PRATHYUSHA ENGINEERING COLLEGE

DEPARTMENT OF COMPUTER SCIENCE AND ENGINEERING

REGULATION R2021

II YEAR - IV SEMESTER

CS3491 – ARTIFICIAL INTELLIGENCE AND MACHINE LEARNING

CS3491 ARTIFICIAL INTELLIGENCE AND MACHINE LEARNING

COURSE OBJECTIVES:

The main objectives of this course are to:

- Study about uninformed and Heuristic search techniques.
- Learn techniques for reasoning under uncertainty
- Introduce Machine Learning and supervised learning algorithms
- Study about ensembling and unsupervised learning algorithms
- Learn the basics of deep learning using neural networks

UNIT I PROBLEM SOLVING

Introduction to AI - AI Applications - Problem solving agents – search algorithms – uninformed search strategies – Heuristic search strategies – Local search and optimization problems – adversarial search – constraint satisfaction problems (CSP)

UNIT II PROBABILISTIC REASONING

Acting under uncertainty – Bayesian inference – naïve bayes models. Probabilistic reasoning – Bayesian networks – exact inference in BN – approximate inference in BN – causal networks.

UNIT III SUPERVISED LEARNING

Introduction to machine learning – Linear Regression Models: Least squares, single & multiple variables, Bayesian linear regression, gradient descent, Linear Classification Models: Discriminant function – Probabilistic discriminative model - Logistic regression, Probabilistic generative model – Naive Bayes, Maximum margin classifier – Support vector machine, Decision Tree, Random forests

UNIT IV ENSEMBLE TECHNIQUES AND UNSUPERVISED LEARNING

Combining multiple learners: Model combination schemes, Voting, Ensemble Learning - bagging, boosting, stacking, Unsupervised learning: K-means, Instance Based Learning: KNN, Gaussian mixture models and Expectation maximization

UNIT V NEURAL NETWORKS

Perceptron - Multilayer perceptron, activation functions, network training – gradient descent optimization – stochastic gradient descent, error backpropagation, from shallow networks to deep networks –Unit saturation (aka the vanishing gradient problem) – ReLU, hyperparameter tuning, batch normalization, regularization, dropout.

ARTIFICIAL INTELLIGENCE AND MACHINE LEARNING
UNIT I
PROBLEM SOLVING

INTRODUCTION

What is artificial intelligence?

- Artificial Intelligence is the branch of computer science concerned with making computers behave like humans.
- Major AI textbooks define artificial intelligence as "the study and design of intelligent agents," where an intelligent agent is a system that perceives its environment and takes actions which maximize its chances of success.
- John McCarthy, who coined the term in 1956, defines it as "the science and engineering of making intelligent machines, especially intelligent computer programs."
- The definitions of AI according to some text books are categorized into four approaches and are summarized in the table below :

- Systems that think like humans "The exciting new effort to make computers think … machines with minds, in the full and literal sense."(Haugeland,1985)

- Systems that think rationally "The study of mental faculties through the use of computer models." (Charniak and McDermont,1985)

- Systems that act like humans The art of creating machines that performs functions that require intelligence when performed by people."(Kurzweil,1990)

- Systems that act rationally "Computational intelligence is the study of the design of intelligent agents."(Poole et al.,1998)

Systems that think like humans "The exciting new effort to make computers think … machines with minds, in the full and literal sense."(Haugeland, 1985)	**Systems that think rationally** "The study of mental faculties through the use of computer models." (Charniak and McDermont, 1985)
Systems that act like humans The art of creating machines that perform functions that require intelligence when performed by people."(Kurzweil, 1990)	**Systems that act rationally** "Computational intelligence is the study of the design of intelligent agents."(Poole et al., 1998)

<u>Four approaches of AI</u>

(a) Acting humanly : The Turing Test approach

- Test proposed by Alan Turing in 1950
- The computer is asked questions by a human interrogator.

The computer passes the test if a human interrogator, after posing some written questions, cannot tell whether the written responses come from a person or not. Programming a computer to pass, the computer need to possess the following capabilities :

- **Natural language processing** to enable it to communicate successfully in English.
- **Knowledge representation** to store what it knows or hears
- **Automated reasoning** to use the stored information to answer questions and to draw new conclusions.
- **Machine learning** to adapt to new circumstances and to detect and extrapolate patterns

To pass the complete Turing Test, the computer will need

- **Computer vision** to perceive the objects, and
- **Robotics** to manipulate objects and move about.

(b)Thinking humanly: The cognitive modeling approach

We need to get inside actual working of the human mind:

(a) through introspection – trying to capture our own thoughts as they go by;

(b) through psychological experiments

Allen Newell and Herbert Simon, who developed **GPS**, the "**General Problem Solver**" tried to trace the reasoning steps to trace the thought process of human subjects while solving the same problems.

The interdisciplinary field of **cognitive science** brings together computer models from AI and experimental techniques from psychology to try to construct precise and testable theories of the workings of the human mind.

(c) Thinking rationally: The "laws of thought approach"

The Greek philosopher Aristotle was one of the first to attempt to codify "right thinking", that is irrefutable reasoning processes. His **syllogism** provided patterns for argument structures that always yielded correct conclusions when given correct premises—

for example,

"Ram is student of III year CSE;
All students are good in III year CSE;
Ram is a good student."

These laws of thought were supposed to govern the operation of the mind; their study initiated a field called **logic.**

(d) Acting rationally: The rational agent approach

An **agent** is something that acts. A **rational agent** is one that acts so as to achieve the best outcome.

The foundations of Artificial Intelligence

The various disciplines that contributed ideas, viewpoints, and techniques to AI are given below :

Philosophy(428 B.C. – present)

Aristotle (384-322 B.C.) was the first to formulate a precise set of laws governing the rational part of the mind. He developed an informal system of syllogisms for proper reasoning, which allowed one to generate conclusions mechanically, given initial premises.

	Computer	Human Brain
Computational units	1 CPU, 10^8 gates	10^{11} neurons
Storage units	10^{10} bits RAM	10^{11} neurons
	10^{11} bits disk	10^{14} synapses
Cycle time	10^{-9} sec	10^{-3} sec
Bandwidth	10^{10} bits/sec	10^{14} bits/sec
Memory updates/sec	10^9	10^{14}

Table 1.1 A crude comparison of the raw computational resources available to computers (*circa* 2003) and brain. The computer's numbers have increased by at least by a factor of 10 every few years. The brain's numbers have not changed for the last 10, 000 years.

Brains and digital computers perform quite different tasks and have different properties. Table 1.1 shows that there are 10000 times more neurons in the typical human brain than there are gates in the CPU of a typical high-end computer. Moore's Law predicts that the CPU's gate count will equal the brain's neuron count around 2020.

Computer agents are not mere programs, but they are expected to have the following attributes also :

(a) operating under autonomous control,

(b) perceiving their environment,

(c) persisting over a prolonged time period,

(d) adapting to change.

Psycology(1879 – present)
The origin of scientific psychology are traced back to the work of German physiologist Hermann von Helmholtz(1821-1894) and his student Wilhelm Wundt(1832 – 1920)
In 1879, Wundt opened the first laboratory of experimental psychology at the University of Leipzig.
In US, the development of computer modeling led to the creation of the field of **cognitive science**.
The field can be said to have started at the workshop in September 1956 at MIT.

Computer engineering (1940-present)
For artificial intelligence to succeed, we need two things: intelligence and an artifact. The computer has been the artifact of choice.
AI also owes a debt to the software side of computer science, which has supplied the operating systems, programming languages, and tools needed to write modern programs

Control theory and Cybernetics (1948-present)
Ktesibios of Alexandria (c. 250 B.C.) built the first self-controlling machine: a water clock with a regulator that kept the flow of water running through it at a constant, predictable pace.

Modern control theory, especially the branch known as stochastic optimal control, has as its goal the design of systems that maximize an **objective function** over time.

Linguistics (1957-present)
Modem linguistics and AI, then, were "born" at about the same time, and grew up together, intersecting in a hybrid field called **computational linguistics** or **natural language processing.**

History of Artificial Intelligence

The gestation of artificial intelligence (1943-1955)
There were a number of early examples of work that can be characterized as AI, but it was Alan Turing who first articulated a complete vision of AI in his 1950 article "Computing Machinery and Intelligence." Therein, he introduced the Turing test, machine learning, genetic algorithms, and reinforcement learning.

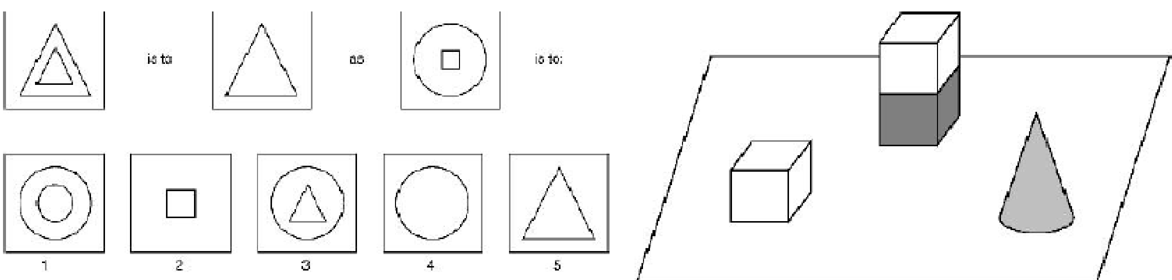
The birth of artificial intelligence (1956)
McCarthy convinced Minsky, Claude Shannon, and Nathaniel Rochester to help him bring together U.S. researchers interested in automata theory, neural nets, and the study of intelligence. They organized a two-month workshop at Dartmouth in the summer of 1956. Perhaps the longest-lasting thing to come out of the workshop was an agreement to adopt McCarthy's new name for the field: **artificial intelligence.**

Early enthusiasm, great expectations (1952-1969)
The early years of AI were full of successes-in a limited way.
General Problem Solver (GPS) was a computer program created in 1957 by Herbert Simon and Allen Newell to build a universal problem solver machine. The order in which the program considered subgoals and possible actions was similar to that in which humans approached the same problems. Thus, GPS was probably the first program to embody the "thinking humanly" approach.

At IBM, Nathaniel Rochester and his colleagues produced some of the first AI programs. Herbert Gelernter (1959) constructed the **Geometry Theorem Prover**, which was able to prove theorems that many students of mathematics would find quite tricky.

Lisp was invented by John McCarthy in 1958 while he was at the Massachusetts Institute of Technology (MIT). In 1963, McCarthy started the AI lab at Stanford.

Tom Evans's ANALOGY program (1968) solved geometric analogy problems that appear in IQ tests, such as the one in Figure 1.1



A dose of reality (1966-1973)

From the beginning, AI researchers were not shy about making predictions of their coming successes. The following statement by Herbert Simon in 1957 is often quoted:

"It is not my aim to surprise or shock you-but the simplest way I can summarize is to say that there are now in the world machines **that think, that learn and that create**. Moreover, their ability to do these things is going to increase rapidly until-in a visible future-the range of problems they can handle will be **coextensive with the range** to which the human mind has been applied.

Knowledge-based systems: The key to power? (1969-1979)

Dendral was an influential pioneer project in artificial intelligence (AI) of the 1960s, and the computer software **expert system** that it produced. Its primary aim was to help organic chemists in identifying unknown organic molecules, by analyzing their mass spectra and using knowledge of chemistry. It was done at Stanford University by Edward Feigenbaum, Bruce Buchanan, Joshua Lederberg, and Carl Djerassi.

AI becomes an industry (1980-present)

In 1981, the Japanese announced the "Fifth Generation" project, a 10-year plan to build intelligent computers running **Prolog**. Overall, the AI industry boomed from a few million dollars in 1980 to billions of dollars in 1988.

The return of neural networks (1986-present)

Psychologists including David Rumelhart and Geoff Hinton continued the study of neural-net models of memory.

AI becomes a science (1987-present)

In recent years, approaches based on **hidden Markov models** (HMMs) have come to dominate the area. **Speech technology and the related field of handwritten character recognition** are already making the transition to widespread industrial and consumer applications.

The **Bayesian network** formalism was invented to allow efficient representation of, and rigorous reasoning with, uncertain knowledge.

The emergence of intelligent agents (1995-present)

One of the most important environments for intelligent agents is the Internet.

APPLICATIONS OF ARTIFICIAL INTELLIGENCE

- Autonomous planning and scheduling:
    - A hundred million miles from Earth, NASA's Remote Agent program became the first on-board autonomous planning program to control the scheduling of operations for a spacecraft (Jonsson et al., 2000).

    - Remote Agent generated plans from high-level goals specified from the ground, and it monitored the operation of the spacecraft as the plans were executed-detecting, diagnosing, and recovering from problems as they occurred.

- Game playing:
    - IBM's Deep Blue became the first computer program to defeat the world champion in a chess match when it bested Garry Kasparov by a score of 3.5 to 2.5 in an exhibition match (Goodman and Keene, 1997).

- Autonomous control:
    - The ALVINN computer vision system was trained to steer a car to keep it following a lane.
    - It was placed in CMU's NAVLAB computer-controlled minivan and used to navigate across the United States-for 2850 miles it was in control of steering the vehicle 98% of the time

- Diagnosis:
    - Medical diagnosis programs based on probabilistic analysis have been able to perform at the level of an expert physician in several areas of medicine.

- Logistics Planning:
    - During the Persian Gulf crisis of 1991, U.S. forces deployed a Dynamic Analysis and Replanning Tool, DART (Cross and Walker, 1994), to do automated logistics planning and scheduling for transportation.
    - This involved up to 50,000 vehicles, cargo, and people at a time, and had to account for starting points, destinations, routes, and conflict resolution among all parameters.
    - The AI planning techniques allowed a plan to be generated in hours that would have taken weeks with older methods.
    - The Defense Advanced Research Project Agency (DARPA) stated that this single application more than paid back DARPA's 30-year investment in AI.

- Robotics:
    - Many surgeons now use robot assistants in microsurgery. HipNav (DiGioia et al., 1996) is a system that uses computer vision techniques to create a three dimensional model of a patient's internal anatomy and then uses robotic control to guide the insertion of a hip replacement prosthesis.

- Language understanding and problem solving:
    - PROVERB (Littman et al., 1999) is a computer program that solves crossword puzzles better than most humans, using constraints on possible word fillers, a large database of past puzzles, and a variety of information sources including dictionaries and online databases such as a list of movies and the actors that appear in them.

Problem-solving agents

- A Problem solving agent is a goal-based agent.
- It decides what to do by finding sequence of actions that lead to desirable states.
- The agent can adopt a goal and aim at satisfying it.
- For example where our agent is in the city of Arad, which is in Romania.
- The agent has to adopt a goal of getting to Bucharest.
- Goal formulation, based on the current situation and the agent's performance measure, is the first step in problem solving.

- The agent's task is to find out which sequence of actions will get to a goal state.
- Problem formulation is the process of deciding what actions and states to consider given a goal.

- Example: Route finding problem
- On holiday in Romania : currently in Arad.
- Flight leaves tomorrow from Bucharest
- Formulate goal: be in Bucharest

Formulate problem:
- states: various cities actions: drive between cities
- Find solution: sequence of cities, e.g., Arad, Sibiu, Fagaras, Bucharest

Goal formulation and problem formulation

- A problem is defined by four items:
- initial state e.g., "at Arad"
- Successor function S(x) = set of action-state pairs
- e.g., S(Arad) = {[Arad -> Zerind;Zerind],….}
- goal test, can be
- explicit, e.g., x = at Bucharest"
- path cost (additive)
- e.g., sum of distances, number of actions executed, etc.
- c(x; a; y) is the step cost, assumed to be >= 0
- A solution is a sequence of actions leading from the initial state to a goal state.

SEARCH :

- An agent with several immediate options of unknown value can decide what to do by examining different possible sequences of actions that leads to the states of known value, and then choosing the best sequence.
- The process of looking for sequences actions from the current state to reach the goal state is called search.
- The search algorithm takes a problem as input and returns a solution in the form of action sequence.
- Once a solution is found, the execution phase consists of carrying out the recommended action.
- The following shows a simple "formulate, search, execute" design for the agent.
- Once solution has been executed, the agent will formulate a new goal.
- It first formulates a goal and a problem, searches for a sequence of actions that would solve a problem, and executes the actions one at a time.

function SIMPLE-PROBLEM-SOLVING-AGENT(percept) returns an action
inputs : *percept*, a percept
static: *seq*, an action sequence, initially empty
        state, some description of the current world state
        goal, a goal, initially null
        problem, a problem formulation
state ← UPDATE-STATE(state, percept)
if seq is empty then do
        goal ← FORMULATE-GOAL(state)
        problem ← FORMULATE-PROBLEM(state, goal)

```
        seq ← SEARCH( problem)
        action ← FIRST(seq);
        seq ← REST(seq)
return action
```

- The agent design assumes the Environment is

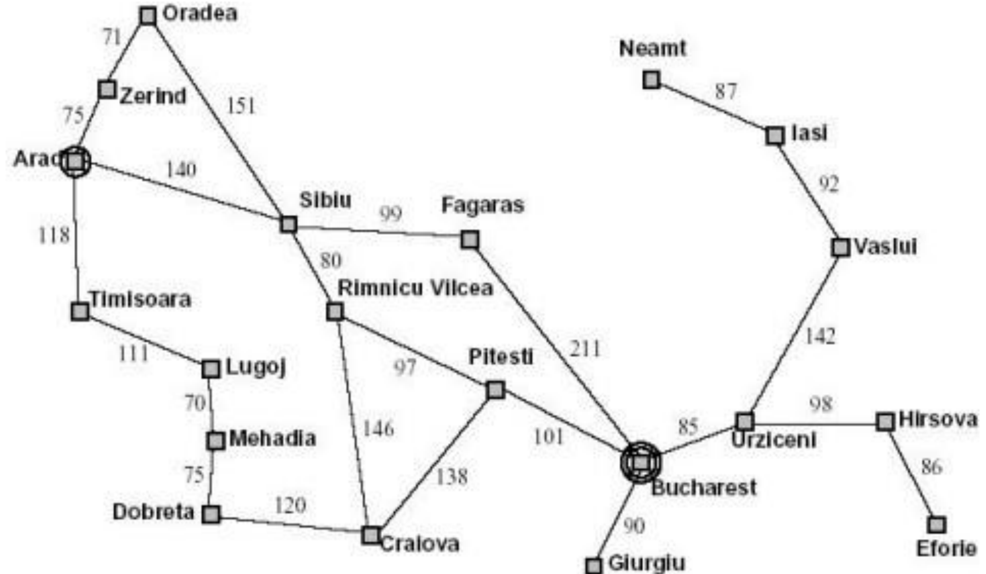
- Static: The entire process carried out without paying attention to changes that might be occurring in the environment.
- Observable : The initial state is known and the agent's sensor detects all aspects that are relevant to the choice of action
- Discrete : With respect to the state of the environment and percepts and actions so that alternate courses of action can be taken
- Deterministic: The next state of the environment is completely determined by the current state and the actions executed by the agent. Solutions to the problem are single sequence of actions
- An agent carries out its plan with eye closed. This is called an open loop system because ignoring the percepts breaks the loop between the agent and the environment.

