to the function $f(\cdot)$ is simply the total input to node X_j from all others, including its own input.

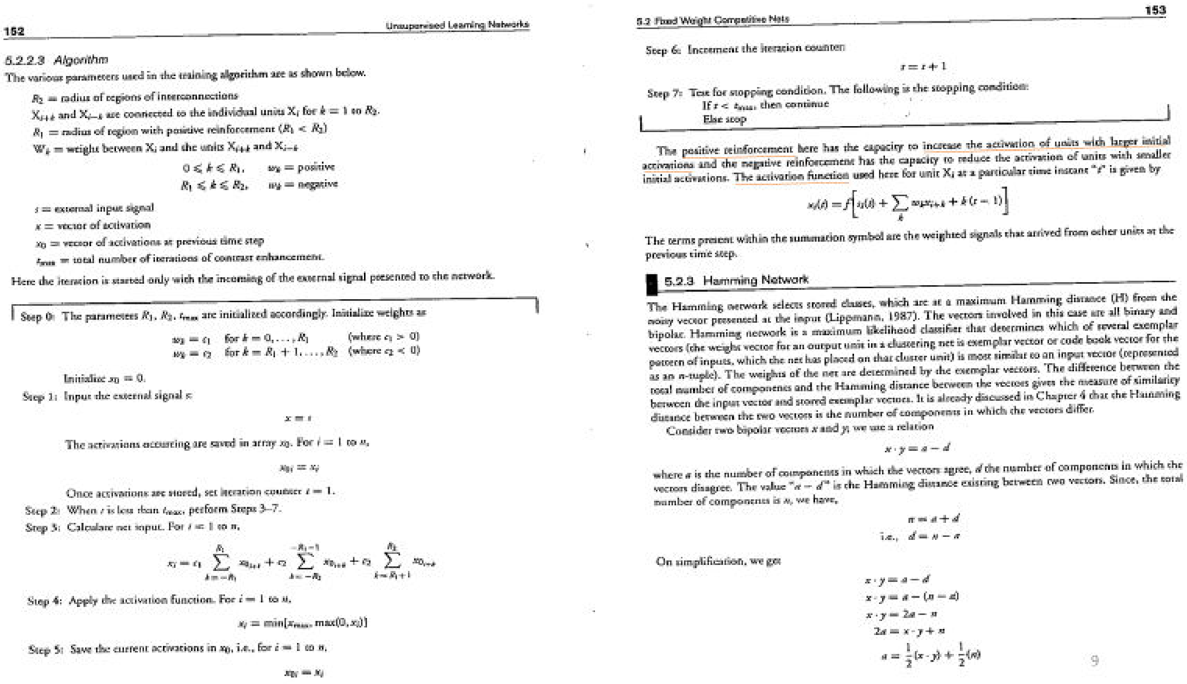
MEXICAN HAT NETWORK

- Kohonen developed the Mexican hat network which is a more generalized contrast enhancement network compared to the earlier Max Net.
- Here, in addition to the connections within a particular layer of neural net, the neurons also receive some other external signals. This interconnection pattern is repeated for several other neurons in the layer.
- The architecture for the network is as shown below:



Architecture

- As shown in the architecture, the interconnection b/w X_i and nearby neurons are connected with Positive weight (Diameter of 2, X_{i+1} , X_{i-1} , X_{i+2} , X_{i-2})
- Farther neurons connected with negative weight (Diameter of 3 X_{i+3} , X_{i-3})
- Farthest neuron-there is no connection (farther than diameter 3 X_{i+4} , X_{i-4})



5.2.2.3 Algorithm

The various parameters used in the training algorithm are as shown below.

R_2 = radius of regions of interconnections

X_{i+k} and X_{i-k} are connected to the individual units X_i for $k = 1$ to R_2.

R_1 = radius of region with positive reinforcement ($R_1 < R_2$)

W_k = weight between X_i and the units X_{i+k} and X_{i-k}

$$0 \leq k \leq R_1, \quad w_k = \text{positive}$$
$$R_1 \leq k \leq R_2, \quad w_k = \text{negative}$$

s = external input signal

x = vector of activation

x_0 = vector of activations at previous time step

t_{max} = total number of iterations of contrast enhancement.

Here the iteration is started only with the incoming of the external signal presented to the network.

Step 0: The parameters R_1, R_2, t_{max} are initialized accordingly. Initialize weights as

$$w_k = c_1 \quad \text{for } k = 0, \ldots, R_1 \quad (\text{where } c_1 > 0)$$
$$w_k = c_2 \quad \text{for } k = R_1 + 1, \ldots, R_2 \quad (\text{where } c_2 < 0)$$

Initialize $x_0 = 0$.

Step 1: Input the external signal s:

$$x = s$$

The activations occurring are saved in array x_0. For $i = 1$ to n,

$$x_{0i} = x_i$$

Once activations are stored, set iteration counter $t = 1$.

Step 2: When t is less than t_{max}, perform Steps 3–7.

Step 3: Calculate net input. For $i = 1$ to n,

$$x_i = c_1 \sum_{k=-R_1}^{R_1} x_{0_{i+k}} + c_2 \sum_{k=-R_2}^{-R_1-1} x_{0_{i+k}} + c_2 \sum_{k=R_1+1}^{R_2} x_{0_{i+k}}$$

Step 4: Apply the activation function. For $i = 1$ to n,

$$x_i = \min[x_{max}, \max(0, x_i)]$$

Step 5: Save the current activations in x_0, i.e., for $i = 1$ to n,

$$x_{0i} = x_i$$

Step 6: Increment the iteration counter:

$$t = t + 1$$

Step 7: Test for stopping condition. The following is the stopping condition:
If $t < t_{max}$, then continue
Else stop

The positive reinforcement here has the capacity to increase the activation of units with larger initial activations and the negative reinforcement has the capacity to reduce the activation of units with smaller initial activations. The activation function used here for unit X_i at a particular time instant "t" is given by

$$x_i(t) = f\left[s_i(t) + \sum_k w_k x_{i+k} + k\,(t-1)\right]$$

The terms present within the summation symbol are the weighted signals that arrived from other units at the previous time step.

5.2.3 Hamming Network

The Hamming network selects stored classes, which are at a maximum Hamming distance (H) from the noisy vector presented at the input (Lippmann, 1987). The vectors involved in this case are all binary and bipolar. Hamming network is a maximum likelihood classifier that determines which of several exemplar vectors (the weight vector for an output unit in a clustering net is exemplar vector or code book vector for the pattern of inputs, which the net has placed on that cluster unit) is most similar to an input vector (represented as an n-tuple). The weights of the net are determined by the exemplar vectors. The difference between the total number of components and the Hamming distance between the vectors gives the measure of similarity between the input vector and stored exemplar vectors. It is already discussed in Chapter 4 that the Hamming distance between the two vectors is the number of components in which the vectors differ.

Consider two bipolar vectors x and y; we use a relation

$$x \cdot y = a - d$$

where a is the number of components in which the vectors agree, d the number of components in which the vectors disagree. The value "$a - d$" is the Hamming distance existing between two vectors. Since, the total number of components is n, we have,

$$n = a + d$$
$$\text{i.e.,} \quad d = n - a$$

On simplification, we get

$$x \cdot y = a - d$$
$$x \cdot y = a - (n - a)$$
$$x \cdot y = 2a - n$$
$$2a = x \cdot y + n$$
$$a = \frac{1}{2}(x \cdot y) + \frac{1}{2}(n)$$

HAMMING NETWORK

- The Hamming network selects stored classes, which are at a maximum Hamming distance (H) from the noisy vector presented at the input.

- The Hamming distance between the two vectors is the number of components in which the vectors differ.

- The Hamming network consists of two layers.
  - The first layer computes the difference between the total number of components in the two given vectors and also computes Hamming distance between the input vector x and the stored pattern of vectors in the feed forward path.
  - The second layer of the Hamming network is composed of Max Net (used as a subnet) or a winner-take-all network which is a recurrent network, which finds the exemplar unit that best matches with the input (winner unit).

Consider two bipolar vectors x and y, we use a relation

$x.y = a-d$ --------- (1)

Where a is the number of components in which the vectors agree, d the number of components in which the vectors disagree. The value a-d is the Hamming distance b/w two vectors. Since, the total number of components is n, we have

$n = a + d$

i.e. $d = n - a$ ------------- (2)

Substituting the value of d in equation 1, we get

$x.y = a - (n - a)$

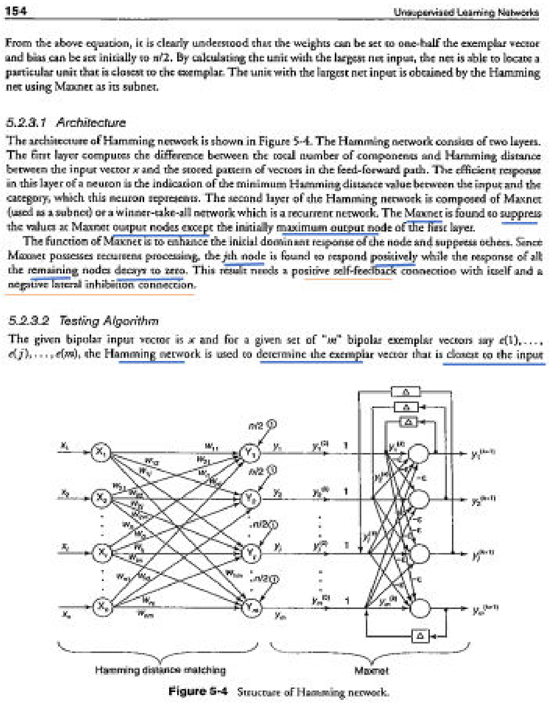
$x.y = 2a - n$

$2a = x.y + n$

$a = \frac{1}{2}(x.y) + \frac{1}{2}(n)$ ------------------ (3)

From equation 3 it is clear that, weights can be set to ½ of the exemplar vector (1/2 x.y) and bias can be set to n/2.