Well-defined problems and solutions

- A problem can be formally defined by four components:
- The initial state that the agent starts in . The initial state for our agent of example problem is described by *In(Arad)*
- A Successor Function returns the possible actions available to the agent.
- Given a state x,SUCCESSOR-FN(x) returns a set of {action,successor} ordered pairs where each action is one of the legal actions in state x,and each successor is a state that can be reached from x by applying the action.
- For example, from the state In(Arad),the successor function for the Romania problem would return

{ [Go(Sibiu),In(Sibiu)],[Go(Timisoara),In(Timisoara)],[Go(Zerind),In(Zerind)] }

- State Space: The set of all states reachable from the initial state. The state space forms a graph in which the nodes are states and the arcs between nodes are actions.
- A path in the state space is a sequence of states connected by a sequence of actions.
- The goal test determines whether the given state is a goal state.
- A path cost function assigns numeric cost to each action.
- For the Romania problem the cost of path might be its length in kilometers.
- The step cost of taking action a to go from state x to state y is denoted by c(x,a,y). It is assumed that the step costs are non negative.
- A solution to the problem is a path from the initial state to a goal state.
- An optimal solution has the lowest path cost among all solutions.



A simplified Road Map of part of Romania

UNINFORMED SEARCH STRATGES

- Uninformed Search Strategies have no additional information about states beyond that provided in the problem definition.
- Strategies that know whether one non goal state is "more promising" than another are called Informed search or heuristic search strategies.

There are five uninformed search strategies as given below.
o Breadth-first search
o Uniform-cost search
o Depth-first search
o Depth-limited search
o Iterative deepening search
o Bidirectional Search

Breadth-first search

- Breadth-first search is a simple strategy in which the root node is expanded first, then all successors of the root node are expanded next, then their successors, and so on.
- In general, all the nodes are expanded at a given depth in the search tree before any nodes at the next level are expanded.
- Breath-first-search is implemented by calling TREE-SEARCH with an empty fringe that is a first-in-first-out(FIFO) queue, assuring that the nodes that are visited first will be expanded first.
- In otherwards, calling TREE-SEARCH (problem,FIFO-QVEVE()) results in breadth-first search.
- The FIFO queue puts all newly generated successors at the end of the queue, which means that Shallow nodes are expanded before deeper nodes.

```
function BREADTH-FIRST-SEARCH(problem) returns a solution, or failure
  node ← a node with STATE = problem.INITIAL-STATE, PATH-COST = 0
  if problem.GOAL-TEST(node.STATE) then return SOLUTION(node)
  frontier ← a FIFO queue with node as the only element
  explored ← an empty set
  loop do
     if EMPTY?(frontier) then return failure
     node ← POP(frontier)  /* chooses the shallowest node in frontier */
     add node.STATE to explored
     for each action in problem.ACTIONS(node.STATE) do
        child ← CHILD-NODE(problem, node, action)
        if child.STATE is not in explored or frontier then
           if problem.GOAL-TEST(child.STATE) then return SOLUTION(child)
           frontier ← INSERT(child, frontier)
```

Figure 3.11 Breadth-first search on a graph.

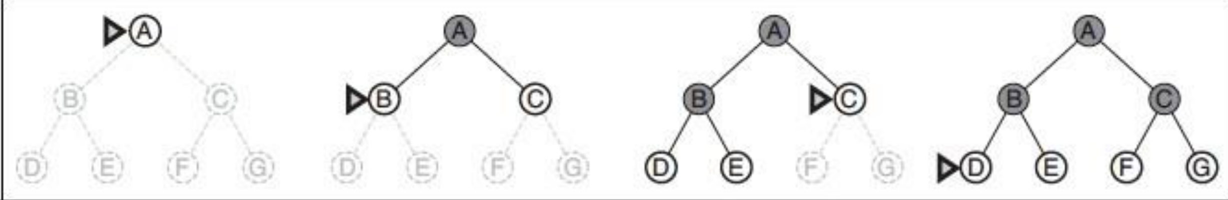


Figure 3.12 Breadth-first search on a simple binary tree. At each stage, the node to be expanded next is indicated by a marker.

Depth	Nodes	Time	Memory
2	110	.11 milliseconds	107 kilobytes
4	11,110	11 milliseconds	10.6 megabytes
6	10^6	1.1 seconds	1 gigabyte
8	10^8	2 minutes	103 gigabytes
10	10^{10}	3 hours	10 terabytes
12	10^{12}	13 days	1 petabyte
14	10^{14}	3.5 years	99 petabytes
16	10^{16}	350 years	10 exabytes

Figure 3.13 Time and memory requirements for breadth-first search. The numbers shown assume branching factor $b = 10$; 1 million nodes/second; 1000 bytes/node.

UNIFORM-COST SEARCH

- Instead of expanding the shallowest node, uniform-cost search expands the node n with the lowest path cost.
- Uniform-cost search does not care about the number of steps a path has,but only about their total cost.

Sibiu to Bucharest.

- The successors of Sibiu are Rimnicu Vilcea and Fagaras, with costs 80 and 99, respectively.

- The least-cost node, Rimnicu Vilcea, is expanded next, adding Pitesti with cost 80 + 97=177. The least-cost node is now Fagaras, so it is expanded, adding Bucharest with cost 99+211=310.
- Now a goal node has been generated, but uniform-cost search keeps going, choosing Pitesti for expansion and adding a second path to Bucharest with cost 80+97+101= 278.
- Now the algorithm checks to see if this new path is better than the old one; it is, so the old one is discarded. Bucharest, now with g-cost 278, is selected for expansion and the solution is returned.

```
function UNIFORM-COST-SEARCH(problem) returns a solution, or failure
  node ← a node with STATE = problem.INITIAL-STATE, PATH-COST = 0
  frontier ← a priority queue ordered by PATH-COST, with node as the only element
  explored ← an empty set
  loop do
     if EMPTY?(frontier) then return failure
     node ← POP(frontier)  /* chooses the lowest-cost node in frontier */
     if problem.GOAL-TEST(node.STATE) then return SOLUTION(node)
     add node.STATE to explored
     for each action in problem.ACTIONS(node.STATE) do
        child ← CHILD-NODE(problem, node, action)
        if child.STATE is not in explored or frontier then
           frontier ← INSERT(child, frontier)
        else if child.STATE is in frontier with higher PATH-COST then
           replace that frontier node with child
```

Figure 3.14 Uniform-cost search on a graph. The algorithm is identical to the general graph search algorithm in Figure 3.7, except for the use of a priority queue and the addition of an extra check in case a shorter path to a frontier state is discovered. The data structure for *frontier* needs to support efficient membership testing, so it should combine the capabilities of a priority queue and a hash table.

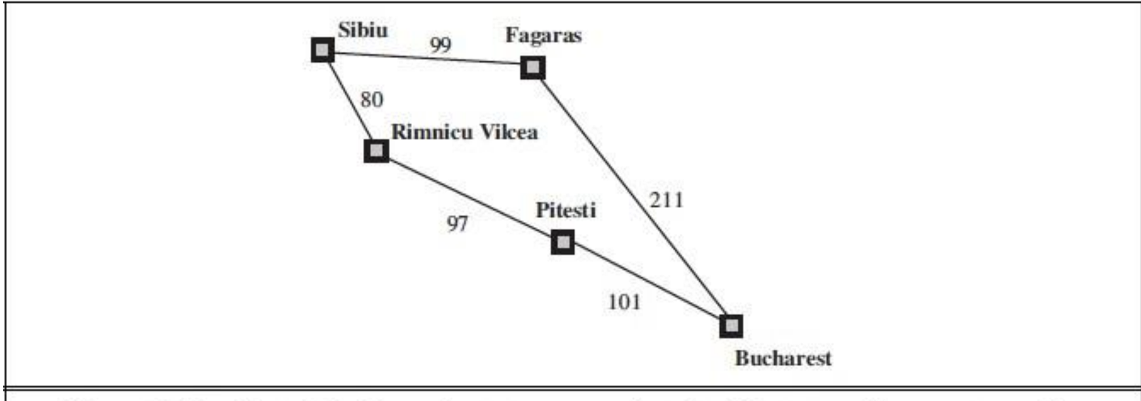


Figure 3.15 Part of the Romania state space, selected to illustrate uniform-cost search.

Depth-first search

- Depth-first search always expands DEPTH-FIRST the deepest node in the current frontier of the search tree.
- Depth-first-search always expands the deepest node in the current fringe of the search tree.
- The search proceeds immediately to the deepest level of the search tree, where the nodes have no successors.

- As those nodes are expanded, they are dropped from the fringe, so then the search "backs up" to the next shallowest node that still has unexplored successors.
- This strategy can be implemented by TREE-SEARCH with a last-in-first out (LIFO) queue, also known as a stack.
- Depth-first-search has very modest memory requirements.
- It needs to store only a single path from the root to a leaf node, along with the remaining unexpanded sibling nodes for each node on the path.
- Once the node has been expanded, it can be removed from the memory, as soon as its descendants have been fully explored.
- For a state space with a branching factor b and maximum depth m, depth first-search requires storage of only bm + 1 nodes.

Drawback of Depth-first-search

- The drawback of depth-first-search is that it can make a wrong choice and get stuck going down very long(or even infinite) path when a different choice would lead to solution near the root of the search tree.
- For example, depth-first-search will explore the entire left subtree even if node C is a goal node

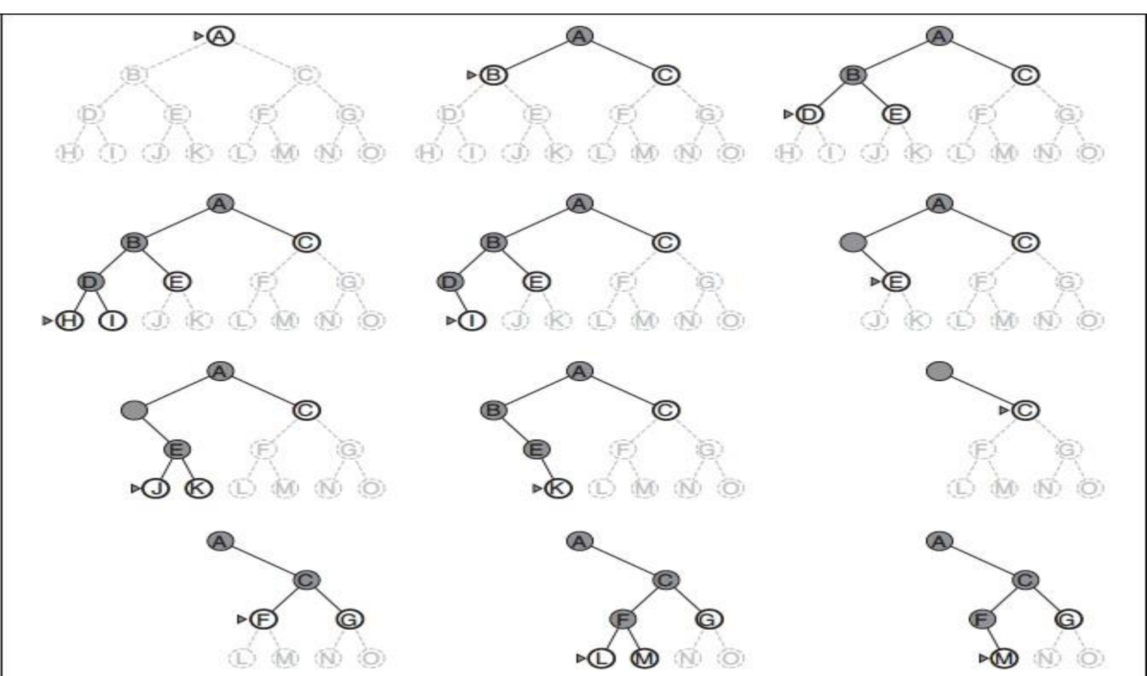


Figure 3.16 Depth-first search on a binary tree. The unexplored region is shown in light gray. Explored nodes with no descendants in the frontier are removed from memory. Nodes at depth 3 have no successors and M is the only goal node.

- A variant of depth-first search called backtracking BACKTRACKING search uses still less memory.
- Backtracking search facilitates yet another memory-saving (and time-saving) trick: the idea of generating a successor by modifying the current state description directly rather than copying it first.

DEPTH-LIMITED-SEARCH

- The problem of unbounded trees can be alleviated by supplying depth-first-search with a pre-determined depth limit l.
- That is,nodes at depth l are treated as if they have no successors.
- This approach is called depth-limited-search.

- The depth limit solves the infinite path problem.
- Depth limited search will be nonoptimal if we choose l > d. Its time complexity is $O(b^l)$ and its space complexity is O(bl).
- Depth-first-search can be viewed as a special case of depth-limited search with l = oo
- Sometimes, depth limits can be based on knowledge of the problem.
- For, example, on the map of Romania there are 20 cities.
- Therefore, we know that if there is a solution, it must be of length 19 at the longest, So l = 10 is a possible choice.
- However,it can be shown that any city can be reached from any other city in at most 9 steps.
- This number known as the diameter of the state space, gives us a better depth limit.
- Depth-limited-search can be implemented as a simple modification to the general tree search algorithm or to the recursive depth-first-search algorithm.
- The pseudocode for recursive depth-limited-search is shown.
- It can be noted that the above algorithm can terminate with two kinds of failure : the standard failure value indicates no solution; the cut off value indicates no solution within the depth limit.
- Depth-limited search = depth-first search with depth limit l, returns cut off if any path is cut off by depth limit

```
function DEPTH-LIMITED-SEARCH(problem, limit) returns a solution, or failure/cutoff
   return RECURSIVE-DLS(MAKE-NODE(problem.INITIAL-STATE), problem, limit)

function RECURSIVE-DLS(node, problem, limit) returns a solution, or failure/cutoff
   if problem.GOAL-TEST(node.STATE) then return SOLUTION(node)
   else if limit = 0 then return cutoff
   else
       cutoff_occurred? ← false
       for each action in problem.ACTIONS(node.STATE) do
           child ← CHILD-NODE(problem, node, action)
           result ← RECURSIVE-DLS(child, problem, limit − 1)
           if result = cutoff then cutoff_occurred? ← true
           else if result ≠ failure then return result
       if cutoff_occurred? then return cutoff else return failure
```

Figure 3.17 A recursive implementation of depth-limited tree search.

- The depth-limited search can terminate with two kinds of failure: the standard failure value indicates no solution; the cutoff value indicates no solution within the depth limit.

Iterative deepening depth-first search

- Iterative deepening search (or iterative deepening depth-first search) is a general strategy, DEEPENING SEARCH often used in combination with depth-first tree search, that finds the best depth limit.
- It does this by gradually increasing the limit—first 0, then 1, then 2, and so on—until a goal is found.
- This will occur when the depth limit reaches d, the depth of the shallowest goal node.
- Iterative deepening combines the benefits of depth-first and breadth-first search.

- Like depth-first search, its memory requirements are modest: O(bd) to be precise.

```
function ITERATIVE-DEEPENING-SEARCH(problem) returns a solution, or failure
  for depth = 0 to ∞ do
    result ← DEPTH-LIMITED-SEARCH(problem, depth)
    if result ≠ cutoff then return result
```

Figure 3.18 The iterative deepening search algorithm, which repeatedly applies depth-limited search with increasing limits. It terminates when a solution is found or if the depth-limited search returns *failure*, meaning that no solution exists.

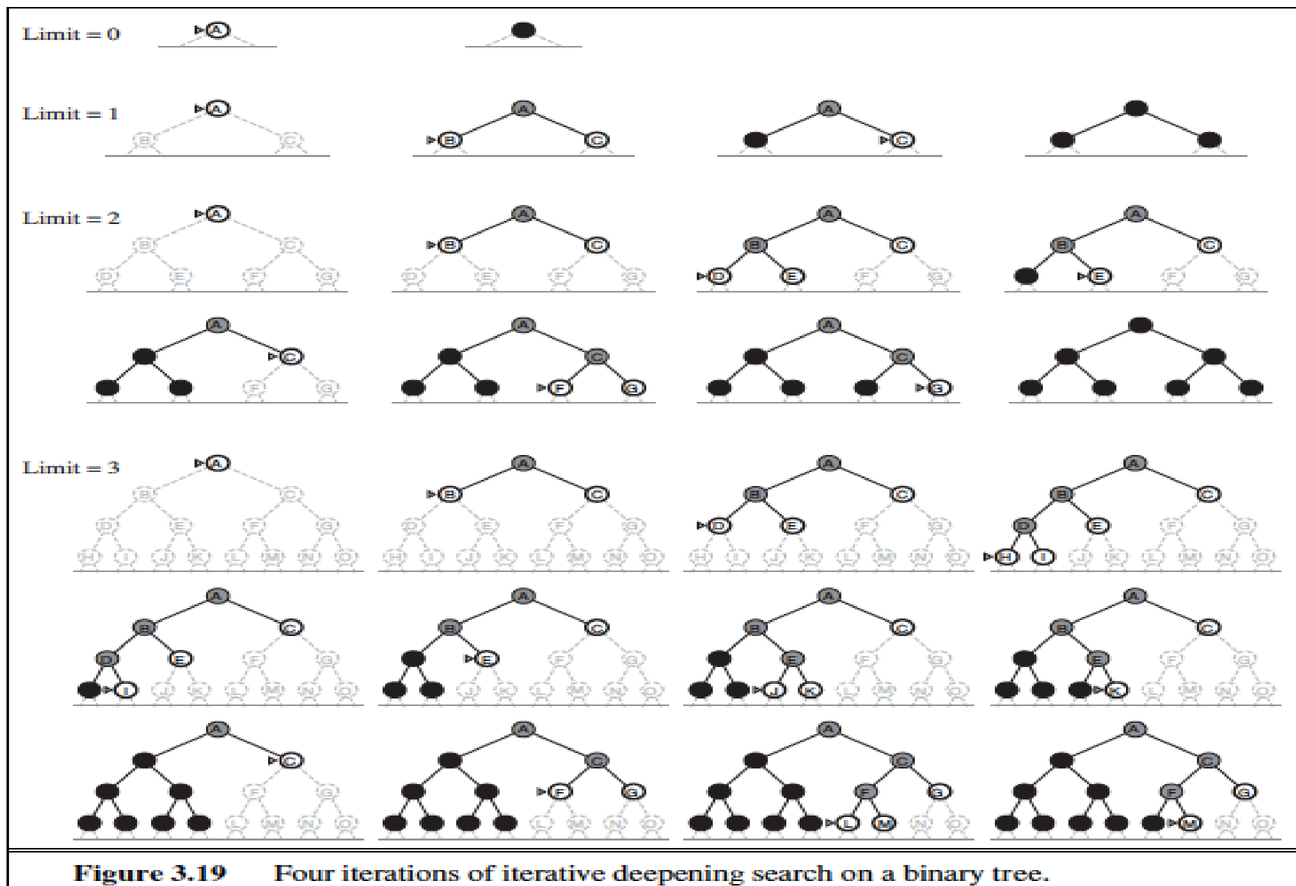


Figure 3.19 Four iterations of iterative deepening search on a binary tree.

Bidirectional Search

- The idea behind bidirectional search is to run two simultaneous searches
- one forward from the initial state and
- other backward from the goal,
- It stops when the two searches meet in the middle.
- The motivation is that $b^{d/2} + b^{d/2}$ much less than b^d

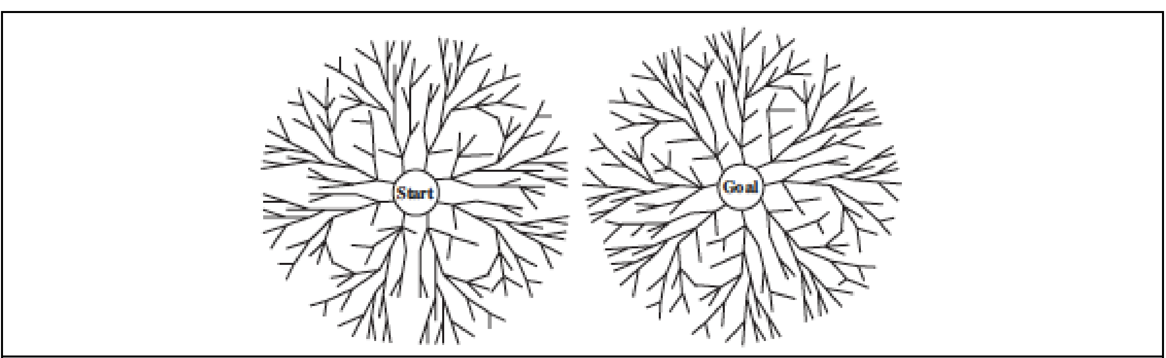


Figure 3.20 A schematic view of a bidirectional search that is about to succeed when a branch from the start node meets a branch from the goal node.

Figure 3.21 compares search strategies in terms of the four evaluation criteria set forth in Section 3.3.2. This comparison is for tree-search versions. For graph searches, the main differences are that depth-first search is complete for finite state spaces and that the space and time complexities are bounded by the size of the state space.

Criterion	Breadth-First	Uniform-Cost	Depth-First	Depth-Limited	Iterative Deepening	Bidirectional (if applicable)
Complete?	Yes^a	Yes^{a,b}	No	No	Yes^a	Yes^{a,d}
Time	$O(b^d)$	$O(b^{1+\lfloor C^*/\epsilon \rfloor})$	$O(b^m)$	$O(b^\ell)$	$O(b^d)$	$O(b^{d/2})$
Space	$O(b^d)$	$O(b^{1+\lfloor C^*/\epsilon \rfloor})$	$O(bm)$	$O(b\ell)$	$O(bd)$	$O(b^{d/2})$
Optimal?	Yes^c	Yes	No	No	Yes^c	Yes^{c,d}

Figure 3.21 Evaluation of tree-search strategies. b is the branching factor; d is the depth of the shallowest solution; m is the maximum depth of the search tree; l is the depth limit. Superscript caveats are as follows: ^{a} complete if b is finite; ^{b} complete if step costs $\geq \epsilon$ for positive ϵ; ^{c} optimal if step costs are all identical; ^{d} if both directions use breadth-first search.

INFORMED (HEURISTIC) SEARCH STRATEGIES

GREEDY BEST – FIRST SEARCH

- Greedy best-first search tries to expand the node that is closest to the goal, on the grounds that this is likely to lead to a solution quickly

• Thus, the evaluation function is f(n) = h(n)

• E.g. in minimizing road distances a heuristic lower bound for distances of cities is their straight-line distance

• Greedy search ignores the cost of the path that has already been traversed to reach n

• Therefore, the solution given is not necessarily optimal

• If repeating states are not detected, greedy best-first search may oscillate forever between two promising states

- Because greedy best-first search can start down an infinite path and never return to try other possibilities, it is incomplete
- Because of its greediness the search makes choices that can lead to a dead end; then one backs up in the search tree to the deepest unexpanded node
- Greedy best-first search resembles depth-first search in the way it prefers to follow a single path all the way to the goal, but will back up when it hits a dead end
- The worst-case time and space complexity is $O(b^m)$
- The quality of the heuristic function determines the practical usability of greedy search
-

GREEDY : Best First Search

Combining the two is to follow a single path at a time, but switch paths whenever some competing path looks more promising than the current one.

- Best first search is an instance of graph search algorithm in which a node is selected for expansion based on evaluation function f (n).
- The node which is the lowest evaluation is selected for the explanation because the evaluation measures distance to the goal.
- Best first search can be implemented within general search frame work via a priority queue, a data structure that will maintain the fringe in ascending order of f values.
- This search algorithm serves as **combination of depth first and breadth first search algorithm**. Best first search algorithm is often referred greedy algorithm this is because they quickly attack the most desirable path as soon as its heuristic weight becomes the most desirable.
- For both depth-first and breadth-first search, which node in the search tree will be considered next only depends on the structure of the tree.
- • The rationale in best-first search is to **expand those paths next that seem the most "promising"**. Making this vague idea of what may be promising precise means defining heuristics.
- Heuristics by means of a heuristic function h that is used to estimate the "distance" of the current node n to a goal node:

  h(n) = estimated cost from node n to a goal node

The definition of h is highly application-dependent. In the route-planning domain, for instance, we could use the straight-line distance to the goal location. For the eight-puzzle, we might use the number of misplaced tiles.

Concept:

Step 1: Traverse the root node

Step 2: Traverse any neighbour of the root node, that is maintaining a least distance from the root node and insert them in ascending order into the queue.

Step 3: Traverse any neighbour of neighbour of the root node, that is maintaining a least distance from the root node and insert them in ascending order into the queue

Step 4: This process will continue until we are getting the goal node

- OPEN: nodes that have been generated, but have not examined. This is organized as a priority queue.
- CLOSED: nodes that have already been examined. Whenever a new node is generated, check whether it has been generated before.

Algorithm

1. OPEN = {initial state}.
2. Loop until a goal is found or there are no nodes left in OPEN:

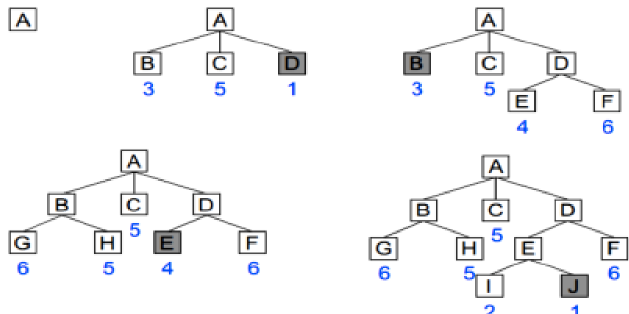
   − Pick the best node in OPEN

   − Generate its successors

   − For each successor:

   new→ evaluate it, add it to OPEN, record its parent

   generated before → change parent, update successors



Greedy search:

h(n) = cost of the cheapest path from node n to agoal state.

Neither optimal nor complete

• Uniform-cost search:

g(n) = cost of the cheapest path from the initial state to node n.

Optimal and complete, but very inefficient

Advantage:

- It is more efficient than that of BFS and DFS.

- Time complexity of Best first search is much less than Breadth first search.
- The Best first search allows us to switch between paths by gaining the benefits of both breadth first and depth first search. Because, depth first is good because a solution can be found without computing all nodes and Breadth first search is good because it does not get trapped in deadends.

Disadvantages:

Sometimes, it covers more distance than our consideration.

A* SEARCH

- A* is a cornerstone name of many AI systems and has been used since it was developed in 1968 by PeterHart; Nils Nilsson and Bertram Raphael.
- It is the combination of Dijkstra's algorithm and Best first search. It can be used to solve many kinds of problems.
- A* search finds the shortest path through a search space to goal state using heuristic function. This technique finds minimal cost solutions and is directed to a goal state called A* search.
- In A*, the * is written for optimality purpose.
- The A* algorithm combines features of uniform-cost search and pure heuristic search to efficiently compute optimal solutions.

The A* algorithm also **finds the lowest cost path between the start and goal state**, where changing from one state to another requires some cost.

A* requires **heuristic function to evaluate the cost of path** that passes through the particular state. This algorithm is complete if the branching factor is finite and every action has fixed cost. It can be defined by following formula.

$$f(n) = g(n) + h(n)$$

Where

$g(n)$: The actual cost path from the *start state to the current state*.

$h(n)$: The estimated cost path from the *current state to goal state*.

$f(n)$: The cost path from the start state to the goal state.

If $h(n)$ is an underestimate of the path costs from node n to a goal node, then $f(p)$ is an underestimate of a path cost of going from a start node to a goal node via p.

Function $f^*(n)$:At any node n,it is the actual cost of an optimal path from node s to node n plus the cost of an optimal path from node n to a goal node.

$f^*(n)=g^*(n)+h^*(n)$

Where $g^*(n)$=cost of the optimal path in the search tree from s to n;

$h^*(n)$=cost of the optimal path in the search tree from n to a goal node;

For the implementation of A* algorithm we will use two arrays namely OPEN and CLOSE.

OPEN: An array which contains the nodes that has been generated but has not been yet examined.

CLOSE: An array which contains the nodes that have been examined.

Algorithm:

1.Create a *search graph* G,consisting solely of the start node s. Puts on a list called OPEN.

2 .Create a list called CLOSED that is initially empty.

3.LOOP: if OPEN is empty, exit with failure.

4. Select the first node on OPEN, remove it from OPEN and put it on CLOSED. Call this node n.

5.If n is a goal node,exit successfully with the solution obtained by tracing a path along the pointers from n to s in G.

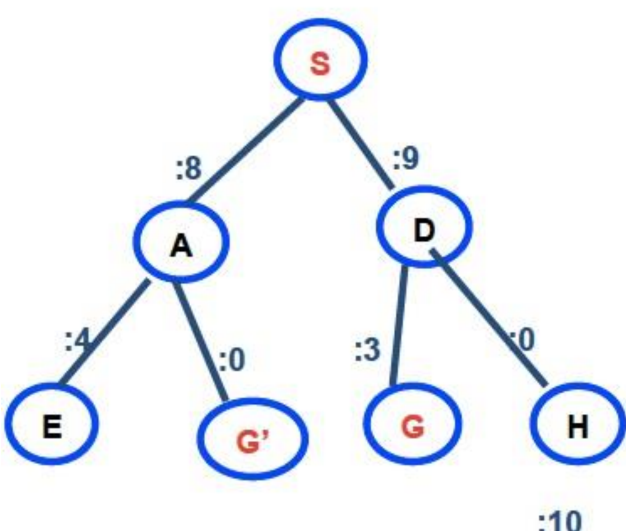
6.Expand node n, generating the set, M, of its successors and install them as successors of n in G.

7.Establish a pointer to n from those members of M that were not already in G(I .e, not already on either OPEN or CLOSED). Add these members of M to OPEN.For each member of M that was already on OPEN or CLOSED,decide whether or not to redirect its pointer to n. For each member of M already on CLOSED, decide for each of its descendents in G whether or not to redirect its pointer.

8.Reorder the list OPEN, either according to some scheme or some heuristic merit.

9.Goto LOOP

Example :



Path cost for S-D-G

$f(S) = g(S) + h(S)$

$= 0 + 10 \rightarrow$ 10

$f(D) = (0+3) + 9 \rightarrow 12$

$f(G) = (0+3+3) + 0 \rightarrow 6$

Total path cost = $f(S)+f(D)+f(G) \rightarrow 28$

Path cost for S-A-G'

$f(S) = 0 + 10 \rightarrow$ 10

$f(A) = (0+2) + 8 \rightarrow 10$

$f(G') = (0+2+2) + 0 \rightarrow 4$

Total path cost = $f(S)+f(A)+f(G') \rightarrow 24$

* Path S-A-G is chosen = Lowest cost

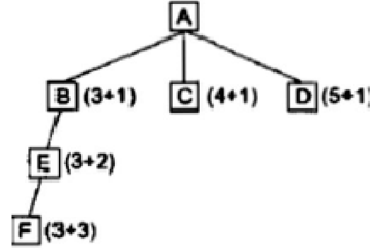


Fig. 3.4 *h' Underestimates h*

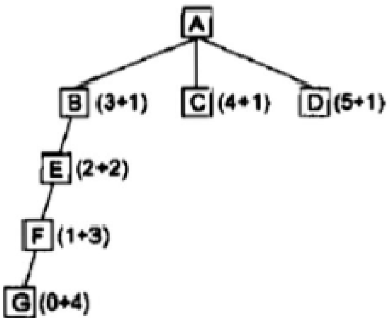


Fig. 3.5 *h' Overestimates h*

Implementation:

The implementation of A* algorithm is 8-puzzle game.

Advantages:

- It is complete and optimal.
- It is the best one from other techniques.
- It is used to solve very complex problems.
- It is optimally efficient, i.e. there is no other optimal algorithm guaranteed to expand fewer nodes than A*.

Disadvantages:

- This algorithm is complete if the branching factor is finite and every action has fixed cost.
- The speed execution of A* search is highly dependant on the accuracy of the heuristic algorithm that is used to compute **h** (n).
- It has complexity problems.

Admissibility

The property that A* always finds an optimal path, if one exists, and that the first path found to a goal is optimal is called the **admissibility** of A*. Admissibility means that, even when the search space is infinite, if solutions exist, a solution will be found and the first path found will be an optimal solution - a lowest-cost path from a start node to a goal node.

Example heuristic functions:

• Let h1(n) be the straight-line distance to the goal location. This is an admissible heuristic, because no solution path will ever be shorter than the straight-line connection.

Memory-bounded heuristic search

- The simplest way to reduce memory requirements for A∗ is to adapt the idea of iterative
- deepening to the heuristic search context, resulting in the iterative-deepening A* (IDA∗) algorithm.

- The main difference between IDA∗ and standard iterative deepening is that the cutoff used is the f-cost (g+h) rather than the depth; at each iteration, the cutoff value is the smallest f-cost of any node that exceeded the cutoff on the previous iteration.
- Recursive best-first search (RBFS) is a simple recursive algorithm that attempts to mimic the operation of standard best-first search, but using only linear space.
- RBFS is somewhat more efficient than IDA∗, but still suffers from excessive node regeneration.
- Like A∗ tree search, RBFS is an optimal algorithm if the heuristic function h(n) is admissible.
- IDA∗ and RBFS suffer from using too little memory.
- Between iterations, IDA∗ retains only a single number: the current f-cost limit.
- Two algorithms that do this memory reduce are MA∗ (memory-bounded A∗) and SMA∗ (simplified MA∗). SMA∗
- SMA∗ proceeds just like A∗, expanding the best leaf until memory is full.
- At this point, it cannot add a new node to the search tree without dropping an old one.
- SMA∗ always drops the worst leaf node—the one with the highest f-value.
- Like RBFS, SMA∗ then backs up the value of the forgotten node to its parent.
- To avoid selecting the same node for deletion and expansion, SMA∗ expands the newest best leaf and deletes the oldest worst leaf.

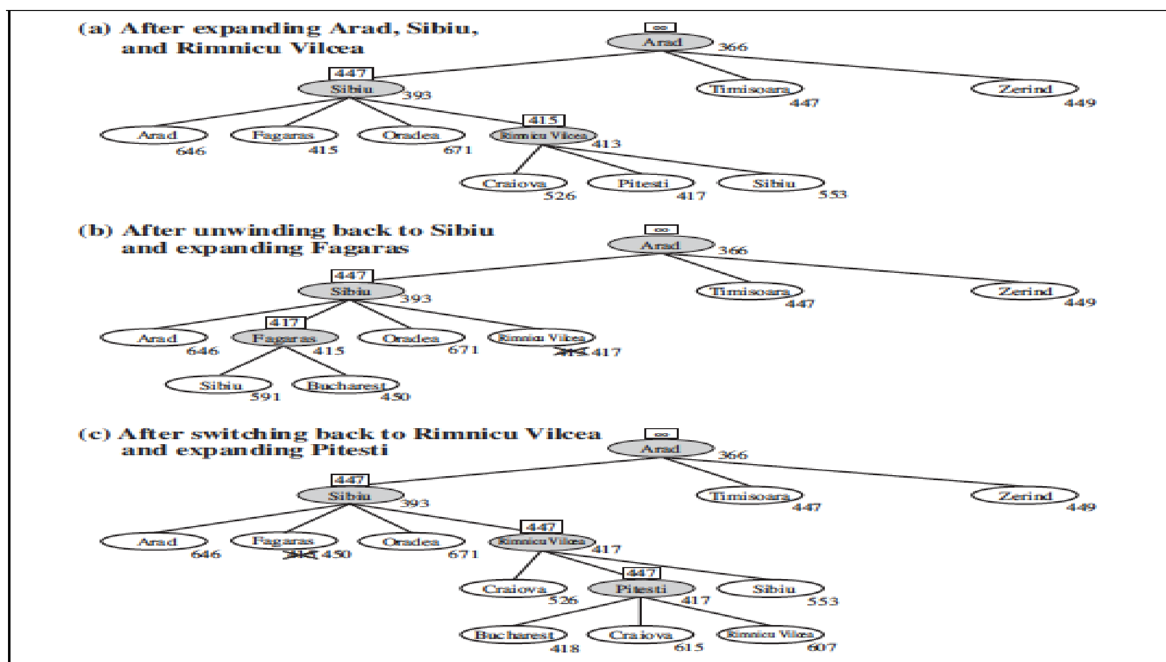


Figure 3.27 Stages in an RBFS search for the shortest route to Bucharest. The f-limit value for each recursive call is shown on top of each current node, and every node is labeled with its f-cost. (a) The path via Rimnicu Vilcea is followed until the current best leaf (Pitesti) has a value that is worse than the best alternative path (Fagaras). (b) The recursion unwinds and the best leaf value of the forgotten subtree (417) is backed up to Rimnicu Vilcea; then Fagaras is expanded, revealing a best leaf value of 450. (c) The recursion unwinds and the best leaf value of the forgotten subtree (450) is backed up to Fagaras; then Rimnicu Vilcea is expanded. This time, because the best alternative path (through Timisoara) costs at least 447, the expansion continues to Bucharest.