- The unit with largest net input is the closest unit to the exemplar.
- The unit with the largest net input is obtained by the Hamming net using Maxnet as its subnet.



From the above equation, it is clearly understood that the weights can be set to one-half the exemplar vector and bias can be set initially to $n/2$. By calculating the unit with the largest net input, the net is able to locate a particular unit that is closest to the exemplar. The unit with the largest net input is obtained by the Hamming net using Maxnet as its subnet.

5.2.3.1 Architecture

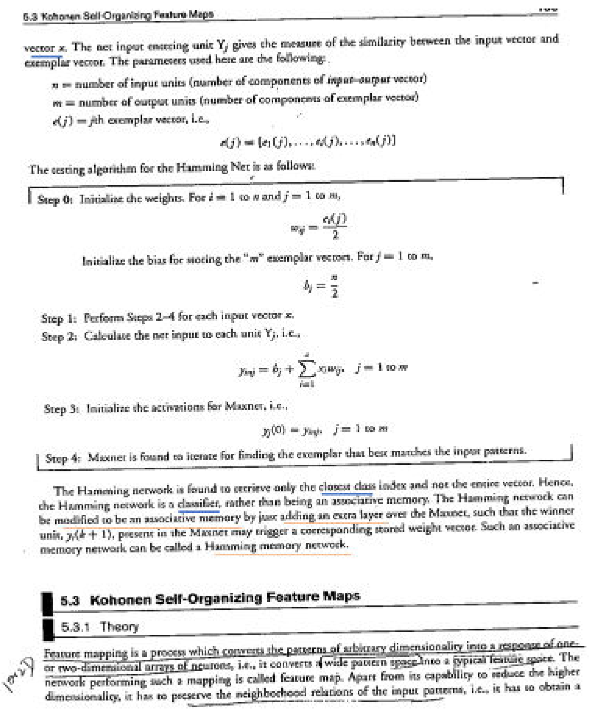
The architecture of Hamming network is shown in Figure 5-4. The Hamming network consists of two layers. The first layer computes the difference between the total number of components and Hamming distance between the input vector x and the stored pattern of vectors in the feed-forward path. The efficient response in this layer of a neuron is the indication of the minimum Hamming distance value between the input and the category, which this neuron represents. The second layer of the Hamming network is composed of Maxnet (used as a subnet) or a winner-take-all network which is a recurrent network. The Maxnet is found to suppress the values at Maxnet output nodes except the initially maximum output node of the first layer.

The function of Maxnet is to enhance the initial dominant response of the node and suppress others. Since Maxnet possesses recurrent processing, the jth node is found to respond positively while the response of all the remaining nodes decays to zero. This result needs a positive self-feedback connection with itself and a negative lateral inhibition connection.

5.2.3.2 Testing Algorithm

The given bipolar input vector is x and for a given set of "m" bipolar exemplar vectors say $e(1), \ldots, e(j), \ldots, e(m)$, the Hamming network is used to determine the exemplar vector that is closest to the input

Figure 5-4 Structure of Hamming network.



vector x. The net input entering unit Y_j gives the measure of the similarity between the input vector and exemplar vector. The parameters used here are the following:

n = number of input units (number of components of *input–output* vector)
m = number of output units (number of components of exemplar vector)
$e(j)$ = jth exemplar vector, i.e.,

$$e(j) = [e_1(j), \ldots, e_i(j), \ldots, e_n(j)]$$

The testing algorithm for the Hamming Net is as follows:

Step 0: Initialize the weights. For $i = 1$ to n and $j = 1$ to m,

$$w_{ij} = \frac{e_i(j)}{2}$$

Initialize the bias for storing the "m" exemplar vectors. For $j = 1$ to m,

$$b_j = \frac{n}{2}$$

Step 1: Perform Steps 2–4 for each input vector x.

Step 2: Calculate the net input to each unit Y_j, i.e.,

$$y_{inj} = b_j + \sum_{i=1}^{n} x_i w_{ij}, \quad j = 1 \text{ to } m$$

Step 3: Initialize the activations for Maxnet, i.e.,

$$y_j(0) = y_{inj}, \quad j = 1 \text{ to } m$$

Step 4: Maxnet is found to iterate for finding the exemplar that best matches the input patterns.

The Hamming network is found to retrieve only the closest class index and not the entire vector. Hence, the Hamming network is a classifier, rather than being an associative memory. The Hamming network can be modified to be an associative memory by just adding an extra layer over the Maxnet, such that the winner unit, $y_j(k+1)$, present in the Maxnet may trigger a corresponding stored weight vector. Such an associative memory network can be called a Hamming memory network.

5.3 Kohonen Self-Organizing Feature Maps

5.3.1 Theory

1 or 2 D

Feature mapping is a process which converts the patterns of arbitrary dimensionality into a response of one- or two-dimensional arrays of neurons, i.e., it converts a wide pattern space into a typical feature space. The network performing such a mapping is called feature map. Apart from its capability to reduce the higher dimensionality, it has to preserve the neighborhood relations of the input patterns, i.e., it has to obtain a

Kohonen self-organizing feature Maps

- Feature mapping is a process which converts the patterns of arbitrary dimensionality into a response of one or two-dimensional arrays of neurons i.e. it converts a wide pattern space into a typical feature space.
- The network performing such a mapping is called feature map.
- Apart from its capability to reduce the higher dimensionality, it has to preserve the neighborhood relations of the input pattern, i.e it has to obtain a topology preserving map.
- For obtaining such feature maps, it is required to find a self-organizing neural array which consists of neurons arranged in a one-dimensional array or a two-dimensional array.
- To depict this, a typical network structure where each component of the input vector x is connected to each of the nodes is shown in figure 5.5.
- On the other hand if the input vector is two-dimensional array then the inputs x(a,b) can arrange themselves in a two dimensional array defining the input space (a,b) as shown in figure 5-6. Here the two layers are fully connected.

- Figure 5.7 shows linear array of cluster units

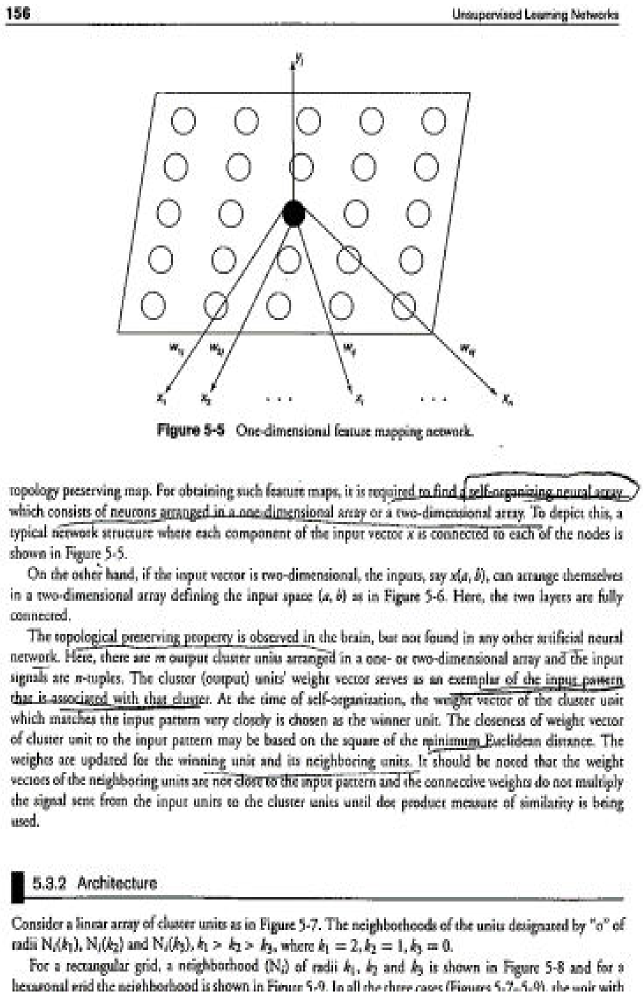
The winning unit is indicated by '#' and remaining by '0'

Figure 5.8 and 5.9 shows rectangular and Hexagonal grids where k_1=2, k_2=1 & k_3=0

Hence $k_1 > k_2 > k_3$

Rectangular grid has – 8 nearest neighbors

Hexagonal grid has – 6 nearest neighbors


y_j
w_{1j}
w_{2j}
w_{ij}
w_{nj}
x_1
x_2
. . .
x_i
. . .
x_n

156 Unsupervised Learning Networks

Figure 5-5 One-dimensional feature mapping network.

topology preserving map. For obtaining such feature maps, it is required to find a self-organizing neural array which consists of neurons arranged in a one-dimensional array or a two-dimensional array. To depict this, a typical network structure where each component of the input vector *x* is connected to each of the nodes is shown in Figure 5-5.