LOCAL SEARCH ALGORITHMS AND OPTIMIZATION PROBLEMS

- Local search algorithms operate using a single current node and generally move only to neighbors of that node.

Advantages:
(1) they use very little
memory—usually a constant amount; and

(2) they can often find reasonable solutions in large or infinite state spaces

- In addition to finding goals, local search algorithms are useful for solving pure optimization problems, in which the aim is to find the best state according to an objective function.
- State-space landscape: A landscape has both “location” and “elevation”.
- If elevation corresponds to cost, then the aim is to find the lowest valley—a global minimum;
- if elevation corresponds to an objective function, then the aim is to find the highest peak—a global maximum.
- A complete local search algorithm always finds a goal if one exists; an optimal algorithm always finds a global minimum/maximum.

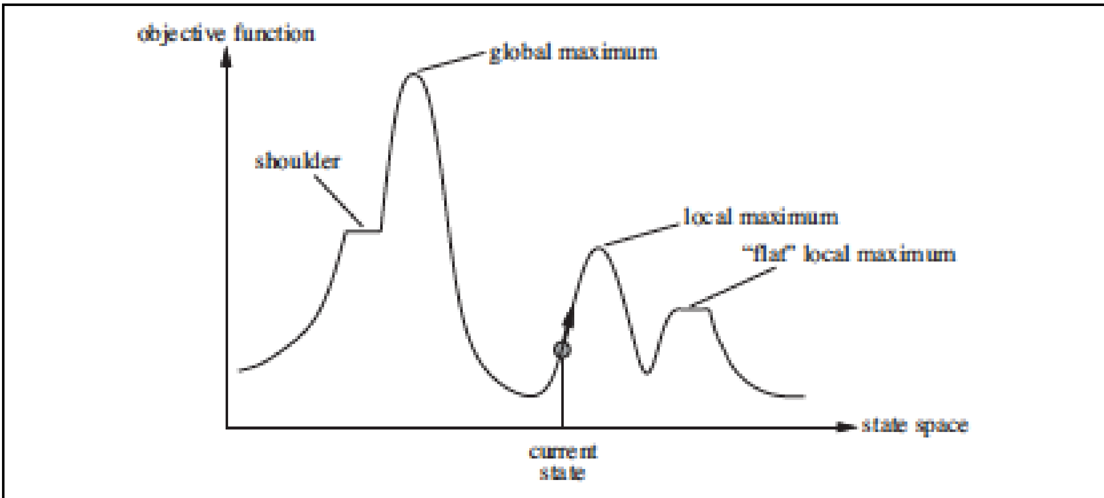


Figure 4.1 A one-dimensional state-space landscape in which elevation corresponds to the objective function. The aim is to find the global maximum. Hill-climbing search modifies the current state to try to improve it, as shown by the arrow. The various topographic features are defined in the text.

Hill-climbing search:

```
function HILL-CLIMBING(problem) returns a state that is a local maximum

  current ← MAKE-NODE(problem.INITIAL-STATE)
  loop do
     neighbor ← a highest-valued successor of current
     if neighbor.VALUE ≤ current.VALUE then return current.STATE
     current ← neighbor
```

Figure 4.2 The hill-climbing search algorithm, which is the most basic local search technique. At each step the current node is replaced by the best neighbor; in this version, that means the neighbor with the highest VALUE, but if a heuristic cost estimate h is used, we would find the neighbor with the lowest h.

- The hill-climbing search algorithm (steepest-ascent version) is shown in Figure 4.2.
- It is simply a loop that continually moves in the direction of increasing value that is, uphill.
- It terminates when it reaches a "peak" where no neighbor has a higher value.
- The algorithm does not maintain a search tree, so the data structure for the current node need only record the state and the value of the objective function.
- To illustrate hill climbing, we will use the 8-queens problem
- Local search algorithms typically use a complete-state formulation, where each state has 8 queens on the board, one per column.
- The successors of a state are all possible states generated by moving a single queen to another square in the same column.
- The heuristic cost function h is the number of pairs of queens that are attacking each other, either directly or indirectly.
- The global minimum of this function is zero, which occurs only at perfect solutions.
- Figure 4.3(a) shows a state with h=17.
- The figure also shows the values of all its successors, with the best successors having h=12.
- Hill-climbing algorithms typically choose randomly among the set of best successors if there is more than one.
    - Local maxima: a local maximum is a peak that is higher than each of its neighboring states but lower than the global maximum.
    - Ridges: Ridges result in a sequence of local maxima that is very difficult for greedy algorithms to navigate.
    - Plateaux: a plateau is a flat area of the state-space landscape.

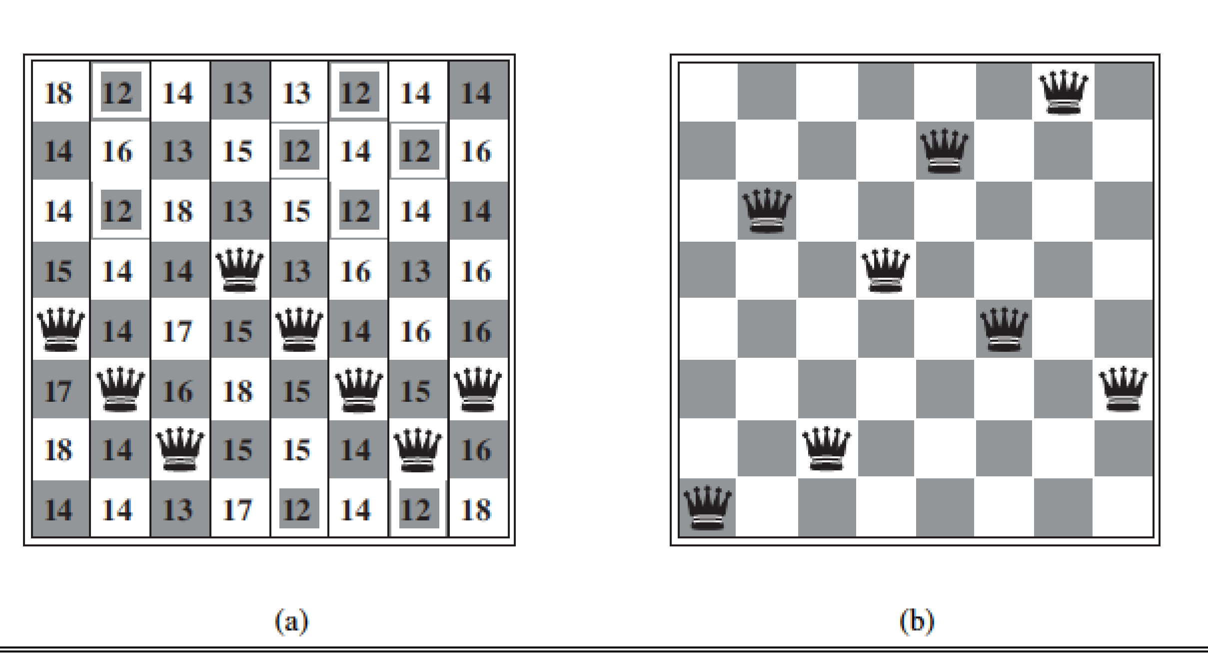


Figure 4.3 (a) An 8-queens state with heuristic cost estimate $h = 17$, showing the value of h for each possible successor obtained by moving a queen within its column. The best moves are marked. (b) A local minimum in the 8-queens state space; the state has $h = 1$ but every successor has a higher cost.

- Stochastic hill climbing chooses at random from among the uphill moves; the probability of selection can vary with the steepness of the uphill move.
- First-choice hill climbing implements stochastic hill climbing by generating successors randomly until one is generated that is better than the current state.

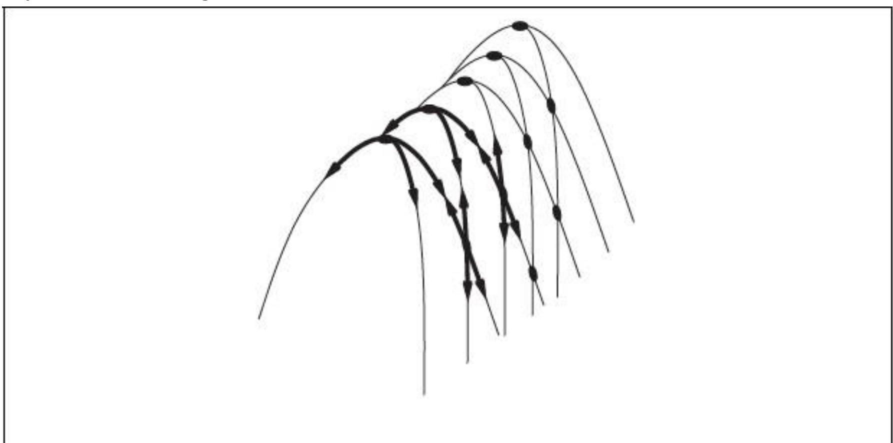


Figure 4.4 Illustration of why ridges cause difficulties for hill climbing. The grid of states (dark circles) is superimposed on a ridge rising from left to right, creating a sequence of local maxima that are not directly connected to each other. From each local maximum, all the available actions point downhill.

Simulated annealing

- Simulated annealing - To combine hill climbing with a random walk - is such an algorithm.
- In metallurgy, annealing is the process used to temper or harden metals and glass by heating them to a high temperature and then gradually cooling them, thus allowing the material to reach a low energy crystalline state.

- To explain simulated annealing, we switch our point of view from hill climbing to gradient descent.
- Figure 2.11 shows simulated annealing algorithm.
- It is quite similar to hill climbing.
- Instead of picking the best move, however,it picks the random move.
- If the move improves the situation,it is always accepted.
- Otherwise,the algorithm accepts the move with some probability less than 1.
- The probability decreases exponentially with the "badness" of the move – the amount E by which the evaluation is worsened.
- Simulated annealing was first used extensively to solve VLSI layout problems in the early 1980s. It has been applied widely to factory scheduling and other large-scale optimization tasks

```
function SIMULATED-ANNEALING(problem, schedule) returns a solution state
  inputs: problem, a problem
          schedule, a mapping from time to "temperature"

  current ← MAKE-NODE(problem.INITIAL-STATE)
  for t = 1 to ∞ do
     T ← schedule(t)
     if T = 0 then return current
     next ← a randomly selected successor of current
     ΔE ← next.VALUE − current.VALUE
     if ΔE > 0 then current ← next
     else current ← next only with probability e^(ΔE/T)
```

Figure 4.5 The simulated annealing algorithm, a version of stochastic hill climbing where some downhill moves are allowed. Downhill moves are accepted readily early in the annealing schedule and then less often as time goes on. The *schedule* input determines the value of the temperature T as a function of time.

Local beam search

- The local beam search algorithm3 keeps track of k states rather than just one.
- It begins with k randomly generated states.
- At each step, all the successors of all k states are generated.
- If any one is a goal, the algorithm halts.
- Otherwise, it selects the k best successors from the complete list and repeats.
- In a local beam search, useful information is passed among the parallel search threads.
- In its simplest form, local beam search can suffer from a lack of diversity among the k states where the search is expensive.
- A variant called stochastic beam search, analogous to stochastic hill climbing, helps alleviate this problem.
- Instead of choosing the best k from the pool of candidate successors, stochastic beam search chooses k successors at random, with the probability of choosing a given successor being an increasing function of its value.

Genetic algorithms

- A genetic algorithm (or GA) is a variant of stochastic beam search in which successor states
- are generated by combining two parent states rather than by modifying a single state.
- The analogy to natural selection is the same as in stochastic beam search, except that now we are dealing with sexual rather than asexual reproduction.

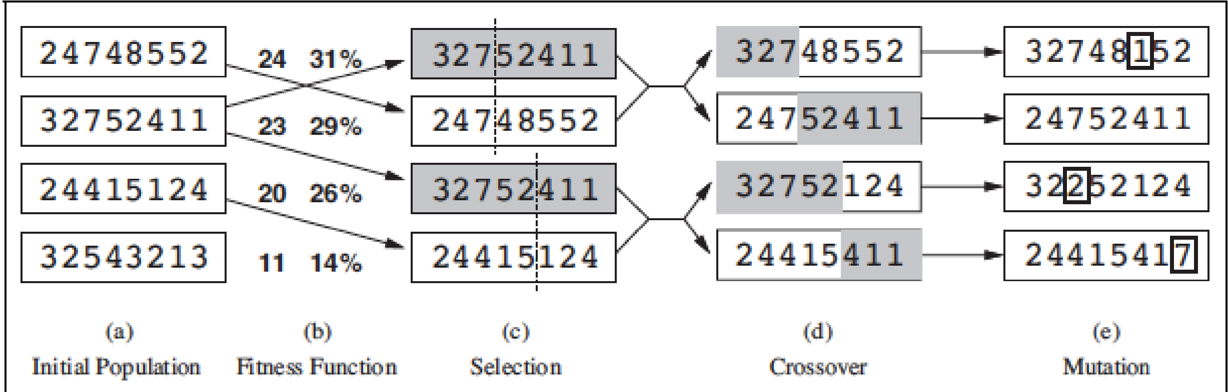


Figure 4.6 The genetic algorithm, illustrated for digit strings representing 8-queens states. The initial population in (a) is ranked by the fitness function in (b), resulting in pairs for mating in (c). They produce offspring in (d), which are subject to mutation in (e).

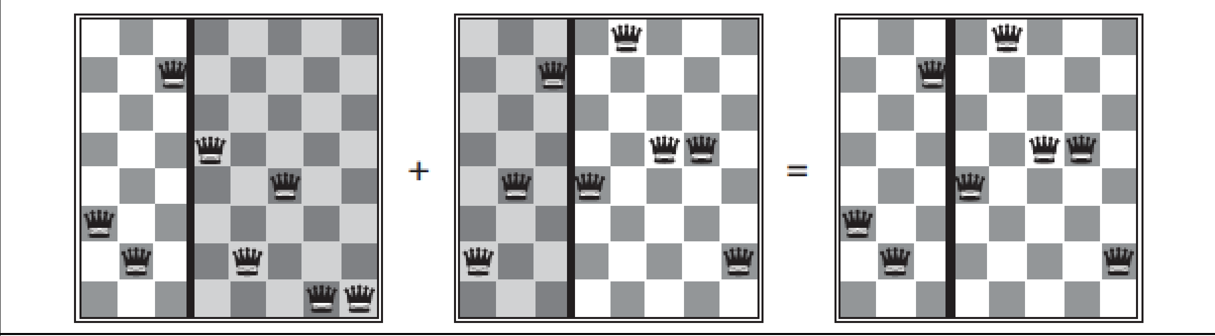


Figure 4.7 The 8-queens states corresponding to the first two parents in Figure 4.6(c) and the first offspring in Figure 4.6(d). The shaded columns are lost in the crossover step and the unshaded columns are retained.

- A Genetic algorithm(or GA) is a variant of stochastic beam search in which successor states are generated by combining two parent states,rather than by modifying a single state.
- Like beam search, Gas begin with a set of k randomly generated states, called the population. Each state, or individual,is represented as a string over a finite alphabet – most commonly, a string of 0s and 1s.
- For example, an 8 8-quuens state must specify the positions of 8 queens,each in acolumn of 8 squares,and so requires 8 x log2 8 = 24 bits.
- Figure 2.12 shows a population of four 8-digit strings representing 8-queen states.
- The production of the next generation of states is shown in Figure 2.12(b) to (e).
- In (b) each state is rated by the evaluation function or the fitness function.
- In (c),a random choice of two pairs is selected for reproduction, in accordance with the probabilities in (b).
- Figure 2.13 describes the algorithm that implements all these steps.

```
function GENETIC-ALGORITHM(population, FITNESS-FN) returns an individual
  inputs: population, a set of individuals
          FITNESS-FN, a function that measures the fitness of an individual

  repeat
     new_population ← empty set
     for i = 1 to SIZE(population) do
        x ← RANDOM-SELECTION(population, FITNESS-FN)
        y ← RANDOM-SELECTION(population, FITNESS-FN)
        child ← REPRODUCE(x, y)
        if (small random probability) then child ← MUTATE(child)
        add child to new_population
     population ← new_population
  until some individual is fit enough, or enough time has elapsed
  return the best individual in population, according to FITNESS-FN
```

```
function REPRODUCE(x, y) returns an individual
  inputs: x, y, parent individuals

  n ← LENGTH(x); c ← random number from 1 to n
  return APPEND(SUBSTRING(x, 1, c), SUBSTRING(y, c + 1, n))
```

Figure 4.8 A genetic algorithm. The algorithm is the same as the one diagrammed in Figure 4.6, with one variation: in this more popular version, each mating of two parents produces only one offspring, not two.

CONSTRAINT SATISFACTION PROBLEMS

A Constraint Satisfaction Problem is characterized by:

□ _a *set of variables* {x1, x2, .., xn},

□ _for each variable xi a *domain* Di with the possible values for that variable, and

□ _a set of *constraints*, i.e. relations, that are assumed to hold between the values of the variables. [These relations can be given intentionally, i.e. as a formula, or extensionally, i.e. as a set, or procedurally, i.e. with an appropriate generating or recognising function.

- We will only consider constraints involving one or two variables.
- The constraint satisfaction problem is to find, for each i from 1 to n, a value in Di for xi so that all constraints are satisfied.
- A CS problem is usually represented as an undirected graph, called *Constraint Graph* where the nodes are the variables and the edges are the binary constraints.
- Unary constraints can be disposed of by just redefining the domains to contain only the values that satisfy all the unary constraints.
- Higher order constraints are represented by hyperarcs.
- In the following we restrict our attention to the case of unary and binary constraints.
- Consistency Based Algorithms use information from the constraints to reduce the search space as early in the search as it is possible

□ _This problem requires a lot of reasoning.

□ _Time complexity of the problem is more as concerned to the other problems.

□ _This problem can also be solved by the evolutionary approach and mutation operations.

□ _This problem is dependent upon some constraints which are necessary part of the problem.

□ _Various complex problems can also be solved by this technique.

Example

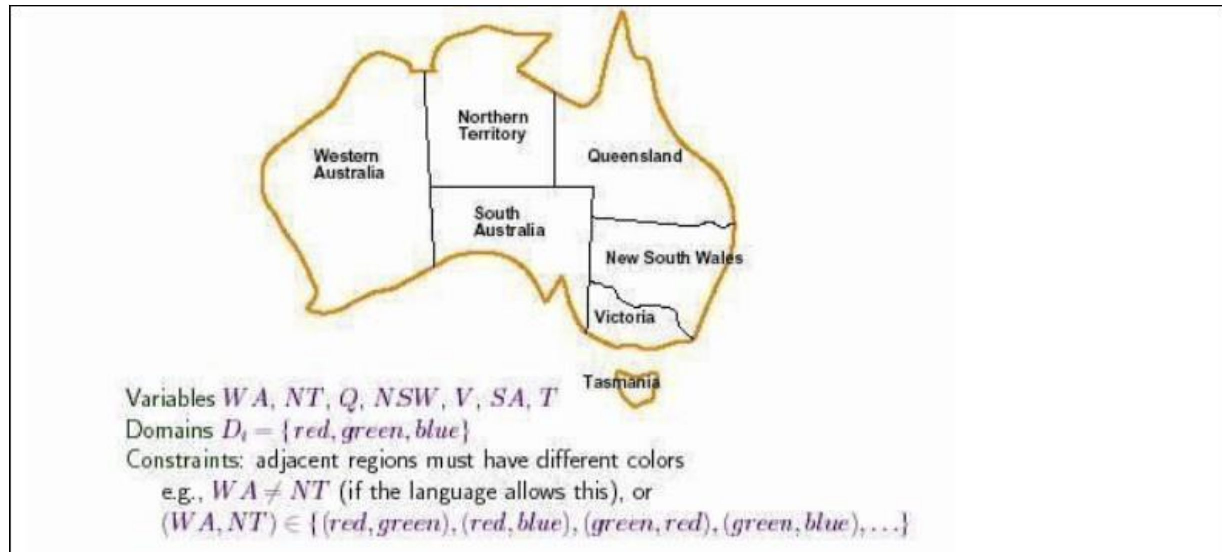


Figure 2.15 (a) Principle states and territories of Australia. Coloring this map can be viewed as a constraint satisfaction problem. The goal is to assign colors to each region so that no neighboring regions have the same color.

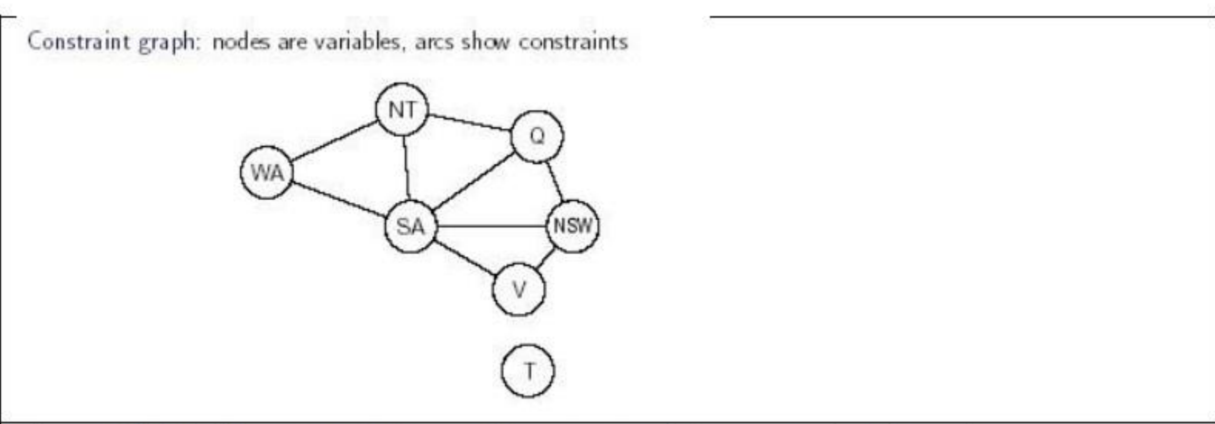


Figure 2.15 (b) The map coloring problem represented as a constraint graph.

	WA	NT	Q	NSW	V	SA	T
Initial domains	R G B	R G B	R G B	R G B	R G B	R G B	R G B
After *WA=red*	(R)	G B	R G B	R G B	R G B	G B	R G B
After *Q=green*	(R)	B	(G)	R B	R G B	B	R G B
After *V=blue*	(R)	B	(G)	R	(B)		R G B

Figure 5.6 The progress of a map-coloring search with forward checking. $WA = red$ is assigned first; then forward checking deletes *red* from the domains of the neighboring variables NT and SA. After $Q = green$, *green* is deleted from the domains of NT, SA, and NSW. After $V = blue$, *blue* is deleted from the domains of NSW and SA, leaving SA with no legal values.

- **CSP can be viewed as a standard search problem as follows :**

- Initial state : the empty assignment {},in which all variables are unassigned.
- Successor function : a value can be assigned to any unassigned variable,provided that it does not conflict with previously assigned variables.
- Goal test : the current assignment is complete.
- Path cost : a constant cost(E.g.,1) for every step.

- Every solution must be a complete assignment and therefore appears at depth n if there are n variables.
- Depth first search algorithms are popular for CSPs

Varieties of constraints :

(i) Unary constraints involve a single variable.
Example: SA # green
(ii) Binary constraints involve pairs of variables.
Example: SA # WA
(iii) Higher order constraints involve 3 or more variables.
Example: cryptarithmetic puzzles.
(iv) Absolute constraints are the constraints, which rules out a potential solution when they are violated
(v) Preference constraints are the constraints indicating which solutions are preferred

CONSTRAINT PROPAGATION

In regular state-space search, an algorithm can do only one thing: search.
In CSPs there is a choice: an algorithm can search (choose a new variable assignment from several possibilities) or do a specific type of **inference** called **constraint propagation**: using the constraints to reduce the number of legal values for a variable, which in turn can reduce the legal values for another variable, and so on.

Constraint propagation may be intertwined with search, or it may be done as a preprocessing step, before search starts. Sometimes this preprocessing can solve the whole problem, so no search is required at all.
The key idea is **local consistency**. If we treat each variable as a node in a and each binary constraint as an arc, then the process of enforcing local consistency in each part of the graph causes inconsistent values to be eliminated throughout the graph. There are different types of local consistency, which we now cover in turn.

So far our search algorithm considers the constraints on a variable only at the time that the Variable is chosen by SELECT-VNASSIGNED-VARIABLE.

But by looking at some of the constraints earlier in the search, or even before the search has started, we can drastically reduce the search space.

Forward checking

- One way to make better use of constraints during search is called **forward checking**.
- Whenever a variable X is assigned, the forward checking process looks at each unassigned variable Y that is connected to X by a constraint and deletes from Y 's domain any value that is inconsistent with the value chosen for X.
- The following figure shows the progress of a map-coloring search with forward checking.

	WA	*NT*	*Q*	*NSW*	*V*	*SA*	*T*
Initial domains	R G B	R G B	R G B	R G B	R G B	R G B	R G B
After *WA=red*	Ⓡ	G B	R G B	R G B	R G B	G B	R G B
After *Q=green*	Ⓡ	B	Ⓖ	R B	R G B	B	R G B
After *V=blue*	Ⓡ	B	Ⓖ	R	Ⓑ		R G B

Figure 5.6 The progress of a map-coloring search with forward checking. $WA = red$ is assigned first; then forward checking deletes red from the domains of the neighboring variables NT and SA. After $Q = green$, $green$ is deleted from the domains of NT, SA, and NSW. After $V = blue$, $blue$ is deleted from the domains of NSW and SA, leaving SA with no legal values.

Although forward checking detects many inconsistencies, it does not detect all of them.

- **Constraint propagation** is the general term for propagating the implications of a constraint on one variable onto other variables.

Node consistency

A single variable (corresponding to a node in the CSP network) is **node-consistent** if all the values in the variable's domain satisfy the variable's unary constraints. For example, in the variant of the Australia map-coloring problem where South Australians dislike green, the variable SA starts with domain {red , green, blue}, and we can make it node consistent by eliminating green, leaving SA with the reduced domain {red , blue}.We say that a network is node-consistent if every variable in the network is node-consistent.

Arc Consistency

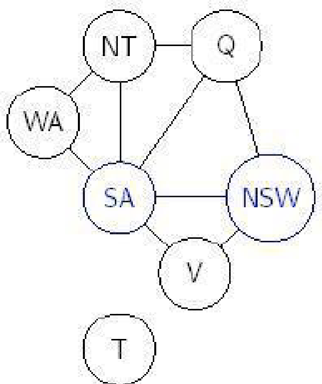


Figure: Australian Territories

- One method of constraint propagation is to enforce **arc consistency**
  - Stronger than forward checking
  - Fast
- *Arc* refers to a *directed* arc in the constraint graph
- Consider two nodes in the constraint graph (e.g., *SA* and *NSW*)
  - An arc is **consistent** if
  - For every value x of SA
  - There is some value y of NSW that is consistent with x
- Examine arcs for consistency in *both* directions

K-Consistency

- Can define stronger forms of consistency

> *k*-Consistency
>
> A CSP is ***k*-consistent** if, for any consistent assignment to $k-1$ variables, there is a consistent assignment for the k-th variable

- 1-consistency (**node consistency**)
  - Each variable by itself is consistent (has a non-empty domain)
- 2-consistency (**arc consistency**)
- 3-consistency (**path consistency**)
  - Any pair of adjacent variables can be extended to a third

Local Search for CSPs

- Local search algorithms good for many CSPs
- Use complete-state formulation
  - Value assigned to every variable
  - Successor function changes one value at a time
- Have already seen this
  - Hill climbing for 8-queens problem (AIMA § 4.3)
- Choose values using **min-conflicts** heuristic
  - Value that results in the minimum number of conflicts with other variables

The key idea of CSP constraint propagation is **local consistency**. If we treat each variable as a node in a graph and each binary constraint as an arc, then the process of enforcing local consistency in each part of the graph causes inconsistent values to be eliminated throughout the graph. There are different types of local consistency, which we now cover in turn.

The Structure of Problems
Problem Structure

- Consider ways in which the structure of the problem's constraint graph can help find solutions
- Real-world problems require decomposition into subproblems

Independent Sub problems

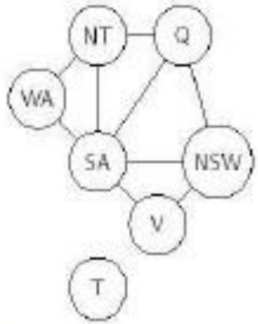


Figure: Australian Territories

- T is not connected
- Coloring T and coloring remaining nodes are **independent subproblems**
- *Any* solution for T combined with *any* solution for remaining nodes solves the problem
- Independent subproblems correspond to **connected components** of the constraint graph
- Sadly, such problems are rare

Tree-Structured CSPs

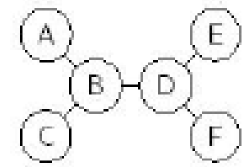


Figure: Tree-Structured CSP

- In most cases, CSPs are connected
- A simple case is when the constraint graph is a **tree**
- Can be solved in time linear in the number of variables
  - Order variables so that each parent precedes its children
  - Working "backward," apply arc consistency between child and parent
  - Working "forward," assign values consistent with parent

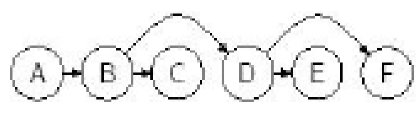


Figure: Linear ordering

The key idea is **local consistency**. If we treat each variable as a node in a graph and each binary constraint as an arc, then the process of enforcing local consistency in each part of the graph causes inconsistent values to be eliminated throughout the graph. There are different types of local consistency, which we now cover in turn.

Global constraints

Global constraint is one involving an arbitrary number of variables (but not necessarily all variables). Global constraints occur frequently in real problems and can be handled by special-purpose algorithms that are more efficient than the general-purpose methods described so far. For example, the Alldiff constraint says that all the variables involved must have distinct values (as in the cryptarithmetic problem above and Sudoku puzzles below). One simple form of inconsistency detection for Alldiff constraints works as follows:
if m variables are involved in the constraint, and if they have n possible distinct values altogether, and m>n, then the constraint cannot be satisfied.

This leads to the following simple algorithm:
First, remove any variable in the constraint that has a singleton domain, and delete that variable's value from the domains of the remaining variables.
Repeat as long as there are singleton variables. If at any point an empty domain is produced or there are more variables than domain values left, then an inconsistency has been detected.

Resource constraint

Another important higher-order constraint is the **resource constraint**, sometimes called the atmost constraint. For example, in a scheduling problem, let P denote the numbers of personnel assigned to each of four tasks. The constraint that no more than 10 personnel are assigned in total is written as Atmost (10,P1,P2,P3,P14,...,P).

Instead, domains are represented by upper and lower bounds and are managed by **bounds propagation**. For example, in an airline-scheduling problem, let's suppose there are two flights, F 1and F2, for which the planes have capacities 165 and 385, respectively. The initial domains for the numbers of passengers on each flight are then D1=[0, 165] and D2=[0, 385] .

Sudoku example

The popular **Sudoku** puzzle has introduced millions of people to constraint satisfaction problems, although they may not recognize it. A Sudoku board consists of 81 squares, some of which are initially

filled with digits from 1 to 9. The puzzle is to fill in all the remaining squares such that no digit appears twice in any row, column, or 3 ×3 box (see Figure 6.4). A row, column, or box is called a **unit**.

The Sudoku puzzles that are printed in newspapers and puzzle books have the property that there is exactly one solution. Although some can be tricky to solve by hand, taking tens of minutes, even the hardest Sudoku problems yield to a CSP solver in less than 0.1 second. Sudoku puzzle can be considered a CSP with 81 variables, one for each square. We use the variable names A1 through A9 for the top row (left to right), down to I1 through I9 for the bottom row. The empty squares have the domain {1, 2, 3, 4, 5, 6, 7, 8, 9} and the pre- filled squares have a domain consisting of a single value. In addition, there are 27 different

	1	2	3	4	5	6	7	8	9
A			3		2		6		
B	9			3		5			1
C			1	8		6	4		
D			8	1		2	9		
E	7								8
F			6	7		8	2		
G			2	6		9	5		
H	8			2		3			9
I			5		1		3		

(a)

	1	2	3	4	5	6	7	8	9
A	4	8	3	9	2	1	6	5	7
B	9	6	7	3	4	5	8	2	1
C	2	5	1	8	7	6	4	9	3
D	5	4	8	1	3	2	9	7	6
E	7	2	9	5	6	4	1	3	8
F	1	3	6	7	9	8	2	4	5
G	3	7	2	6	8	9	5	1	4
H	8	1	4	2	5	3	7	6	9
I	6	9	5	4	1	7	3	8	2

(b)

Figure 6.4 (a) A Sudoku puzzle and (b) its solution.

Alldiff constraints: one for each row, column, and box of 9 squares.

$$Alldiff(A1, A2, A3, A4, A5, A6, A7, A8, A9)$$
$$Alldiff(B1, B2, B3, B4, B5, B6, B7, B8, B9)$$
$$\ldots$$
$$Alldiff(A1, B1, C1, D1, E1, F1, G1, H1, I1)$$
$$Alldiff(A2, B2, C2, D2, E2, F2, G2, H2, I2)$$
$$\ldots$$
$$Alldiff(A1, A2, A3, B1, B2, B3, C1, C2, C3)$$
$$Alldiff(A4, A5, A6, B4, B5, B6, C4, C5, C6)$$

It is interesting to note how far we can go without saying much that is specific to Sudoku. We do of course have to say that there are 81 variables, that their domains are the digits 1 to 9, and that there are 27 *Alldiff* constraints. But beyond that, all the strategies—arc consistency, path consistency, etc.—apply generally to all CSPs, not just to Sudoku problems. Even naked triples is really a strategy for enforcing consistency of *Alldiff* constraints and has nothing to do with Sudoku *per se*. This is the power of the CSP formalism: for each new problem area, we only need to define the problem in terms of constraints; then the general constraint-solving mechanisms can take over.

BACKTRACKING SEARCH:

The term backtracking search is used for depth-first search that chooses values for one variable at a time and backtracks when a variable has no legal values left to assign.

```
function BACKTRACKING-SEARCH(csp) returns a solution, or failure
  return BACKTRACK({ }, csp)

function BACKTRACK(assignment, csp) returns a solution, or failure
  if assignment is complete then return assignment
  var ← SELECT-UNASSIGNED-VARIABLE(csp)
  for each value in ORDER-DOMAIN-VALUES(var, assignment, csp) do
     if value is consistent with assignment then
        add {var = value} to assignment
        inferences ← INFERENCE(csp, var, value)
        if inferences ≠ failure then
           add inferences to assignment
           result ← BACKTRACK(assignment, csp)
           if result ≠ failure then
              return result
     remove {var = value} and inferences from assignment
  return failure
```

Figure 6.5 A simple backtracking algorithm for constraint satisfaction problems. The algorithm is modeled on the recursive depth-first search of Chapter 3. By varying the functions SELECT-UNASSIGNED-VARIABLE and ORDER-DOMAIN-VALUES, we can implement the general-purpose heuristics discussed in the text. The function INFERENCE can optionally be used to impose arc-, path-, or k-consistency, as desired. If a value choice leads to failure (noticed either by INFERENCE or by BACKTRACK), then value assignments (including those made by INFERENCE) are removed from the current assignment and a new value is tried.