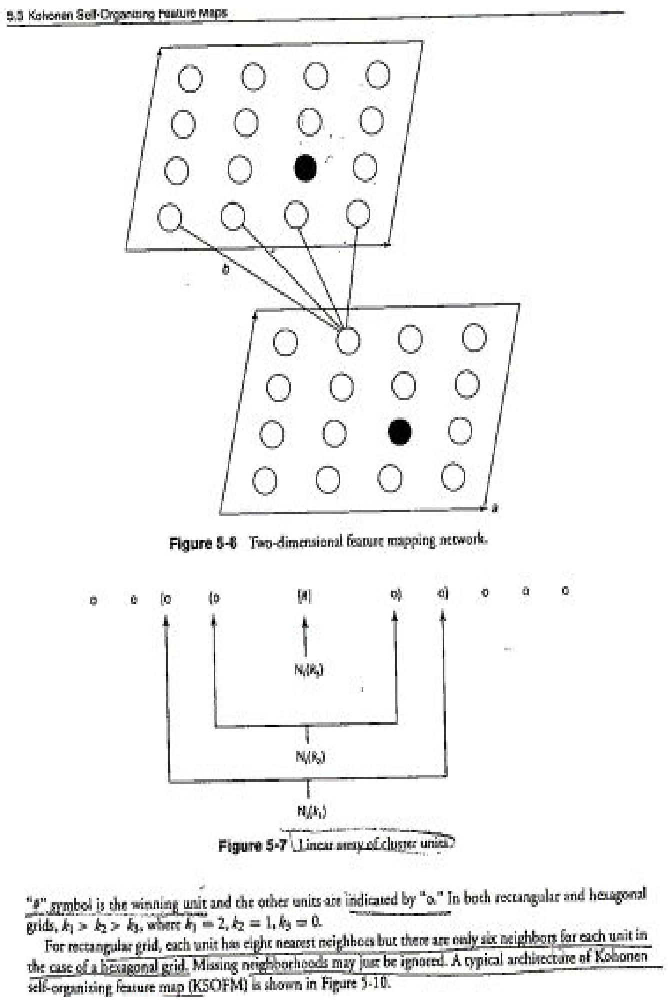
On the other hand, if the input vector is two-dimensional, the inputs, say $x(a, b)$, can arrange themselves in a two-dimensional array defining the input space (a, b) as in Figure 5-6. Here, the two layers are fully connected.

The topological preserving property is observed in the brain, but not found in any other artificial neural network. Here, there are *m* output cluster units arranged in a one- or two-dimensional array and the input signals are *n*-tuples. The cluster (output) units' weight vector serves as an exemplar of the input pattern that is associated with that cluster. At the time of self-organization, the weight vector of the cluster unit which matches the input pattern very closely is chosen as the winner unit. The closeness of weight vector of cluster unit to the input pattern may be based on the square of the minimum Euclidean distance. The weights are updated for the winning unit and its neighboring units. It should be noted that the weight vectors of the neighboring units are not close to the input pattern and the connective weights do not multiply the signal sent from the input units to the cluster units until dot product measure of similarity is being used.

5.3.2 Architecture

Consider a linear array of cluster units as in Figure 5-7. The neighborhoods of the units designated by "o" of radii $N_i(k_1)$, $N_i(k_2)$ and $N_i(k_3)$, $k_1 > k_2 > k_3$, where $k_1 = 2$, $k_2 = 1$, $k_3 = 0$.

For a rectangular grid, a neighborhood (N_i) of radii k_1, k_2 and k_3 is shown in Figure 5-8 and for a hexagonal grid the neighborhood is shown in Figure 5-9. In all the three cases (Figures 5-7–5-9), the unit with


b
a
o o (o (o (#) o) o) o o o
$N_i(k_3)$
$N_i(k_2)$
$N_i(k_1)$

5.3 Kohonen Self-Organizing Feature Maps

Figure 5-6 Two-dimensional feature mapping network.

Figure 5-7 Linear array of cluster units.

"#" symbol is the winning unit and the other units are indicated by "o." In both rectangular and hexagonal grids, $k_1 > k_2 > k_3$, where $k_1 = 2$, $k_2 = 1$, $k_3 = 0$.

For rectangular grid, each unit has eight nearest neighbors but there are only six neighbors for each unit in the case of a hexagonal grid. Missing neighborhoods may just be ignored. A typical architecture of Kohonen self-organising feature map (KSOFM) is shown in Figure 5-10.

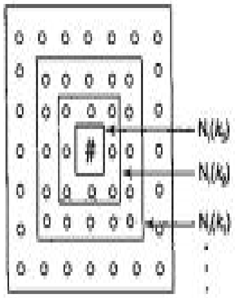


Figure 5-8 Rectangular grid.

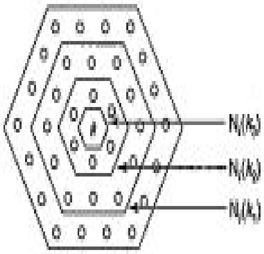


Figure 5-9 Hexagonal grid.

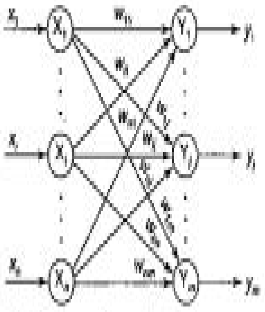


Figure 5-10 Kohonen self-organizing feature map architecture.

5.3.3 Flowchart

The flowchart for KSOFM is shown in Figure 5-11, which indicates the flow of training process. The process is continued for particular number of epochs or till the learning rate reduces to a very small rate.

The architecture consists of two layers: input layer and output layer (cluster). There are "n" units in the input layer and "m" units in the output layer. Basically, here the winner unit is identified by using either dot product or Euclidean distance method and the weight updation using Kohonen learning rules is performed over the winning cluster unit.

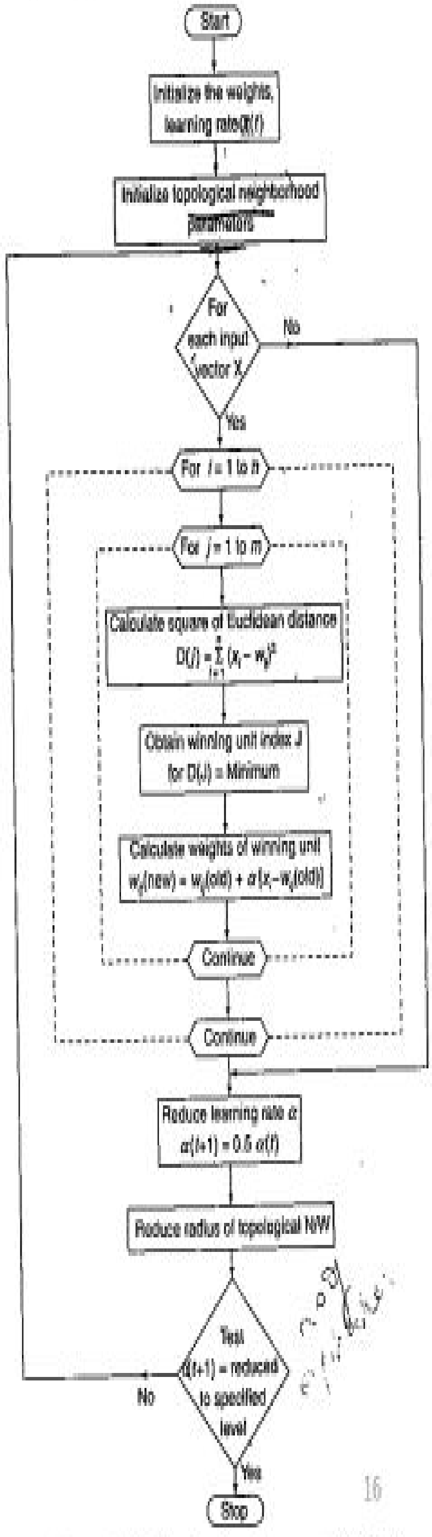

Start
Initialize the weights, learning rate $\alpha(t)$
Initialize topological neighborhood parameters
For each input vector X
No
Yes
For i = 1 to n
For j = 1 to m
Calculate square of Euclidean distance
$D(j) = \sum_{i=1}^{n} (x_i - w_{ij})^2$
Obtain winning unit index J for D(J) = Minimum
Calculate weights of winning unit
$w_{ij}(\text{new}) = w_{ij}(\text{old}) + \alpha\,[x_i - w_{ij}(\text{old})]$
Continue
Continue
Reduce learning rate α
$\alpha(t+1) = 0.5\,\alpha(t)$
Reduce radius of topological N/W
Test $\alpha(t+1)$ = reduced to specified level
No
Yes
Stop

Figure 5-11 Flowchart for training process of KSOFM.

5.3.4 Training Algorithm

The steps involved in the training algorithm are as shown below.

```
Step 0: • Initialize the weights $w_{ij}$: Random values may be assumed. They can be chosen as the same
          range of values as the components of the input vector. If information related to distribution
          of clusters is known, the initial weights can be taken to reflect that prior knowledge.
        • Set topological neighborhood parameters: As clustering progresses, the radius of the neighbor-
          hood decreases.
        • Initialize the learning rate $\alpha$: It should be a slowly decreasing function of time.
Step 1: Perform Steps 2–8 when stopping condition is false.
Step 2: Perform Steps 3–5 for each input vector $x$.
Step 3: Compute the square of the Euclidean distance, i.e., for each $j = 1$ to $m$,
            $D(j) = \sum_{i=1}^{n}\sum_{j=1}^{m}(x_i - w_{ij})^2$
Step 4: Find the winning unit index J, so that D(J) is minimum. (In Steps 3 and 4, dot product method
        can also be used to find the winner, which is basically the calculation of net input, and the winner
        will be the one with the largest dot product.)
Step 5: For all units $j$ within a specific neighborhood of J and for all $i$, calculate the new weights:
            $w_{ij}(\text{new}) = w_{ij}(\text{old}) + \alpha[x_i - w_{ij}(\text{old})]$
        or
            $w_{ij}(\text{new}) = (1-\alpha)w_{ij}(\text{old}) + \alpha x_i$
Step 6: Update the learning rate $\alpha$ using the formula $\alpha(t+1) = 0.5\alpha(t)$.
Step 7: Reduce radius of topological neighborhood at specified time intervals.
Step 8: Test for stopping condition of the network.
```

Thus using this training algorithm, an efficient training can be performed for an unsupervised learning network.

5.3.5 Kohonen Self-Organizing Motor Map

The extension of Kohonen feature map for a multilayer network involves the addition of an association layer to the output of the self-organizing feature map layer. The output node is found to associate the desired output values with certain input vectors. This type of architecture is called as Kohonen self-organizing motor map (KSOMM; Ritter, 1992) and layer that is added is called a motor map in which the movement command are being mapped into two-dimensional locations of excitation. The architecture of KSOMM is shown in Figure 5-12. Here, the feature map is a hidden layer and this acts as a competitive network which classifies the input vectors. The feature map is trained as discussed in Section 5.3.3. The motor map formation is based on the learning of a control task. The motor map learning may be either supervised or unsupervised learning and can be performed by delta learning rule or outstar learning rule (to be discussed later). The motor map learning is an extension of Kohonen's original learning algorithm.