The term **backtracking search** is used for a depth-first search that chooses values for one variable at a time and backtracks when a variable has no legal values left to assign. It repeatedly chooses an unassigned variable, and then tries all values in the domain of that variable in turn, trying to find a solution. If an inconsistency is detected, then BACKTRACK returns failure, causing the previous call to try another value. Part of the search tree for the Australia problem is shown in Figure 6.6, where we have assigned variables in the order WA, NT,Q,.... Because the representation of CSPs is tandardized, there is no need to supply BACKTRACKING-SEARCH with a domain-specific initial state, action function, transition model, or goal test.

Notice that BACKTRACKING-SEARCH keeps only a **single representation of a state and alters that representation rather than creating new ones**.

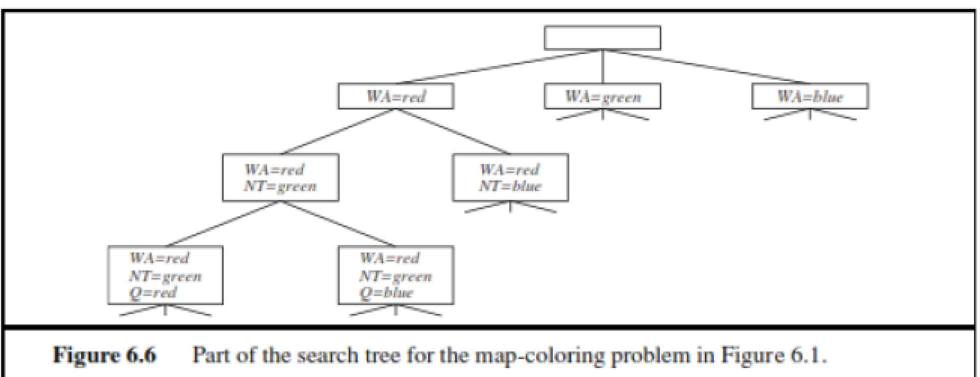


Figure 6.6 Part of the search tree for the map-coloring problem in Figure 6.1.

It turns out that we can solve CSPs efficiently *without* such domain-specific knowledge. Instead, we can add some sophistication to the unspecified functions, using them to address the following questions:

1. Which variable should be assigned next (SELECT-UNASSIGNED-VARIABLE), and in what order should its values be tried (ORDER-DOMAIN-VALUES)?

2. What inferences should be performed at each step in the search (INFERENCE)?

3. When the search arrives at an assignment that violates a constraint, can the search avoid repeating this failure?

ADVERSARIAL SEARCH

GAME PLAYING

There were **two reasons that games appeared to be a good domain** in which to explore machine intelligence:

- They provide a structured task in which it is very easy to measure success or failure.
- They did not obviously require large amounts of knowledge. They were thought to be solvable by straightforward search from the starting state to a winning position

The first of these reasons remains valid and accounts for continued interest in the area of game playing by machine.

Unfortunately, the second is not true for any but the simplest games.

For example, consider chess.

- The average branching factor is around 35.
- In an average game, each player might make 50 moves.
- So in order to examine the complete game tree, we would have to examine 35^{100} positions.

Thus it is clear that a program that simply does a straight forward search of the game tree will not be able to select even its first move during the lifetime of its opponent.

Some kinds of heuristic search procedure is necessary to improve the effectiveness of a search-based problem-solving program two things can be done:

1. *Improve the generate procedure so that only good moves (or paths) are generated.*
2. *Improve the test procedure so that the best moves (or paths) will be recognized and explored first.*

As the number of legal moves available increases, it becomes increasingly important to apply heuristics to select only those that have some kind of promise.

The ideal way to use a search procedure to find a solution to a problem is to generate moves through the problem space until a goal state is reached.

In the context of game-playing programs, a goal state is one in which we win. Unfortunately, for interesting games such as chess, it is not usually possible, even with a good Plausible- move generator, to search until a goal state is found.

Static evaluation function

In order to choose the best move, the resulting positions must be compared to discover which is most advantageous. This is done using a **static evaluation function**, which uses whatever information it has to evaluate individual board positions by estimating how likely they are to lead eventually to a win.

A very simple static evaluation function for chess based on piece advantage was proposed by Turing —simply add the values of black's pieces (B), the values of white's pieces (W), and then

compute the quotient W/B.

A more sophisticated approach was that taken in Samuel's checkers program, in which the static evaluation function was a linear combination of several simple functions. Thus the complete evaluation function had the form:

c1 x pieceadvantage + c2 x advancement + c3 x centercontrol

Unfortunately, deciding which moves have contributed to wins and which to losses is not always easy. Suppose we make a very bad move, but then, because the opponent makes a mistake, we ultimately win the game. We would not like to give credit for winning to our mistake.

Two important knowledge-based components of a good game-playing program:

- a good plausible-move generator and
- a good static evaluation function.

They must both incorporate a great deal of knowledge about the particular game being played. But unless these functions are perfect, we also need a search procedure that makes it possible to look ahead as many moves as possible to see what may occur.
For a simple one-person game or puzzle, the A* algorithm can be used. It can be applied to reason forward from the current state as far as possible in the time allowed. The heuristic function h' can be applied at terminal nodes and used to propagate values back up the search graph so that the best next move can be chosen. But because of their adversarial nature, this procedure is inadequate for two-person games such as chess.
As values are passed back up, different assumptions must be made at levels where the program chooses the move and at the alternating levels where the opponent chooses. There are several ways that this can be done. The most commonly used method is the minimax procedure.
An alternative approach is the B* algorithm which works on both standard problem-solving trees and on game trees.

MINIMAX SEARCH PROCEDURE

- The minimax search procedure is a depth-first, depth-limited search procedure.
- The idea is to *start at the current position and use the plausible-move generator* to generate the set of possible successor positions.
- *Static evaluation function* is applied to the positions and simply choose the best one. After doing so, the values are backed up to the starting position to represent the evaluation of it.

The starting position is exactly as good for us as the position generated by the best move we can make next. Here we assume that the static evaluation function returns large values to indicate good situations for us, so our goal is to maximize the value of the static evaluation function of the next board position.
An example of this operation is shown in figure 2.1. It assumes a static evaluation functions that returns values ranging from -10 to 10, with 10 indicating a win for us, -10 a win for the opponent, and 0 an even match. Since our goal is to maximize the value of the heuristic function, we choose to move to B. Backing B's value up to A, we can conclude that A's value is 8, since we know we can move to a position with a value of 8.

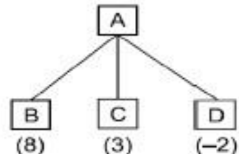


Figure 2.1 One Ply Search

But since we know that the static evaluation function is not completely accurate, we would like to carry the search farther ahead than one ply.
This could be very important, for example, in a chess game in which we are in the middle of a piece exchange. After our move, the situation would appear to be very good, but, if we look one move

ahead, we will see that one of our pieces also gets captured and so the situation is not as favorable as it seemed.

- **Look ahead to see what will happen to each of the new game positions at the next move** which will be made by the opponent.
- Instead of applying the static evaluation function to each of the positions that we just generated, we apply the **plausible-move generator, generating a set of successor positions for each position.**
- If we wanted to stop here, at two-ply look ahead, we could apply the static evaluation function to each of these positions, as shown in figure 2.2.

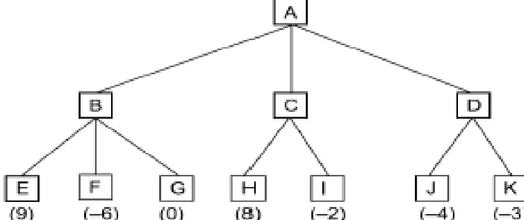


Figure 2.2: Two Ply Search

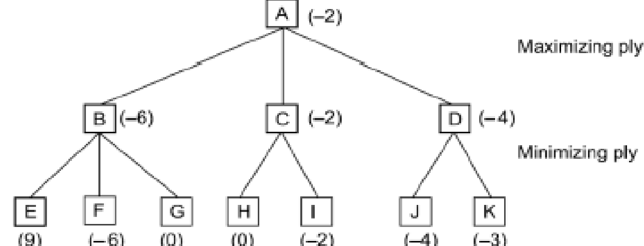


Figure 2.3: **Backing Up the Values of a Two – Ply Search**

But now we must take into account that the opponent gets to choose which successor moves to make and thus which terminal value should be backed up to the next level.

Suppose we made move B. Then the opponent must choose among moves E,F, and G.

The opponent's goal is to minimize the value of the evaluation function, so he or she can be expected to choose move F. This means that if we make move B, the actual position in which we will end up one move later is very bad for us. This is true even though a possible configuration is that represented by node E, which is very good for us. But since at this level we are not the ones to move, we will not get to choose it. Figure 2.3 shows the result of propagating the new values up the tree.

- **At the level representing the opponent's choice, the minimum value was chosen and backed up. At the level representing our choice, the maximum value was chosen.**

Once the values from the second ply are backed up, it becomes clear that the correct move for us to make at the first level, given the information we have available, is C, since there is nothing the opponent can do from there to produce a value worse than -2.

This process can be repeated for as many ply as time allows, and the more accurate evaluations that are produced can be used to choose the correct move at the top level.

- *The alternation of maximizing at alternate ply when evaluations are being pushed back up corresponds to the opposing strategies of the two players and gives this method the name minimax.*

Game can be formally defined as a search problem as below:

- S_0: The **initial state**, which specifies how the game is set up at the start.
- PLAYER(s): Defines which player has the move in a state.
- ACTIONS(s): Returns the set of legal moves in a state.
- RESULT(s, a): The **transition model**, which defines the result of a move.
- TERMINAL-TEST(s): A **terminal test**, which is true when the game is over and false otherwise. States where the game has ended are called **terminal states**.
- UTILITY(s, p): A **utility function** (also called an objective function or payoff function), defines the final numeric value for a game that ends in terminal state s for a player p. In

Minimax algorithm

```
function MINIMAX-DECISION(state) returns an action
   inputs: state, current state in game
   return the a in ACTIONS(state) maximizing MIN-VALUE(RESULT(a, state))

function MAX-VALUE(state) returns a utility value
   if TERMINAL-TEST(state) then return UTILITY(state)
   v ← −∞
   for a, s in SUCCESSORS(state) do v ← MAX(v, MIN-VALUE(s))
   return v

function MIN-VALUE(state) returns a utility value
   if TERMINAL-TEST(state) then return UTILITY(state)
   v ← ∞
   for a, s in SUCCESSORS(state) do v ← MIN(v, MAX-VALUE(s))
   return v
```

Types of games:

	deterministic	chance
perfect information	chess, checkers, go, othello	backgammon monopoly
imperfect information	battleships, blind tictactoe	bridge, poker, scrabble nuclear war

It is a straightforward recursive procedure that relies on two auxiliary procedures that are specific to the game being played:

1. MOVEGEN(Position, Player)-The plausible-move generator, which returns a list of nodes representing the move that can be made by Player in Positions.
2. STATIC(Position, Player)- The static evaluation function, which returns a number representing the goodness of position from the standpoint of player[2].

As with any recursive program, a critical issue in the design of the MINIMAX procedure is when to stop the recursion and simply call the static evaluation function. There are a variety of factors that may influence this decision. They include:

- Has one side won?
- How many ply have we already explored?
- How promising is this path?
- How much time is left?
- How stable is the configuration?

DEEP-ENOUGH, which is assumed to evaluate all of these factors and to return TRUE if the search should be stopped at the current level and FALSE otherwise. Our simple implementation of DEEP-ENOUGH will take two parameters, Position and Depth. It will ignore its Position parameter and simply return TRUE if its Depth parameter exceeds a constant cutoff value.

One problem that arises in defining MINIMAX as a recursive procedure is that it needs to return not one but two results:

- The backed-up value of the path it chooses.
- The path itself. We return the entire path even though probably only the first element, representing the best move from the current position, is actually needed.

We assume that MINIMAX returns a structure containing both results and that we have two functions, VALUE and PATH, that extract the separate components.

Since we define the MINIMAX procedure as a recursive functions, we must also specify how it is to be called initially. It takes three parameters, a board position, the current depth of the search, and the player to move. So the initial call to compute the best move from the position CURRENT should be

MINIMAX(CURRENT, 0,PLAYER-ONE) if PLAYER-ONE is to move, or
MINIMAX(CURRENT,0,PLAYER-TWO) if PLAYER-TWO is to move.

MINIMAX(Position, Depth, Player)

1. If DEEP-ENOUGH(Position, Depth), then return the structure
   VALUE=STATIC(Position, Player);
   PATH=nil

This indicated that there is no path from this node and that its value is that determined by the static evaluation functions.

2. Otherwise, generate one more ply of the tree by calling the function MOVE-GEN(Position Player) and setting SUCCESSORS to the list it returns.

3. If SUCCESSORS is not empty, then there are no moves to be made, so return the same structure that would have been that would have been returned if DEEP-ENOUGH had returned true.

4. If SUCCESSORS is not empty, then examine each element in turn and keep track of the best one. This is done as follows.

Initialize BEST-SCORE to the minimum value that STATIC can return. It will be updated to reflect the best score that can be achieved by an element of SUCCESSORS.

For each element SUCC of SUCCESSORS, do the following:

(a) Set RESULT-SUCC TO
    MINIMAX(SUCC, Depth+1, OPPOSITE(Player))[3]

This recursive call to MINIMAX will actually carry out the exploration of SUCC.

(b) Set NEW-VALUE to-VALUE(RESULT-SUCC).This will cause it to reflect the merits of the position from the opposite perspective from that of the next lower level.

(c)If NEW-VALUE > BEST-SCORE, Then we have found a successor that is better than any that have been examined so far. Record this by doing the following:

i. Set BEST-SCORE to NEW-VALUE.

ii. The best known path is now from CURRENT to SUCC and then on to the appropriate path down from SUCC as determined by the recursive call to MINIMAX, So set BEST-PATH to the result of attaching SUCC to the front of PATH(RESULT-SUCC).

5. Now that all the successors have been examined, we know the value of Position as well as which path to take form it. So return the structure.

VALUE=BEST-SCORE

PATH=BEST-PATH

When the initial call to MINIMAX returns, the best move from CURRENT is the first element of PATH.

OPTIMAL DECISION IN GAMES : ALPHA BETA PRUNING:

The MINMAX search procedure is slightly modified to handle both maximizing and minimizing players. It is also necessary to modify the branch and bound strategy to include two bounds, one for each of the players. This modified strategy is called alpha beta pruning.

It requires the maintenance of two threshold values,

- Alpha α : **lower bound on the value that a maximizing node** may ultimately be assigned
- Beta β : **upper bound on the value that a minimizing** node may be assigned.

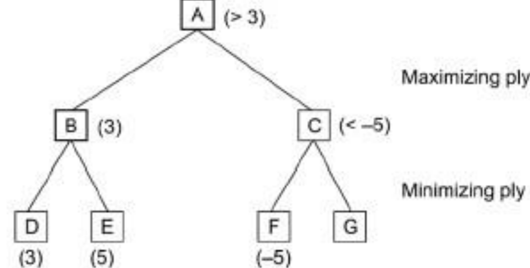


Figure 2.4 An Alpha Cutoff

After examining node F, we know that the opponent is guaranteed a score of -5 or less at C (since the opponent is the minimizing player). But we also know that we are guaranteed a score of 3 or greater at node A which we can achieve if we move to B.

Any other move that produces a score of less than 3 is worse than the move to B, and we can ignore it. After examining only F we are sure that a move to C is worse (it will be <=-5) regardless of the score of node G. Thus we need not bother to explore node G at all. Of course, cutting out one node may not appear to justify the expense of keeping track of the limits and checking them, but if we were exploring this tree to 6 ply, then we would have eliminated not a single node but an entire tree 3 ply deep.

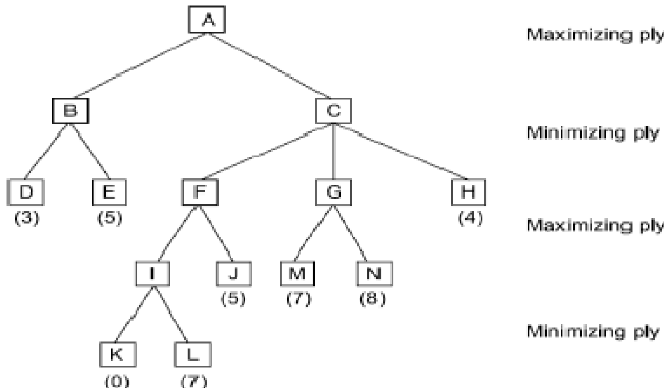


Figure 2.5 Alpha and Beta Cutoffs

To see how the two thresholds, alpha and beta can both be used, consider the example shown in figure 2.5. In searching this tree the entire sub tree headed by B is searched, and we discover that at A we can expect a score of at least 3.

When this alpha value is passed down to F**, it will enable us to skip the exploration of L**. After K is examined, we see that I is guaranteed a maximum score of 0, which means that F is guaranteed a minimum of 0. But this is less than alpha's value of 3, so no more branches of I need to be considered. The maximizing player already knows not to choose to move to C and then to I since, if that move is made, the resulting score will be no better than 0 and a score of 3 can be achieved by moving to B instead.

Now let's see how the value of beta can be used. After cutting off further exploration of I, J is examined, yielding a value of 5, which is assigned as the value of F (since it is the maximum of 5 and 0). This value becomes the value of beta at node C. it indicates that C is guaranteed to get a 5 or less. Now we must expand G. First M is examined and it has a value of 7, which is passed back to G as its tentative value. But now 7 is compared to beta (5). It is greater, and the player whose turn it is at node C is trying to minimize. So this player will not choose G, which would lead to a score of at least 7, since there is an alternative move to F, which will lead to a score of 5. Thus it is not necessary to explore any of the other branches of G.

From this example,

- At **maximizing levels**, we can rule out a move early if it becomes clear that its **value will be less than the current threshold**,
- At **minimizing levels**, search will be terminated if **values that are greater than the current threshold are discovered**.
- ***At maximizing levels, only beta is used to determine whether to cut off the search, and at minimizing levels only alpha is used.***

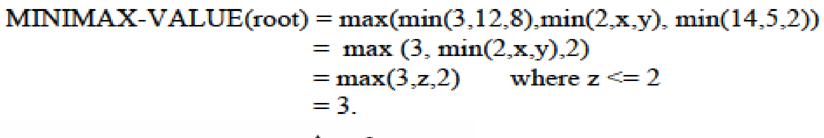
ALPHA-BETA SEARCH ALGORITHM

- The problem with minimax search is that the number of game states it has to examine is exponential in the number of moves. Unfortunately we can't eliminate the exponent, but we can effectively cut it in half. The trick is that it is possible to compute the correct minimax decision without looking at every node in the game tree.

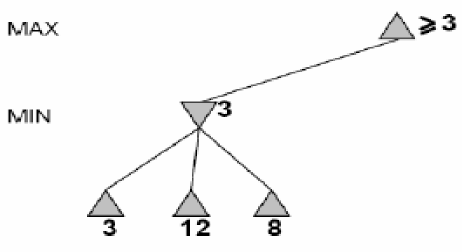
- Alpha-beta pruning technique is applied to a standard minimax tree, it returns the same move as minimax would, but prunes away branches that cannot possibly influence the final decision.

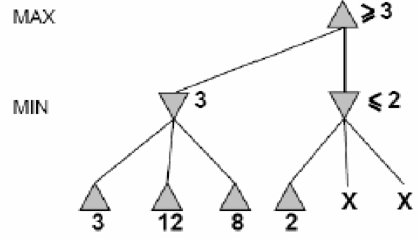
The outcome is that we can identify the minimax decision without ever evaluating two of the leaf nodes.

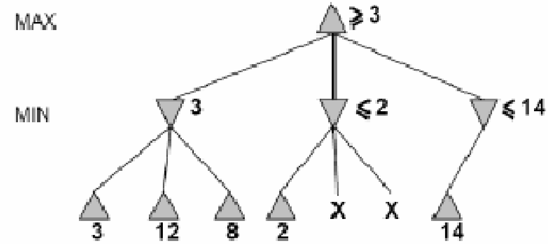
- Another way to look at this is as a simplification of the formula for MINIMAX-VALUE. Let the two unevaluated successors of node C in figure have values x and y and let z be the minimum of x and y. The value of the root node is given by

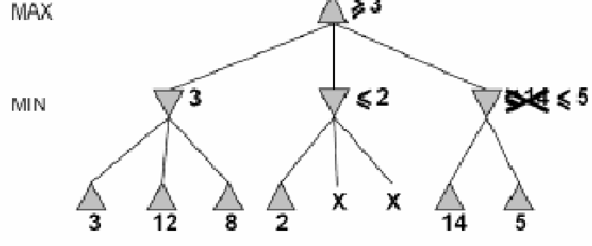


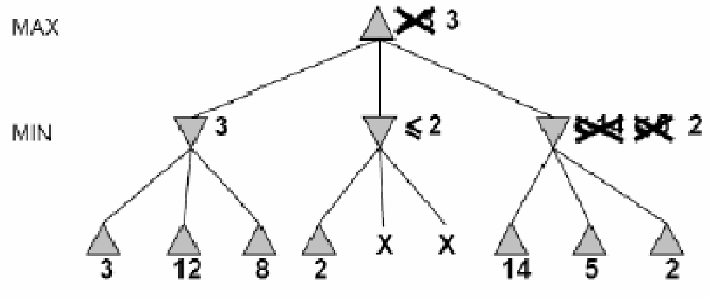
$$\begin{aligned}\text{MINIMAX-VALUE(root)} &= \max(\min(3,12,8),\min(2,x,y),\min(14,5,2))\\ &= \max(3,\min(2,x,y),2)\\ &= \max(3,z,2) \quad \text{where } z \le 2\\ &= 3.\end{aligned}$$







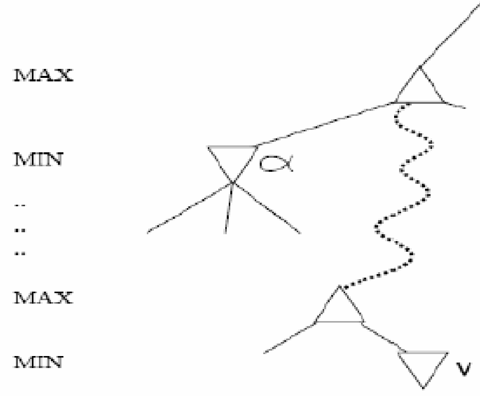




- In other words, the value of the root and hence the minimax decision are independent of the values of the pruned leaves x and y

- Alpha-beta pruning can be applied to trees of any depth, and it is often possible to prune entire subtrees rather than just leaves.

- The general principle is this: consider a node n somewhere in the tree such that Player has a choice of moving to that node.



- If Player has a better choice m either at the parent node of n or at any choice point further up, then n will never be reached in actual play. So once we have found out enough about n to reach this conclusion, we can prune it.

- Alpha-beta pruning gets its name from the following two parameters that describe bounds on the backed-up values that appear anywhere along the path:

  α = the value of the best (i.e., highest-value) choice we have found so far at any choice point along the path for MAX.
  β= the value of the best (i.e., lowest-value) choice we have found so far at any choice point along the path for MIN

- Alpha-beta search updates the values of α and β as it goes along and prunes the remaining branches at a node as soon as the value of the current node is known to be worse than the current α or β value for MAX or MIN respectively.

- The effectiveness of alpha-beta pruning is highly dependent on the order in which the successors are examined. For ex, in figure we could not prune any successors of D at all because the worst successors were generated first. If the third successor had been generated first, we would have been able to prune the other two. This suggests that it might be worthwhile to try to examine first the successors that are likely to be best.

The alpha-beta search algorithm

```
function ALPHA-BETA-SEARCH(state) returns an action
  inputs: state, current state in game

  v ← MAX-VALUE(state, −∞, +∞)
  return the action in SUCCESSORS(state) with value v
```

```
function MAX-VALUE(state, α, β) returns a utility value
  inputs: state, current state in game
          α, the value of the best alternative for MAX along the path to state
          β, the value of the best alternative for MIN along the path to state

  if TERMINAL-TEST(state) then return UTILITY(state)
  v ← −∞
  for a, s in SUCCESSORS(state) do
    v ← MAX(v, MIN-VALUE(s, α, β))
    if v ≥ β then return v
    α ← MAX(α, v)
  return v
```

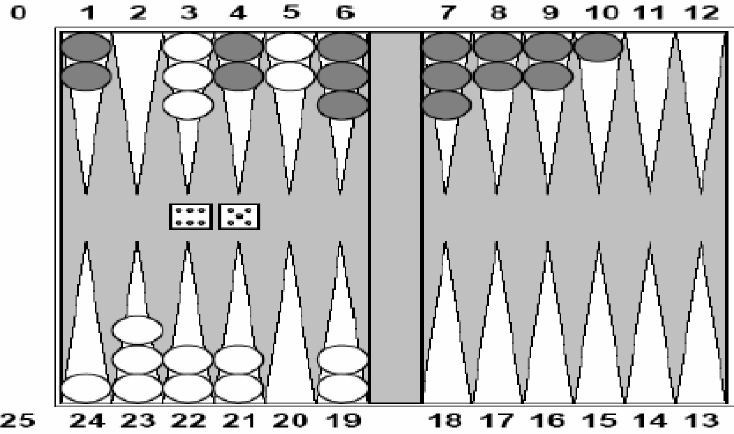
```
function MIN-VALUE(state, α, β) returns a utility value
  inputs: state, current state in game
          α, the value of the best alternative for MAX along the path to state
          β, the value of the best alternative for MIN along the path to state

  if TERMINAL-TEST(state) then return UTILITY(state)
  v ← +∞
  for a, s in SUCCESSORS(state) do
    v ← MIN(v, MAX-VALUE(s, α, β))
    if v ≤ α then return v
    β ← MIN(β, v)
  return v
```

STOCHASTIC GAMES

- In real life, there are many unpredictable external events that put us into unforeseen situations.
- Many games mirror this unpredictability by including a random element, such as the throwing of dice.
- In this way, they take us a step nearer reality, and it is worthwhile to see how this affects the decision-making process.
- Backgammon is a typical game that combines luck and skill. Dice are rolled at the beginning of a player's turn to determine the legal moves. In the backgammon position of figure, for example, white has rolled a 6-5, and has four possible moves.
- Although White knows what his or her own legal moves are, White does not know what Black is going to roll and thus does not know what Black's legal moves will be.
- That means White cannot construct a standard game tree of the sort we saw in chess and tic-tac-toe.
- A game tree in backgammon must include **chance nodes** in addition to MAX and MIN node Chance nodes are shown as circles in Fig.
- The branches leading from each chance node denote the possible dice rolls, and each is labeled with the roll and the chance that it will occur.

Figure: A typical backgammon position. The goal of the game is to move all one's pieces off the board.



- There are 36 ways to roll two dice, each equally likely; but because a 6-5 is the same as a 5-6, there are only 21 distinct rolls a 1/18 chance each.
- The next step is to understand how to make correct decisions.
- Obviously, we still want to pick the move that leads to the best position.
- However, the resulting positions do not have definite minimax values.
- Instead, we can only calculate the expected value, where the expectation is taken over all the possible dice rolls that could occur.

This leads us to generalize the minimax value for deterministic games to an expectiminimax value for games with chance nodes.

- Terminal nodes and MAX and MIN nodes work exactly the same way as before; chance nodes are evaluated by taking the weighted average of the values resulting from all possible dice rolls, that is,

EXPECTIMINIMAX(n) =

UTILITY(n)	if n is a terminal state
$\text{Max}_{s \in \text{Successors}(n)}$ EXPECTIMINIMAX(s)	if n is a MAX node
$\text{min}_{s \in \text{Successors}(n)}$ EXPECTIMINIMAX(s)	if n is a MIN node
$\sum_{s \in \text{Successors}(n)} P(s)$. EXPECTIMINIMAX(s)	if n is a chance node

Where the successor function for a chance node n simply augments the state of n with each possible dice roll to produce each successor s and P(s) is the probability that that dice roll occurs. These equations can be backed up recursively all the way to the root of the tree, just as in minimax.

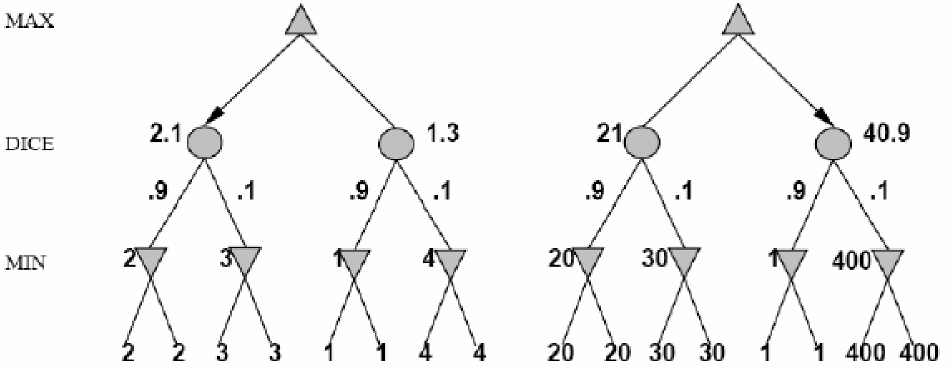
Position evaluation in games with chance nodes

- As with minimax, the obvious approximation to make with expectiminimax is to cut the search off at some point and apply an evaluation function to each leaf.

- One might think that evaluation functions for games such as backgammon should be just like evaluation functions for chess – they just need to give higher scores to better positions.

- But in fact, the presence of chance nodes means that one has to be more careful about what the evaluation values mean.

Figure: An order preserving transformation on leaf values changes the best move.



- Figure shows what happens: with an evaluation function that assigns values [1,2,3,4] to the leaves, move A1 is best; with values [1,20,30,400], move A2 is best. Hence, the program behaves totally differently if we make a change in the scale of some evaluation values.
- It turns out that, to avoid this sensitivity, the evaluation function must be a positive linear transformation of the probability of winning from a position

Complexity of expectiminimax

- If the program knew in advance all the dice rolls that would occur for the rest of the game, solving a game with dice would be just like solving a game without dice, which minimax does in $O(b^m)$ time. Because expectiminimax is also considering all the possible dice-roll sequences, it will take $O(b^m n^m)$, where n is the number of distinct rolls.

- Even if the search depth is limited to some small depth d , the extra cost compared with that of minimax makes it unrealistic to consider looking ahead very far in most games of chance.
- In backgammon n is 21 and b is usually around 20, but in some situations can be as high as 4000 for dice rolls that are doubles. Three plies is probably all we could manage

UNIT II PROBABILISTIC REASONING

Acting Under Uncertainty

- When an agent knows enough facts about its environment, the logical plans and actions produces a guaranteed work.
- Unfortunately, agents never have access to the whole truth about their environment. Agents act under uncertainty.
  [Without full knowledge about the environment, taking decisions are difficult or it will go wrong.]

Nature of Uncertain Knowledge

- The Diagnosis: medicine, automobile repair, or whatever is a task that almst always involves uncertainty.
- Let us try to write rules for dental diagnosis using first-order logic, so that we can see how the logical approach breaks down. Consider the following rule:

$$\forall p\ \boldsymbol{symptom}(\boldsymbol{p}, \boldsymbol{Toothache}) \Rightarrow \boldsymbol{Disease}(\boldsymbol{p}, \boldsymbol{Cavity})$$

- The problem is that this rule is wrong.
- Not all the patients with toothaches have cavities; some of them have gum disease, swelling, or one of several other problems

$$\forall p\ \boldsymbol{Symptom}(\boldsymbol{p}, \boldsymbol{Toothache}) \\ \Rightarrow \boldsymbol{Diseae}(\boldsymbol{p}, \boldsymbol{Cavity}) \vee \boldsymbol{Disease}(\boldsymbol{p}, \boldsymbol{GumDisease}) \vee \boldsymbol{Disease}(\boldsymbol{p}, \boldsymbol{Swelling}) \ldots$$

- To make the rule true, we have to add almost unlimited list of possible causes.
- We could try a casual rule:

$$\forall p\ \boldsymbol{Disease}(\boldsymbol{p}, \boldsymbol{Cavity}) \Rightarrow \boldsymbol{Symptom}(\boldsymbol{p}, \boldsymbol{Toothache})$$

- But this rule is also not right either; not all cavities cause pain
- Toothache and a Cavity are unconnected, so the judgement may go wrong.
- This is a type of the medical domain, as well as most other judgmental domains: law, business, design, automobile repair, gardening, dating, and so on.
- Three main reasons of failures
  i. **Laziness**- we are too much lazy to represent all antecedants/consequents (this is our inability, we don't know all the reasons of toothache.)
  ii. **Theoritical ignorance**- there is no complete knowledge(we don't know the exact reason/ all the reasons of toothache.
  iii. **Practical ignorance**- not all tests can be run (we cant take all test to determine the problem)
- The agent take action, only a **degree of belief** relevant sentences.
- Our main tool for dealing with degrees of belief will be **probability theory.**
- **The Probability** assigns to each sentence a numerical degree of belief between 0 and 1.
- Probability theory provides a way of summarizing the uncertainty that come from the laziness & ignorance.

Downloaded by Viji Jai (nancymeena@gmail.com)

Uncertainty and rational decisions (we have uncertainty but still we want to take rational decisions)

- **Example** : Automated taxi
    i. Plan 1(A_{90}) – Leave 90 mins early
    ii. Plan 2((A_{180}) – Leave 180 mins early
    iii. Plan 3(A_{1440})- Leave 24 hours early

We have to evaluate the plan. Inorder to overcome, I have to evaluate the problem using utility values (timely arrival, whether my ride was legal, whether the ride was comfortable, whether it is safe ride)

Based on this we have to make rational decisions. We will leave that decision to decision theory , it will make the use of probability theory and utility theory. So, preferences will be given for utility values.

- Preferences, as expressed by utilities, are combined with probabilities in the general theory of rational decisions called **decision theory:**

$$Decision\ theory = probability\ theory + utility\ theory.$$

- Decision theory combines the agent's beliefs and desires, defining the best action as the one that **maximizes expected utility. (i.e,** not all the time the utility values are satisfied, but we have to maximize the expected utilities).

function DT-AGENT(*percept*) **returns** an *action*
  persistent: *belief_state*, probabilistic beliefs about the current state of the world
    action, the agent's action

  update *belief_state* based on *action* and *percept*
  calculate outcome probabilities for actions,
    given action descriptions and current *belief_state*
  select *action* with highest expected utility
    given probabilities of outcomes and utility information
  return *action*

A decision-theoretic agent that selects rational actions.

Basic Probability Notation

- **Sample space**- the set of all possible worlds (one instance is known as possible world)
  For example: two dice are rolled – 36 possible worlds (if I enumerate them I have (1,1) (1,2…(1,6), (2,1)….(6,6). 6*6 options total 36. These 36 instances are called 36 possible worlds)
- **Probability model**- associates a numerical probability with each possible world

Downloaded by Viji Jai (nancymeena@gmail.com)

$$0 \leq P(\omega) \leq 1 \text{ for every } \omega \text{ and } \sum_{\omega \in \Omega} P(\omega) = 1.$$

(In this example all events are equally likely, so there is no biasing for any instance, So we can say that the probability of each event is 1/36. Assigning this probability to a particular possible world is govern by this probability model)

W is representing one of the possible world, probability of any world will lie between 0 and 1. 0represent impossible event, 1represent certain event. So, it is true always for every w, & the summation of probability of all the possible world will 1. (we have 36 events each wit probability 1/36 , so summation is 1)

Unconditional Probability/ Prior Probability

- For example, rolling the 2 dices and they add up to 11(which instance will add 115 & 6 or 6 & 5. And their probability is 1/36 +1/36 i.e 2/36 1/18)

$$P(Total = 11) = P((5,6)) + P((6,5)) = 1/36 + 1/36 = 1/18.$$

Conditional Probability/Posterior Probability

- For example, rolling the two dices given that the first die is a 5, (this condition is imposed here, P(5,6)|$die_1 = 5$)

$$P\left(doubles \mid Die_1 = 5\right)$$

- Mathematically, conditional probability is given by, (find probability of A given B that is already occurred.

$$P(a \mid b) = \frac{P(a \wedge b)}{P(b)},$$

which holds whenever $P(b) > 0$. For example,

$$P(doubles \mid Die_1 = 5) = \frac{P(doubles \wedge Die_1 = 5)}{P(Die_1 = 5)}.$$

Product rule

$$P(a \wedge b) = P(a \mid b)P(b).$$

Downloaded by Viji Jai (nancymeena@gmail.com)

Random variables – variables in probability theory are called random variables.(we don't know the exact occurance of those variables, tat's why we call them as random variables.)