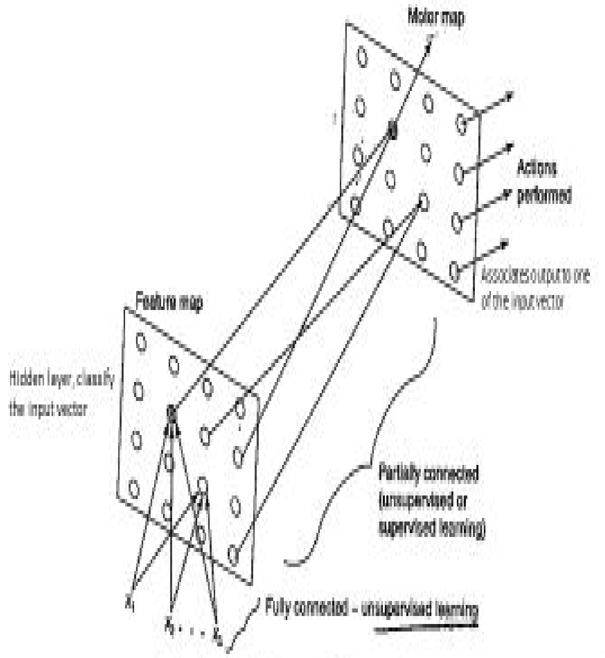

Motor map
Actions performed
Associates output to one of the input vector
Feature map
Hidden layer, classify the input vector
Partially connected (unsupervised or supervised learning)
x_1
x_i . . . x_n
Fully connected – unsupervised learning

Figure 5-12 Architecture of Kohonen self-organizing motor map.

5.4 Learning Vector Quantization

5.4.1 Theory

Learning vector quantization (LVQ) is a process of classifying the patterns, wherein each output unit represents a particular class. Here, for each class several units should be used. The output unit weight vector is called the reference vector or code book vector for the class which the unit represents. This is a special case of competitive net, which uses supervised learning methodology. During training, the output units are found to be positioned to approximate the decision surfaces of the existing Bayesian classifier. Here, the set of training patterns with known classifications is given to the network, along with an initial distribution of the reference vectors. When the training process is complete, an LVQ net is found to classify an input vector by assigning it to the same class as that of the output unit, which has its weight vector very close to the input vector. Thus, LVQ is a classifier paradigm that adjusts the boundaries between categories to minimize existing misclassification. LVQ is used for optical character recognition, converting speech into phonemes and other applications as well. LVQ net may resemble KSOFM net. Unlike LVQ, KSOFM output nodes do not correspond to the known classes but rather correspond to unknown clusters that the KSOFM finds in the data autonomously.

5.4.2 Architecture

Figure 5-13 shows the architecture of LVQ, which is almost the same as that of KSOFM, with the difference being that in the case of LVQ, the topological structure at the output unit is not being considered. Here, each output unit has knowledge about what a known class represents.

From Figure 5-13 it can be noticed that there exists input layer with "n" units and output layer with "m" units. The layers are found to be fully interconnected with weighted linkage acting over the links.

Learning vector Quantization(LVQ)

- Learning vector quantization (LVQ) is a process of classifying the patterns, wherein each output represents a particular class.
- Here, every class has several units.
- The output unit weight vector is called the reference vector or code book vector.
- During training the output units are found to be positioned to approximate the decision surfaces of the existing Baysian classifier.
- The set of training patterns with known classifications is given to the network.
- After the training is completed an LVQ net is found to classify an input vector by assigning the same class as that of output unit.
- LVQ adjusts the boundaries between categories to minimize the existing misclassification.
- LVQ is used for optical character recognition, which converts speech into phonemes and other applications.

Architecture

Figure 5-13 shows the architecture of LVQ, which is almost same as KSOFM, with the difference being that in LVQ the topological structure at the output unit is not considered.

- Here each output unit has a known class.
- The input layer has n units and output layer has m units. The layers are fully connected by weighted links.

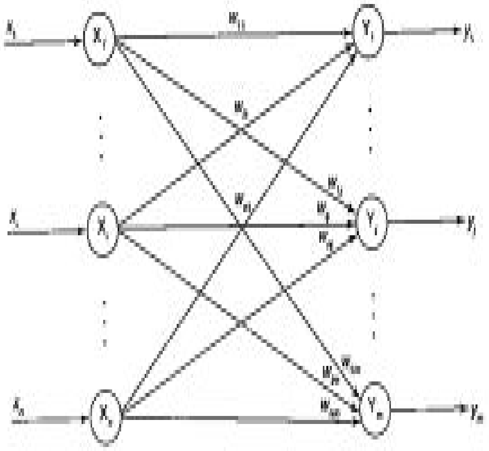


Figure 5-13 Architecture of LVQ.

5.4.3 Flowchart

The parameters used for the training process of a LVQ include the following:

x = training vector $(x_1, \ldots, x_i, \ldots, x_n)$

T = category or class for the training vector x

w_j = weight vector for jth output unit $(w_{1j}, \ldots, w_{ij}, \ldots, w_{nj})$

c_j = cluster or class or category associated with jth output unit.

The Euclidean distance of jth output unit is $D(j) = \sum (x_i - w_{ij})^2$. The flowchart indicating the flow of training process is shown in Figure 5-14.

5.4.4 Training Algorithm

In case of training, a set of training input vectors with a known classification is provided with some initial distribution of reference vector. Here, each output unit will have a known class. The objective of the algorithm is to find the output unit that is closest to the input vector.

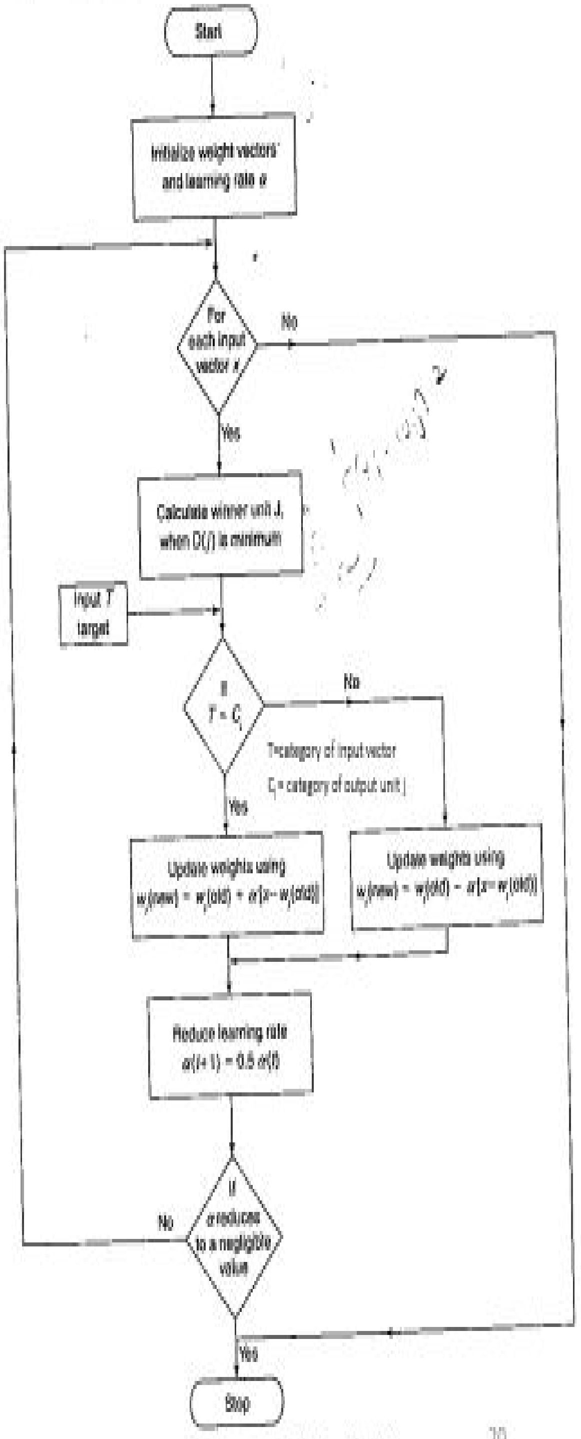
Step 0: Initialize the reference vectors. This can be done using the following steps.

- From the given set of training vectors, take the first "m" (number of clusters) training vectors and use them as weight vectors, the remaining vectors can be used for training.
- Assign the initial weights and classifications randomly.
- K-means clustering method.

Set initial learning rate α.

Step 1: Perform Steps 2–6 if the stopping condition is false.

Step 2: Perform Steps 3–4 for each training input vector x.



Start
Initialize weight vectors and learning rate α
For each input vector x
No
Yes
Calculate winner unit J, when $D(j)$ is minimum
Input T target
If $T = C_j$
No
T=category of input vector
C_j = category of output unit j
Yes
Update weights using $w_j(\text{new}) = w_j(\text{old}) + \alpha[x - w_j(\text{old})]$
Update weights using $w_j(\text{new}) = w_j(\text{old}) - \alpha[x - w_j(\text{old})]$
Reduce learning rate $\alpha(t+1) = 0.5\,\alpha(t)$
If α reduces to a negligible value
No
Yes
Stop

Figure 5-14 Flowchart for LVQ.

Step 3: Calculate the Euclidean distance for i =1 to n, j=1 to m

$$D(j) = \sum_{i=1}^{n}\sum_{j=1}^{m} (X_i - W_{ij})^2$$

Find the winning unit index, when D(j) is minimum.

Setp 4: Update the weights on the winning unit, W_j using the following conditions

if T = Cj, then Wj(new) = Wj(old) + α [x – Wj (old)]

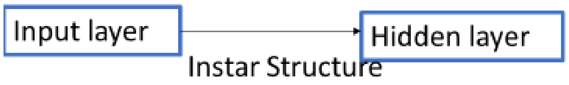
if T ≠ Cj, then Wj(new) = Wj(old) - α [x – Wj (old)]

Step 5: Reduce the learning rate α .

Step 6: Test for the stopping condition of the training process (the stopping condition may be fixed number of epochs or if learning rate has reduced to a negligible value.)

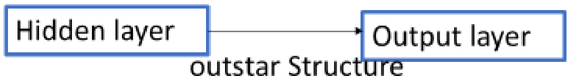
Conuter-propogation Networks

- Counter-propogation networks were proposed by Hechr Nielsen in 1987.
- These are multilayer networks based on the combination of the input, output and clustering layers.
- Applications – data compression, function approximation and pattern association
- This network is constructed from an instar-outstar model.
- It performs input-output mapping, producing an output vector in response to an input vector x, on the basis of competitive learning.
- The three layers in an instar-outstar model are the input layer, hidden layer (competitive) and output layer.
- All the layers are fully connected.


Input layer

Hidden layer

Instar Structure


Hidden layer

Output layer

outstar Structure

- In counter-propogation net a look-up-table is constructed which consists of training input vector pairs.
- The accuracy of the function approximation is based on the number of entries in the look-up-table, which equals the number of units in the cluster layer of the net

There are two stages in the training of the counter-propogation net:

1st stage: The input vectors are clustered using Euclidean distance method or dot product method.

2nd stage: The weights from the input unit are tuned to the output layer to get the desired output.

There are two types of counter-propogation nets:

1. Full counter-propogation net
2. Forward-only counter-propogation net.

Full counter-propogation Net

- Full counter-propogation net (full CPN) has many i/p vector pairs x.y by constructing a look-up-table.
- The output is x*. y*, which is based on input vector pairs with distorted or missing elements in either or both vectors.
- The network approximates a continuous function.
- The full CPN works best if the inverse function exists and is continuous.
- During competition the winner can be determined either by Euclidean distance or by dot product method.
- In dot product method, the one with the largest input is the winner.
- To avoid complexity Euclidean distance method can be used.
- Based on this, direct comparison of full CPN and forward-only CPN can be made.

- For continuous function, CPN is as efficient as back-propogation net.
- But advantage is that, to achieve a particular level of accuracy, the number of nodes in hidden layer of CPN are more than that of back-propogation.
- Another advantage is that, it require few steps of training to achieve best performance. This is true for any hybrid learning method that combines unsupervised learning (instar learning) and supervised learning (outstar learning).
- The training of CPN occurs in two phases:

1. Input phase: In this phase the units in input layer and output layer are active.

- There is no fixed topology in the output layer; only the winning units are allowed to learn.

The weight updation in learning units is as follows

$$V_{ij}\,(\text{new}) = V_{ij}\,(\text{old}) + \alpha\,[x_i - V_{ij}\,(\text{old})],\ i = 1 \text{ to } n$$

$$W_{kj}\,(\text{new}) = W_{kj}\,(\text{old}) + \beta[y_k - W_{kj}\,(\text{old})],\ k = 1 \text{ to } m$$

- The above is standard *Kohonen* learning which consists of competition among the units and selection of winner unit. The weight updation is performed for the winning unit.

2. In the second phase, only the winner unit J remains active in the cluster layer.

- The weights between the winning cluster unit J and output units are adjusted to get Y-output as Y*, which is an approximation of input vector y and X-output as X*, which is an approximation of input vector x.

The weight updation for the units in Y-output and X-output layers are

$$u_{Jk}(\text{new}) = u_{Jk}(\text{old}) + a[y_k - u_{Jk}(\text{old})],\ k = 1 \text{ to } m - Y \text{ output layer}$$

$$t_{Ji}(\text{new}) = t_{Ji}(\text{old}) + b[x_i - t_{Ji}(\text{old})],\ i = 1 \text{ to } n - x \text{ output layer}$$

- This is Grossberg learning, a more general case of outstar learning. It can be applied to all the units in a layer.
- For output layer *delta learning* can also be used.
- The weight updation in this case is the product of learning rate and error.
- When tie occurs the unit with smallest index is chosen as the winner.

Architecture

The general structure of full CPN is shown in figure 5-15. The complete architecture of full CPN is shown in figure 5-16.

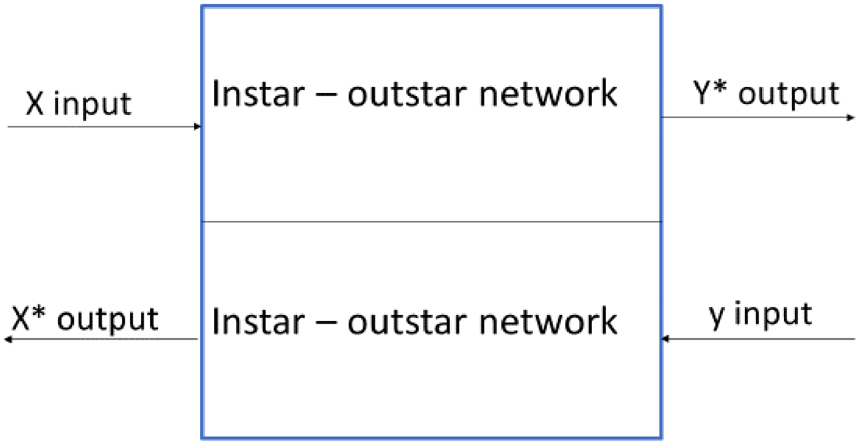


Figure 5-15 General structure of full CPN

- The four major components of instar-outstar model are the 1) input layer, 2) the instar, 3) the competitive layer and 4) the outstar.
- For each node i in the input layer, there is an input value x_i.
- An instar responds to the input vectors from a particular cluster only.
- All the instars are grouped into competitive layer.
- The winning instar has strongest response and the competitive layer set its output to a non-zero value and all other outputs are set to zero.
- That is, it is a winner-take-all or a Maxnet-type of network.
- The outstar looks like the fan-out of a node.
- Figure 5-17 and 5-18 indicate the units that are active during each of the two phases of training a full CPN.

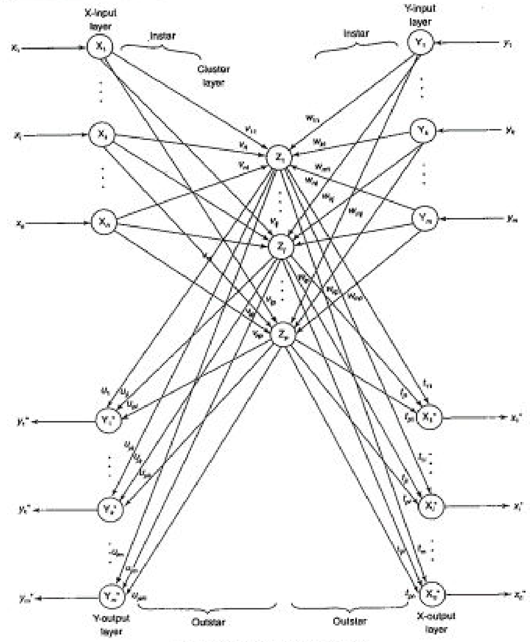


Figure 5-16 Architecture of full CPN.

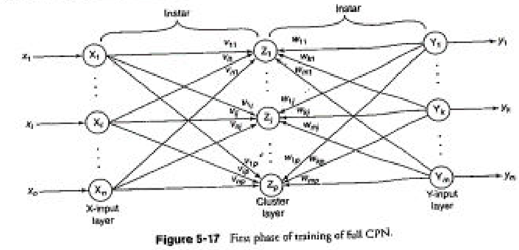


Figure 5-17 First phase of training of full CPN.

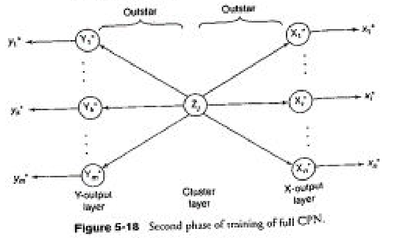


Figure 5-18 Second phase of training of full CPN.

5.5.2.2 *Flowchart*

The flowchart for the training process of full CPN is shown in Figure 5-19. The parameters used in the CPN are as follows:

x = input training vector $x = (x_1, \ldots, x_i, \ldots, x_n)$
y = target output corresponding to input x, $y = (y_1, \ldots, y_k, \ldots, y_m)$
z_j = the output of cluster layer unit z_j
v_{ij} = weight from X-input layer unit X_i to cluster layer unit z_j
w_{kj} = weight from Y-input layer unit Y_k to cluster layer unit z_j
u_{jk} = weight from cluster layer unit z_j to Y-output layer unit Y_k^*
t_{ji} = weight from cluster layer unit z_j to X-output layer unit X_i^*

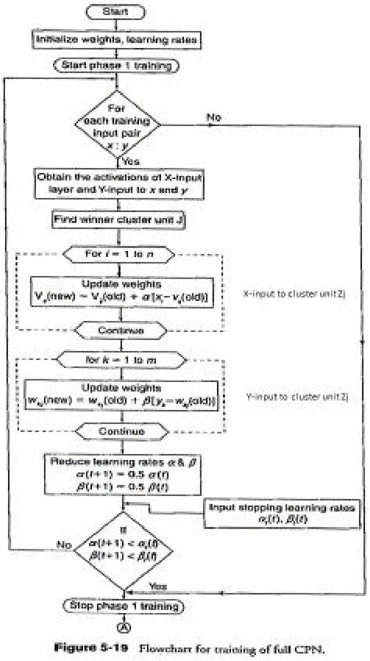

Start
Initialize weights, learning rates
Start phase 1 training
For each training input pair $x : y$
No
Yes
Obtain the activations of X-input layer and Y-input to x and y
Find winner cluster unit J
For i = 1 to n
Update weights
$V_{ij}(new) = V_{ij}(old) + \alpha[x_i - v_{ij}(old)]$
X-input to cluster unit Zj
Continue
for k = 1 to m
Update weights
$w_{kj}(new) = w_{kj}(old) + \beta[y_k - w_{kj}(old)]$
Y-input to cluster unit Zj
Continue
Reduce learning rates α & β
$\alpha(t+1) = 0.5\,\alpha(t)$
$\beta(t+1) = 0.5\,\beta(t)$
Input stopping learning rates $\alpha_i(t)$, $\beta_i(t)$
If
$\alpha(t+1) < \alpha_i(t)$
$\beta(t+1) < \beta_i(t)$
No
Yes
Stop phase 1 training
A

Figure 5-19 Flowchart for training of full CPN.

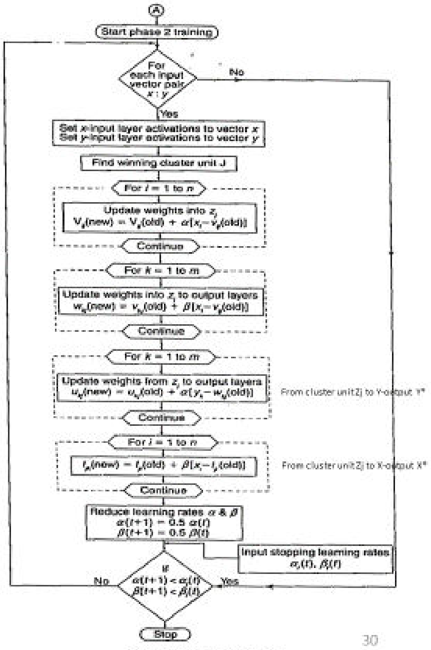
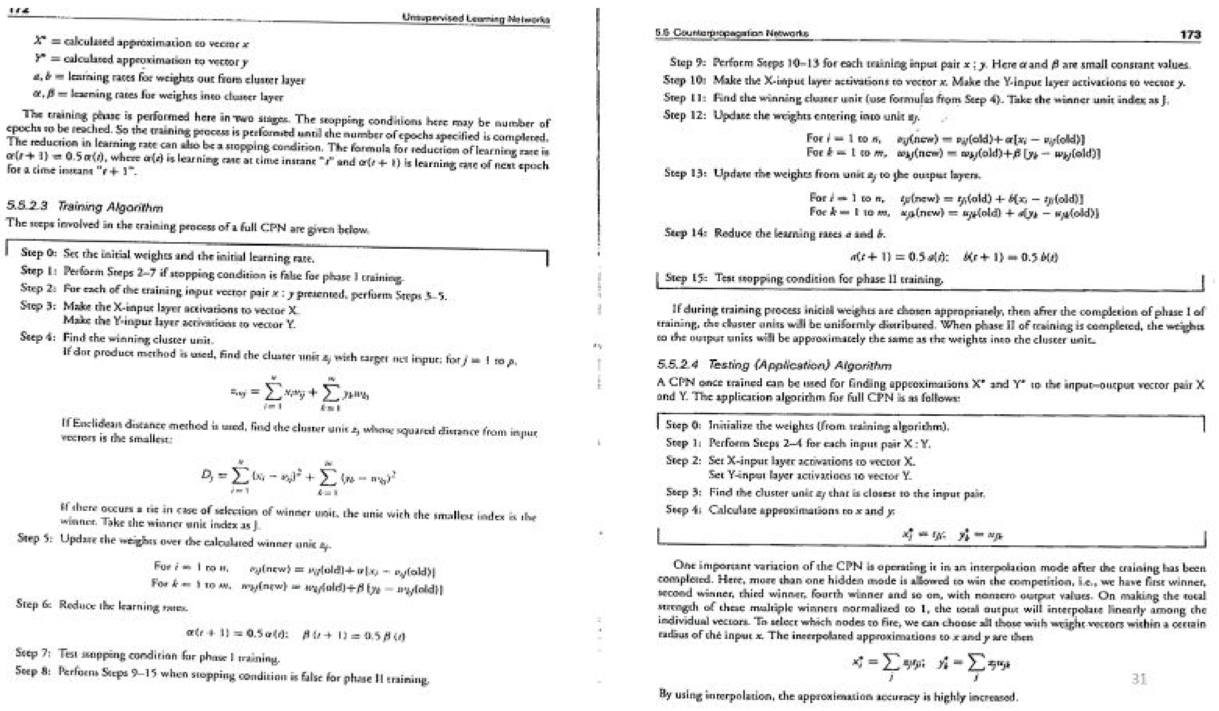

A
Start phase 2 training
For each input vector pair $x : y$
No
Yes
Set x-input layer activations to vector x
Set y-input layer activations to vector y
Find winning cluster unit J
For i = 1 to n
Update weights into z_j
$V_{ij}(new) = V_{ij}(old) + \alpha[x_i - v_{ij}(old)]$
Continue
For k = 1 to m
Update weights into z_j to output layers
$w_{kj}(new) = v_{kj}(old) + \beta[x_i - v_{ij}(old)]$
Continue
For k = 1 to m
Update weights from z_j to output layers
$u_{kj}(new) = u_{kj}(old) + \alpha[y_k - w_{kj}(old)]$
From cluster unit Zj to Y-output Y*
Continue
For i = 1 to n
$t_{ji}(new) = t_{ji}(old) + \beta[x_i - t_{ji}(old)]$
From cluster unit Zj to X-output X*
Continue
Reduce learning rates α & β
$\alpha(t+1) = 0.5\,\alpha(t)$
$\beta(t+1) = 0.5\,\beta(t)$
Input stopping learning rates $\alpha_i(t)$, $\beta_i(t)$
If
$\alpha(t+1) < \alpha_i(t)$
$\beta(t+1) < \beta_i(t)$
No
Yes
Stop

Figure 5-19 (*continued*).



172 Unsupervised Learning Networks

X^* = calculated approximation to vector x
Y^* = calculated approximation to vector y
a, b = learning rates for weights out from cluster layer
α, β = learning rates for weighes into cluster layer

The training phase is performed here in two stages. The stopping conditions here may be number of epochs to be reached. So the training process is performed until the number of epochs specified is completed. The reduction in learning rate can also be a stopping condition. The formula for reduction of learning rate is $\alpha(t+1) = 0.5\,\alpha(t)$, where $\alpha(t)$ is learning rate at time instant "t" and $\alpha(t+1)$ is learning rate of next epoch for a time instant "$t+1$".

5.5.2.3 Training Algorithm

The steps involved in the training process of a full CPN are given below.

Step 0: Set the initial weights and the initial learning rate.

Step 1: Perform Steps 2–7 if stopping condition is false for phase I training.

Step 2: For each of the training input vector pair $x : y$ presented, perform Steps 3–5.

Step 3: Make the X-input layer activations to vector X.
Make the Y-input layer activations to vector Y.

Step 4: Find the winning cluster unit.
If dot product method is used, find the cluster unit z_j with target net input: for $j = 1$ to p.

$$z_{inj} = \sum_{i=1}^{n} x_i v_{ij} + \sum_{k=1}^{m} y_k w_{kj}$$

If Euclidean distance method is used, find the cluster unit z_j whose squared distance from input vectors is the smallest:

$$D_j = \sum_{i=1}^{n} (x_i - v_{ij})^2 + \sum_{k=1}^{m} (y_k - w_{kj})^2$$

If there occurs a tie in case of selection of winner unit, the unit with the smallest index is the winner. Take the winner unit index as J.

Step 5: Update the weights over the calculated winner unit z_J.

$$\text{For } i = 1 \text{ to } n, \quad v_{iJ}(\text{new}) = v_{iJ}(\text{old}) + \alpha[x_i - v_{iJ}(\text{old})]$$
$$\text{For } k = 1 \text{ to } m, \quad w_{kJ}(\text{new}) = w_{kJ}(\text{old}) + \beta[y_k - w_{kJ}(\text{old})]$$

Step 6: Reduce the learning rates.

$$\alpha(t+1) = 0.5\,\alpha(t); \quad \beta(t+1) = 0.5\,\beta(t)$$

Step 7: Test stopping condition for phase I training.

Step 8: Perform Steps 9–15 when stopping condition is false for phase II training.

5.5 Counterpropagation Networks 173

Step 9: Perform Steps 10–13 for each training input pair $x : y$. Here α and β are small constant values.

Step 10: Make the X-input layer activations to vector x. Make the Y-input layer activations to vector y.

Step 11: Find the winning cluster unit (use formulas from Step 4). Take the winner unit index as J.

Step 12: Update the weights entering into unit z_J.

$$\text{For } i = 1 \text{ to } n, \quad v_{iJ}(\text{new}) = v_{iJ}(\text{old}) + \alpha[x_i - v_{iJ}(\text{old})]$$
$$\text{For } k = 1 \text{ to } m, \quad w_{kJ}(\text{new}) = w_{kJ}(\text{old}) + \beta[y_k - w_{kJ}(\text{old})]$$

Step 13: Update the weights from unit z_J to the output layers.

$$\text{For } i = 1 \text{ to } n, \quad t_{Ji}(\text{new}) = t_{Ji}(\text{old}) + b[x_i - t_{Ji}(\text{old})]$$
$$\text{For } k = 1 \text{ to } m, \quad u_{Jk}(\text{new}) = u_{Jk}(\text{old}) + a[y_k - u_{Jk}(\text{old})]$$

Step 14: Reduce the learning rates a and b.

$$a(t+1) = 0.5\,a(t); \quad b(t+1) = 0.5\,b(t)$$

Step 15: Test stopping condition for phase II training.

If during training process initial weights are chosen appropriately, then after the completion of phase I of training, the cluster units will be uniformly distributed. When phase II of training is completed, the weights to the output units will be approximately the same as the weights into the cluster unit.

5.5.2.4 Testing (Application) Algorithm

A CPN once trained can be used for finding approximations X* and Y* to the input–output vector pair X and Y. The application algorithm for full CPN is as follows:

Step 0: Initialize the weights (from training algorithm).

Step 1: Perform Steps 2–4 for each input pair X : Y.

Step 2: Set X-input layer activations to vector X.
Set Y-input layer activations to vector Y.

Step 3: Find the cluster unit z_J that is closest to the input pair.

Step 4: Calculate approximations to x and y:

$$x_i^* = t_{Ji}; \quad y_k^* = u_{Jk}$$

One important variation of the CPN is operating it in an interpolation mode after the training has been completed. Here, more than one hidden node is allowed to win the competition, i.e., we have first winner, second winner, third winner, fourth winner and so on, with nonzero output values. On making the total strength of these multiple winners normalized to 1, the total output will interpolate linearly among the individual vectors. To select which nodes to fire, we can choose all those with weight vectors within a certain radius of the input x. The interpolated approximations to x and y are then

$$x_i^* = \sum_j z_j t_{ji}; \quad y_k^* = \sum_j z_j u_{jk}$$

By using interpolation, the approximation accuracy is highly increased.

Adaptive resonance theory network(ART)

- Developed by Grossberg and Carpenter(1987).
- It is an unsupervised learning, based on competition, that finds categories and learns new categories if needed.
- The adaptive resonance model was developed to solve the problem of instability occurring in feed-forward systems.

Following are the two types of ART nets:

1. ART1: Designed for clustering binary vectors.
2. ART2: Designed for continuous-valued vectors.

- In both the nets input pattern can be presented in any order to the cluster unit and the cluster unit is made to learn the pattern.
- Each training pattern is presented several times. Each time the pattern is placed on different cluster unit. This decides the stability of the network.
- The ability of the network to respond to a new pattern equally at any stage of learning is called plasticity.

- The stability-plasticity can be resolved by system in which the network includes bottom-up (input-output) competitive learning combined with top-down (output-input) learning.
- The stability of the network can be increased by reducing the learning rate, but this will reduce the plasticity.
- Hence it is difficult to posses both stability and plasticity at a time.

Three types of neurons are used to build an ART network

1. Input processing neurons (F1 layer).
2. Clustering units (F2 layer).
3. Control mechanism. (controls degree of similarity of patterns placed on the same cluster).

- F1 layer consists of two portions, Input portion and Interface portion.
- Input portion processes the input it receives.
- Interface portion combines the input portion of F1 and F2 layers for comparing the similarity of the input signal with the weight vector for the cluster unit that has been selected as a unit for learning.

F1 layer input portion is denoted as F1(a) and interface portion as F1(b).

Two types of weights are used to control the degree of similarity between the units in interface portion and cluster layer.

1. Bottom-up weights: used to connect F1(b) layer and F2 layer and are represented by b_{ij} (ith F1 unit to jth F2 unit).

2. Top-down weights: are used for connection from F2 layer to F1(b) and are represented by t_{ji} (jth F2 unit to ith F1 unit).

- The competitive layer in this case is the cluster layer and the unit with largest net input is the winner selected to learn the input pattern. Activation of all other units in F2 are made zero.

Fundamental operating principle

- In ART network before any pattern is presented to training all the units in the net are set to zero.
- All the units in F2 layer are inactive.
- There is a user defined parameter called vigilance parameter, which controls the degree of similarity of the pattern assigned to the same cluster unit.

Each unit in F2 layer can be in any one of the 3 states

1. Active: Unit is ON. Activation = 1

For ART1 $d = 1$ and for ART2 $0<d<1$

2. Inactive: Unit is OFF. Activation = 0, and the unit may be available to participate in competition.

3. Inhibited: Unit is OFF. Activation = 0, and the unit is not available to participate in competition.

- In ART nets learning is of two types

Fast learning: 1) The weight updation takes place rapidly and the weights reach equilibrium in each trail.

2) The net is said to be stabilized when each pattern closes its correct cluster unit.

Slow learning: 1) Weight updation is slow and the weights do not reach equilibrium in each trail.

2) More patterns have to be presented.

3) Minimum number of calculations

Sl. No.	ART 1	ART2
1	Patterns are binary, hence weight in each cluster stabilizes with fast learning	patterns are continuous, hence after each trail the weights changes. Hence weights produced by slow learning are better than fast learning
2	Weights reach equilibrium very easily	weights does not reach equilibrium as easily as in ART1.

Fundamental algorithm

The algorithm discovers clusters of a set of pattern vectors. Following are the steps in training algorithm

Step 0: Initialize the necessary parameters.

Step 1: Perform steps 2-9 when stopping is false.

Step 2: Perform steps 3-8 for each input vector.

Step 3: F1 layer i/p processing is done.

Step 4: Perform steps 5-7 when reset condition is true.

Step 5: Find the victim unit to learn the current pattern. It is the unit of F2 with largest input.

Step 6: F1(b) units combine the inputs from F1(a) and F2.

Step 7: Test for reset condition. If reset is true, then the current victim unit is rejected, go to step 4.

(if $||x||/||s||\geq\rho$ vigilance parameter, reset is true current unit is selected for learning and weight updation is done, so goto next step 8.

if $||x||/||s||<\rho$, current unit is rejected and is not selected other time for learning. Goto step 4 to Select other unit for learning)

Step 8: Weight updation is performed.

Step 9: Test for stopping condition. (1. No significant weight change, 2. No unit is reset, 3. sufficient no.of ephocs(iterations) are reached.)

Adaptive Resonance theory 1(ART1)

- ART1 is consisting of two units, Input unit (F1), and output unit (F2). Rest is control unit for controlling the degree of similarity of patterns on same cluster unit.
- There are two interconnection path b/w F1 and F2 layers, the bottom-up(i/p to o/p) and top-down(o/p to i/p) weights, which are controlled by differential equations.
- ART1 can be implemented by analog circuits governing differential equations.
- Patterns can be presented in any order.
- ART1 network is trained using fast learning method, in which weights reach equilibrium during each learning trail.
- It performs well with binary input patterns, but it is sensitive to noise. Hence care should be taken to handle the noise.

182 Unsupervised Learning Networks

5.6.2.1 Architecture

The ART 1 network is made up of two units:

1. Computational units.
2. Supplemental units.

In this section we will discuss in detail about these two units.

Computational units

The computational unit for ART 1 consists of the following:

1. Input units (F_1 unit – both input portion and interface portion).
2. Cluster units (F_2 unit – output unit).
3. Reset control unit (controls degree of similarity of patterns placed on same cluster).

The basic architecture of ART 1 (computational unit) is shown in Figure 5-22. Here each unit present in the input portion of F_1 layer (i.e., F_1(a) layer unit) is connected to the respective unit in the interface portion of F_1 layer (i.e., F_1(b) layer unit). Reset control unit has connections from each of F_1(a) and F_1(b) units. Also, each unit in F_1(b) layer is connected through two weighted interconnection paths to each unit in F_2 layer and the reset control unit is connected to every F_2 unit. The X_i unit of F_1(b) layer is connected to Y_j unit of F_2 layer through bottom-up weights (b_{ij}) and the Y_j unit of F_2 is connected to X_i unit of F_1 through top-down weights (t_{ji}). Thus ART 1 includes a bottom-up competitive learning system combined with a top-down outstar learning system. In Figure 5-22 for simplicity only the weighted interconnections b_{ij} and t_{ji} are shown, the other units' weighted interconnections are in a similar way. The cluster layer (F_2 layer) unit is a competitive layer, where only the uninhibited node with the largest net input has nonzero activation.

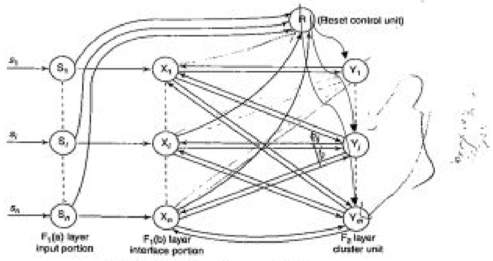


Figure 5-22 Basic architecture of ART 1

5.6 Adaptive Resonance Theory Network 183

Supplemental units

Figure 5-23 shows the supplemental unit interconnection involving two gain control units along with one reset unit. The discussion on supplemental units is important based on theoretical point of view.

Difficulty faced by computational units: It is necessary for these units to respond differently at different stages of the process, and these are not supported by any of the biological neuron to decide what to do when. The other difficulty is that the operation of the reset mechanism is not well defined for its implementation in neural systems.

The above difficulties are rectified by the introduction of two supplemental units (called as gain control units) G_1 and G_2, along with the reset control unit F. These three units receive signals from and send signals to all of the units in input layer and cluster layer. In Figure 5-23, the excitatory weighted signals are denoted by "+" and inhibitory signals are indicated by "−." Whenever any unit in designated layer is "on," a signal is sent. F_1(b) unit and F_2 unit receive signal from three sources. F_1(b) unit can receive signal from either F_1(a) unit or F_2 units or G_1 unit. In the similar way, F_2 unit receives signal from either F_1(b) unit or reset control unit R or gain control unit G_2. An F_1(b) unit or F_2 unit should receive two excitatory signals for them to be on. Both F_1(b) unit and F_2 unit can receive signals through three possible ways; this is called as two-thirds rule. The F_1(b) unit should send a signal whenever it receives input from F_1(a) and no F_2 node is active. After an F_2 node has been chosen in competition, it is necessary that only F_1(b) units whose input signal and top-down signal match remain constant. This is performed by the two gain control units G_1 and G_2, in addition with two-thirds rule. Whenever F_2 unit is on, G_1 unit is inhibited. When no F_2 unit is on, each F_1 interface unit receives a signal from G_1 unit; here, all of the units that receive a positive input signal from the input vector presented fire. In the same way, G_2 unit controls the firing of F_2 units, obeying the *two thirds rule*. The choice of parameters and initial weights may also be based on two-thirds rule. On the other hand, the vigilance matching is controlled by the reset control unit R. An excitatory signal is always sent to R when any unit in F_1(a) layer is on. The strength of the signal depends on how many F_1 (input) units are on. It should be noted that the reset control unit R also receives inhibitory signals from the F_1 interface units that are on. If sufficient number of interface units is on, then unit "F" may be prevented from firing. When unit "R" fires, it will inhibit any F_2 unit that is on. This may force the F_2 layer to choose a new winning node.

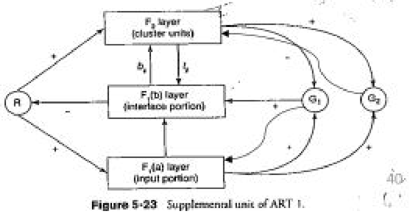

F_2 layer (cluster units)
b_{ij}
t_{ji}
F_1(b) layer (interface portion)
F_1(a) layer (input portion)
R
G_1
G_2
+
−

Figure 5-23 Supplemental unit of ART 1.

40

Supplement Unit

The computational unit faces two types of difficulties

1. It is necessary to respond differently at different stages of processing. It is not supported by even biological neuron not it is defined what to do when?
2. The reset mechanism is not well defined.

These difficulties are rectified by including two gain units G1 & G2 and one Reset unit R in the supplement unit.

➢In the supplement unit the units F1(b) and F2 receives signals from 3 sources:

F1(b) receives signals from F1(a), F2 or G1

F2 receives signals from F1(b), Reset unit R and G2.

➢F1(b) and F2 could be ON when they receives 2 excitatory signals. That means out of 3 signals 2 should be excitatory, this is called two-third rule.

➢When F1(b) receives signal from F1(a) then it will send signal to F2 and F2 gets ON.

➢When F2 is ON it makes G1 inhibit. When no unit in F2 is ON then each F1 layer receives signal from G1 and they fire.

➢Similarly G2 controls firing of F2 layer, obeying the two-third rule.

➢The choice of parameters and initial weights are based on two-third rule.

➢When any unit in F1(a) is ON then an excitatory signal is sent to R.

➢R also receives an inhibitory signal from F1(b) and the strength of this signal depends upon how many units are ON in F1(b). If it is sufficiently strong then it prevents F2 from firing.

➢If R fires then it inhibits any unit in F2 ON and forces F2 to choose a new winning unit.

5.6.2.2 Flowchart of Training Process

The flowchart for the training process of ART 1 network is shown in Figure ??. The parameters used in flowchart and training algorithm are as follows:

n = number of components in training input vector
m = maximum number of cluster units that can be formed
ρ = vigilance parameter (0 to 1)
b_{ij} = bottom-up weights (weights from X_i unit of F_1(b) layer to Y_j unit of F_2 layer)
t_{ji} = top-down weights (weights from Y_j units of F_2 layer to X_i unit of F_1(b) layer)
s = binary input vector
x = activation vector for F_1(b) layer
$\|x\|$ = norm of vector x that is defined as the sum of components of x_i (i = 1 to n)

Initially, binary input vector "s" is presented in the F_1(a) layer. Then the signals are sent to the corresponding X layer, i.e., F_1(b) layer. Each F_1(b) layer sends the activation to the F_2 layer over the weighted interconnection paths. Each F_2 layer unit then calculates the net input. The unit with the largest net input is selected as the winner and will have activation "1," the other units' activation will be 0. The winning unit is specified by its index "J." Only this winner unit can learn the current input pattern. Then the signal is send from F_2 layer to F_1(b) layer over the top-down weights (i.e., signals get multiplied with top-down weights). The X units present in the interface portion F_1(b) layer remain on, only if they receive a nonzero signal from both F_1(a) and F_2 layer units.

Now we calculate the factor $\|x\|$. The norm of vector x gives the number of components in which the top-down weight vector for the winning F_2 unit t_{Ji} and input vector s are both 1. This is called *Match*. The ratio of norm of x, $\|x\|$, to norm of s, $\|s\|$, is called *Match Ratio*, which if greater than or equal to vigilance parameter, then both the top-down and bottom-up weights have to be adjusted. This is called reset condition. That is

- If $\|x\|/\|s\| \geq \rho$, then weight updation is done. This testing condition is called reset condition.
- If $\|x\|/\|s\| < \rho$, then current unit is rejected and another unit should be chosen. The current winning cluster unit becomes inhibited, so this unit again cannot be chosen as a unit, on this particular learning trial, and the activations of the F_1 units are reset to zero.

This process is repeated until a satisfactory match is found (units get accepted) or until all the units are inhibited.

5.6.2.3 Training Algorithm

The training algorithm for ART 1 network is shown below.

Step 0: Initialize the parameters:

$$\alpha > 1 \text{ and } 0 < \rho \leq 1$$

Initialize the weights:

$$0 < b_{ij}(0) < \frac{\alpha}{\alpha - 1 + n} \text{ and } t_{ji}(0) = 1$$

Step 1: Perform Steps 2–13 when stopping condition is false.
Step 2: Perform Steps 3–12 for each of the training input.
Step 3: Set activations of all F_2 units to zero. Set the activations of F_1(a) units to input vectors.
Step 4: Calculate the norm of s:

$$\|s\| = \sum_i s_i$$

Step 5: Send input signal from F_1(a) layer to F_1(b) layer:

$$x_i = s_i$$

Step 6: For each F_2 node that is not inhibited, the following rule should hold: If $y_j \neq -1$, then $y_j = \sum_i b_{ij} x_i$
Step 7: Perform Steps 8–11 when reset is true.
Step 8: Find J for $y_J \geq y_j$ for all nodes j. If $y_J = -1$, then all the nodes are inhibited and note that this pattern cannot be clustered.
Step 9: Recalculate activation X of F_1(b):

$$x_i = s_i t_{Ji}$$

Step 10: Calculate the norm of vector x:

$$\|x\| = \sum_i x_i$$

Step 11: Test for reset condition.
If $\|x\|/\|s\| < \rho$, then inhibit node J, $y_J = -1$. Go back to step 7 again.
Else if $\|x\|/\|s\| \geq \rho$, then proceed to the next step (Step 12).
Step 12: Perform weight updation for node J (fast learning):

$$b_{iJ}(\text{new}) = \frac{\alpha x_i}{\alpha - 1 + \|x\|}$$

$$t_{Ji}(\text{new}) = x_i$$

Step 13: Test for stopping condition. The following may be the stopping conditions:
a. No change in weights.
b. No reset of units.
c. Maximum number of epochs reached.

When calculating the winner unit, if there occurs a tie, the unit with smallest index is chosen as winner. Note that in Step 3 all the inhibitions obtained from the previous learning trial are removed. When $y_J = -1$, the node is inhibited and it will be prevented from becoming the winner. The unit x_i in Step 9 will be ON only if it receives both an external signal s_i and the other signal from F_2 unit to F_1(b) unit, t_{Ji}. Note that t_{Ji} is either 0 or 1, and once it is set to 0, during learning, it can never be set back to 1 (provides stable learning method).

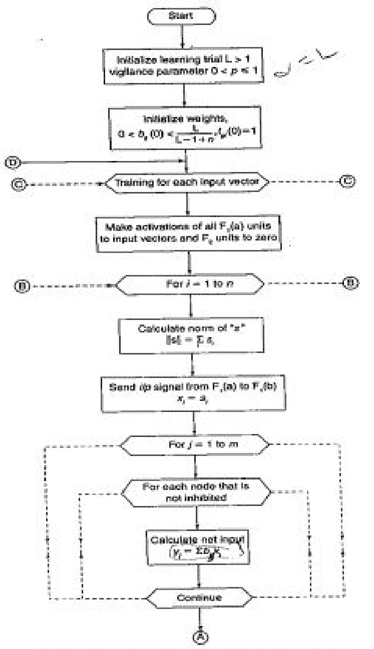

Start
Initialize learning trial L > 1
vigilance parameter 0 < ρ ≤ 1
Initialize weights,
0 < b_ij(0) < L/(L−1+n), t_ji(0)=1
D
C
Training for each input vector
C
Make activations of all F1(a) units
to input vectors and F2 units to zero
B
For i = 1 to n
B
Calculate norm of "s"
‖s‖ = Σ s_i
Send i/p signal from F1(a) to F1(b)
x_i = s_i
For j = 1 to m
For each node that is
not inhibited
Calculate net input
y_j = Σ b_ij x_i
Continue
A

Figure 5-24 Flowchart for training of ART 1 network.

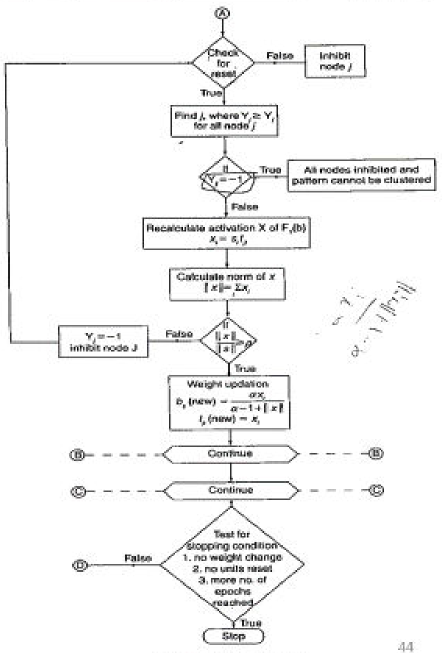


Figure 5-24 (*continued*).

- The optimal values of the initial parameters are α =2, ρ = 0.9, b_{ij} = 1/1+n and t_{ji} = 1.
- The algorithm uses fast learning and the pattern is presented for a longer period of time for weights to reach equilibrium.