Domain- Each variable will have domain

(we have our syntax of logical statement inorder to represent our knowledge)

Example: "The probability that the patient has a cavity, given that she is a teenager with no toothache, is 0.1" as follows:

$$P(cavity \mid \neg toothache \wedge teen) = 0.1.$$

Probability distribution

- For example, Weather={ sunny, rain, cloudy, snow}

Sometimes we will want to talk about the probabilities of all the possible values of a random variable. We could write:

$$\begin{aligned} P(Weather = sun) &= 0.6 \\ P(Weather = rain) &= 0.1 \\ P(Weather = cloud) &= 0.29 \\ P(Weather = snow) &= 0.01 , \end{aligned}$$

but as an abbreviation we will allow,

$$\mathbf{P}(Weather) = \langle 0.6, 0.1, 0.29, 0.01 \rangle ,$$

- Statement P defines a probability distributions for the random variable Weather.
- For a continuous variables, P defines the probability density function(pdf)

Probability axioms

- Relationship between a proposition and its negation

$$\begin{aligned} P(\neg a) &= \textstyle\sum_{\omega \in \neg a} P(\omega) \\ &= \textstyle\sum_{\omega \in \neg a} P(\omega) + \sum_{\omega \in a} P(\omega) - \sum_{\omega \in a} P(\omega) \\ &= \textstyle\sum_{\omega \in \Omega} P(\omega) - \sum_{\omega \in a} P(\omega) \\ &= 1 - P(a) \end{aligned}$$

Downloaded by Viji Jai (nancymeena@gmail.com)

Inclusion-exclusion principle

$$P(a \vee b) = P(a) + P(b) - P(a \wedge b).$$

Sum rule

$P(A \vee B) = P(A) + P(B)$

Independent event

$P(A|B) = P(A)$

So, that we have product rule,

$P(A \wedge B) = P(A|B). P(B)$

$= P(A). P(B)$

So, this is true for independent events.

Full Joint Probability Distribution

- Distributions on multiple variables
- For example,
    - Weather= {sunny, rain, cloudy, snow}
    - Cavity={ cavity, $\neg cavity$}
    - Joint Probability distribution of Weather & Cavity

$$
\begin{aligned}
P(W = sun \wedge C = true) &= P(W = sun|C = true)\, P(C = true) \\
P(W = rain \wedge C = true) &= P(W = rain|C = true)\, P(C = true) \\
P(W = cloud \wedge C = true) &= P(W = cloud|C = true)\, P(C = true) \\
P(W = snow \wedge C = true) &= P(W = snow|C = true)\, P(C = true) \\
P(W = sun \wedge C = false) &= P(W = sun|C = false)\, P(C = false) \\
P(W = rain \wedge C = false) &= P(W = rain|C = false)\, P(C = false) \\
P(W = cloud \wedge C = false) &= P(W = cloud|C = false)\, P(C = false) \\
P(W = snow \wedge C = false) &= P(W = snow|C = false)\, P(C = false).
\end{aligned}
$$

Here, Weather & Cavity are the variables. Weather has 4 values & Cavity having 2 values. What are the possible combination of these variables. Consider (W, C) = 8 combinations.

So, find the probability of distribution on these 2 variables.

One of the instance we have, sunny weather with cavity 7 sunny with no cavity

Similarly, rain with cavity, rain without cavity. So, 4*2=8 combinations possible

Downloaded by Viji Jai (nancymeena@gmail.com)

So, every possible world will have some probability. Some possible world will be more probable i.e) probability will be more, called joint probability.

When you have multiple variables, probability distribution over multiple variables is known as joint probability.

And if you enlist all the possible world then it becomes, full joint probability distribution.

- Can be written as a single equation:

$$\mathbf{P}(Weather, Cavity) = \mathbf{P}(Weather \mid Cavity)\mathbf{P}(Cavity),$$

Inference Using Full joint Distributions

- To study the method for probabilistic inference. So, how to infer a new fact in case of uncertainty.
- When you infer a new fact, that fact will have some probability, because uncertainty is there.
- Given data itself will have a probability
- Consider an instance with three variables

	toothache		*¬toothache*	
	catch	*¬catch*	*catch*	*¬catch*
cavity	0.108	0.012	0.072	0.008
¬cavity	0.016	0.064	0.144	0.576

A full joint distribution for the *Toothache*, *Cavity*, *Catch* world.

catch (the dentist's nasty steel probe catches in my tooth).

The table is fully depicting the joint distribution of these three variables.

Toothache has 2 values, Cavity 2 values, Catch 2 values. Total 8 entries

- How to infer the probability of any proposition (new fact).
  Here, our new proposition is Cavity or Toothache
  Now, what is the probability of Cavity with Toothache we can have OR, we have only Cavity, only Toothache or both
- direct way to calculate the probability of any proposition, simple or complex: simply identify those possible worlds in which the proposition is true and add up their probabilities. For example, there are six possible worlds in which $Cavity \vee toothache$ holds:

$$P(cavity \vee toothache) = 0.108 + 0.012 + 0.072 + 0.008 + 0.016 + 0.064 = 0.28.$$

To compute conditional probabilities

- For example, we can compute the probability of a cavity, given evidence of a toothache, as follows:

$$P(cavity \mid toothache) = \frac{P(cavity \wedge toothache)}{P(toothache)}$$
$$= \frac{0.108 + 0.012}{0.108 + 0.012 + 0.016 + 0.064} = 0.6.$$

Just to check, we can also compute the probability that there is no cavity, given a toothache:

$$P(\neg cavity \mid toothache) = \frac{P(\neg cavity \wedge toothache)}{P(toothache)}$$
$$= \frac{0.016 + 0.064}{0.108 + 0.012 + 0.016 + 0.064} = 0.4.$$

In order to **reduce computation complexity**, we can find denominator only once, because it is repeated.
For that we have a concept of **Normalization.**

The two values sum to 1.0, as they should. Notice that the term $P(toothache)$ is in the denominator for both of these calculations. If the variable $Cavity$ had more than two values, it would be in the denominator for all of them. In fact, it can be viewed as a **normalization** constant for the distribution $\mathbf{P}(Cavity \mid toothache)$, ensuring that it adds up to 1. Throughout the chapters dealing with probability, we use α to denote such constants. With this notation, we can write the two preceding equations in one:

$$\mathbf{P}(Cavity|toothache) = \alpha \mathbf{P}(Cavity, toothache)$$
$$= \alpha [\mathbf{P}(Cavity, toothache, catch) + \mathbf{P}(Cavity, toothache, \neg catch)]$$
$$= \alpha [\langle 0.108, 0.016 \rangle + \langle 0.012, 0.064 \rangle] = \alpha \langle 0.12, 0.08 \rangle = \langle 0.6, 0.4 \rangle .$$

In other words, we can calculate $\mathbf{P}(Cavity \mid toothache)$ even if we don't know the value of $P(toothache)$! We temporarily forget about the factor $1/P(toothache)$ and add up the values for $cavity$ and $\neg cavity$, getting 0.12 and 0.08. Those are the correct relative proportions, but they don't sum to 1, so we normalize them by dividing each one by $0.12 + 0.08$, getting the true probabilities of 0.6 and 0.4. Normalization turns out to be a useful shortcut in many probability calculations, both to make the computation easier and to allow us to proceed when some probability assessment (such as $P(toothache)$) is not available.

Independence

- Independence between propositions a and b can be written as

$$P(a \mid b) = P(a) \quad \text{or} \quad P(b \mid a) = P(b) \quad \text{or} \quad P(a \wedge b) = P(a)P(b).$$

- Example, add Weather variable in our previous example

Downloaded by Viji Jai (nancymeena@gmail.com)

$$P(toothache, catch, Cavity, Weather)$$

- To find P(toothache, catch, cavity, cloudy), we use the product rule

$$P(toothache, catch, cavity, cloud)$$
$$= P(cloud \mid toothache, catch, cavity)P(toothache, catch, cavity).$$

- It seems safe to say that the weather does not influence the dental variables. Therefore, the following assertion seems reasonable

$$P(cloud \mid toothache, catch, cavity) = P(cloud).$$

From this, we can deduce

$$P(toothache, catch, cavity, cloud) = P(cloud)P(toothache, catch, cavity).$$

- Thus, the 32-element table for four variables can be constructed from one 8-element table and one 4-element table.

Bayes' Rule and its use

- When you have 2 independent events, two forms of product rule

$$P(a \wedge b) = P(a \mid b)P(b) \quad \text{and} \quad P(a \wedge b) = P(b \mid a)P(a).$$

Equating the two right-hand sides and dividing by $P(a)$, we get

$$P(b \mid a) = \frac{P(a \mid b)P(b)}{P(a)}.$$

This equation is known as **Bayes' rule** (also Bayes' law or Bayes' theorem). This simple equation underlies most modern AI systems for probabilistic inference.

In a task such as medical diagnosis, we often have conditional probabilities on causal relationships. The doctor knows $P(symptoms|disease))$) and want to derive a diagnosis, $P(disease|symptoms)$,

For example, a doctor knows that the disease meningitis causes a patient to have a stiff neck, say, 70% of the time. The doctor also knows some unconditional facts: the prior probability that any patient has meningitis is 1/50,000, and the prior probability that any patient has a stiff neck is 1%. Letting s be the proposition that the patient has a stiff neck and m be the proposition that the patient has meningitis, we have

Downloaded by Viji Jai (nancymeena@gmail.com)

$$P(s \mid m) = 0.7$$
$$P(m) = 1/50000$$
$$P(s) = 0.01$$
$$P(m \mid s) = \frac{P(s \mid m)P(m)}{P(s)} = \frac{0.7 \times 1/50000}{0.01} = 0.0014.$$

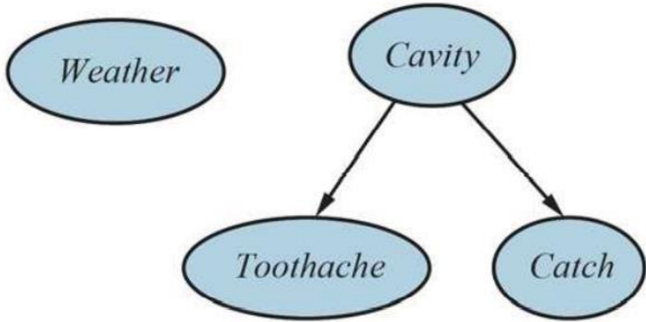
That is, we expect only 0.14% of patients with a stiff neck to have meningitis. Notice that even though a stiff neck is quite strongly indicated by meningitis (with probability 0.7), the probability of meningitis in patients with stiff necks remains small. This is because the prior probability of stiff necks (from any cause) is much higher than the prior for meningitis.

Probabilistic Reasoning

Bayesian Network

- Bayesian Network is to represent the dependencies among variables and to give a brief specification of any full joint probability distribution.
- Bayesian network is a data structure also called as belief network, probabilistic network, casual network, all knowledge map.
- The extension of Bayesian network is called as a decision network or influence diagram.
- A Bayesian is a directed graph in which each node is annotated with quantitative probability information.
- The full specification is a s follows:
    - A set of random variables makes up the **nodes** of the network. Variables may be discrete or continuous.
    - A set of directed links or **arrows** connects a pairs of nodes. If there is an arrow from node X to node Y, X is said to be a parent of Y.
    - Each node X has a conditional probability distribution P(X,(Parents(X)) that quantifies the effect of the parents on the node.**(X is parent of Y)**
    - The graph has no directed cycles (and hence is a directed, acyclic graph, or DAG.)

Example: Simple Bayesian network



A simple Bayesian network in which *Weather* is independent of the other three variables and *Toothache* and *Catch* are conditionally independent, given *Cavity*.

Downloaded by Viji Jai (nancymeena@gmail.com)

- Now, consider the example Burglar Alarm
- If a thief or unknown person enter into your compound, then the alarm rings.
- You have installed a new burglar alarm at home.
- It is fairly reliable at detecting a burglary, but also responds on occasion to minor earthquakes.
- You also have two neighbors, John and Mary, who have promised to call you at work when they hear the alarm.
- John always calls when he hears the alarm, but sometimes confuses the telephone ringing with the alarm and calls then, too.
- Mary, on the other hand, likes loud music and sometimes misses the alarm altogether.
- Given the evidence of who has or has not called, we would like to estimate the probability of a burglary.

Burglary

P(B=true)
.001

Earthquake

P(E=true)
.002

Alarm

B	E	P(A=true|B,E)
t	t	.70
t	f	.01
f	t	.70
f	f	.01

JohnCalls

A	P(J=true|A)
t	.90
f	.05

MaryCalls

A	P(M=true|A)
t	.70
f	.01

A typical Bayesian network, showing both the topology and the conditional probability tables (CPTs). In the CPTs, the letters *B*, *E*, *A*, *J*, and *M* stand for *Burglar*, *Earthquake*, *Alarm*, *JohnCalls*, and *MaryCalls*, respectively.

- This, is the Bayesian network for our example.
- Each node is having its own conditional probability table CPT,
- And in this diagram, alarm is directly depending on Burglary and Earthquake but John and Mary are depending on only the Alarm.
- So, in each CPT, they are having letters B→ Burglary, E→Earthquake, S→ alarm, J→John calls, M→ Mary calls.
- From the network, the topology shows that
    - Burglary and earthquakes directly affect the probability of the alarm,
    - But John and Mary call depends on the alarm.
- Our assumptions from the network,
    - They do not perceive any burglaries directly,
    - They do not notice the minor earthquakes, and

Downloaded by Viji Jai (nancymeena@gmail.com)

- They do not discuss before calling.
- Notice that the burglar alarm network does not have any nodes corresponding to
  - Mary is currently listening to loud music or
  - The telephone ringing and confusing john
- These factors are summarizing in the uncertainty, associated with the links from Alarm to JohnCalls and MaryCalls.
- This shows both laziness and ignorance in operation.

Conditional Probability Tables- CPT

- The conditional probability tables in the network give the probabilities for the values of the random variable depending on the combination of values for the parent nodes.
- Each row must sum to 1.
- All variables are Boolean, and therefore, the probability of a true value is p, the probability of false must be 1-p.
- A table for a Boolean variable with k parents contains 2^k independently specifiable probabilities.
- A variable with no parents has only one row, representing the prior probabilities of each possible value of the variable.

Semantics of Bayesian Networks

Before that, lets discuss about Joint Probability Distribution

- The full joint probability distribution specifies the probability of values to random variables.
- It is usually too large to create or use in its explicit form.
- Joint probability distribution of two variables X and Y are

Joint Probabilities	X	X'
Y	0.20	0.12
Y'	0.65	1.03

- Joint Probability distribution for n variables require 2^n entries with all possible combination.(entries also increased & this is the drawback of joint probability)

Drawbacks of joint probability distribution

i. Large number of variables and grows rapidly.
ii. Time and space complexity are huge.
iii. Statistical estimation with probability is difficult.
iv. Human tends signal out few propositions.
v. The alternative to this is Bayesian Networks.

Example:

- We can calculate the probability that the alarm has sounded, but neither a burglary nor an earthquake has occurred, and both John and Mary call.

$$\begin{aligned} P(j,m,a,\neg b,\neg e) &= P(j|a)P(m|a)P(a|\neg b \wedge \neg e)P(\neg b)P(\neg e) \\ &= 0.90 \times 0.70 \times 0.01 \times 0.999 \times 0.998 = 0.00628. \end{aligned}$$

Downloaded by Viji Jai (nancymeena@gmail.com)

The Semantics of Bayesian Networks

- An entry in joint distribution is the probability of conjunction of particular assignment to each variable, such as (here X random variables, x- values to random variables,$\pi \rightarrow product$ & here x_i is depending on the parent X_i value)

  $P(X_1 = x_1 \wedge \ldots \wedge X_n = x_n)$,

  which is equal to

$$P(x_1, \ldots, x_n) = \prod_{i=1}^{n} \theta(x_i | parents(X_i)),$$

<u>Method for constructing Bayesian Network</u>

- Rewrite the joint distribution in terms of a conditional probability, using the **product rule**

$$P(x_1, \ldots, x_n) = P(x_n | x_{n-1}, \ldots, x_1) P(x_{n-1}, \ldots, x_1).$$

- Then we repeat the process, reducing each conjunctive probability to a conditional probability and a smaller conjunction. We end up with one big product:

$$\begin{aligned} P(x_1, \ldots, x_n) &= P(x_n | x_{n-1}, \ldots, x_1) P(x_{n-1} | x_{n-2}, \ldots, x_1) \cdots P(x_2 | x_1) P(x_1) \\ &= \prod_{i=1}^{n} P(x_i | x_{i-1}, \ldots, x_1). \end{aligned}$$

- This identity is called the **chain rule**. The specification of the joint distribution is equivalent to the general assertion that, for every variable X_i in the network,

$$\mathbf{P}(X_i | X_{i-1}, \ldots, X_1) = \mathbf{P}(X_i | Parents(X_i)),$$

- We can directly implement this formula into our example

$$\mathbf{P}(MaryCalls | JohnCalls, Alarm, Earthquake, Burglary) = \mathbf{P}(MaryCalls | Alarm).$$

- No need to consider all the other things, because the parent of MaryCalls is Alarm. So, MaryCalls is the child node of Alarm.

<u>Compactness and node ordering</u>

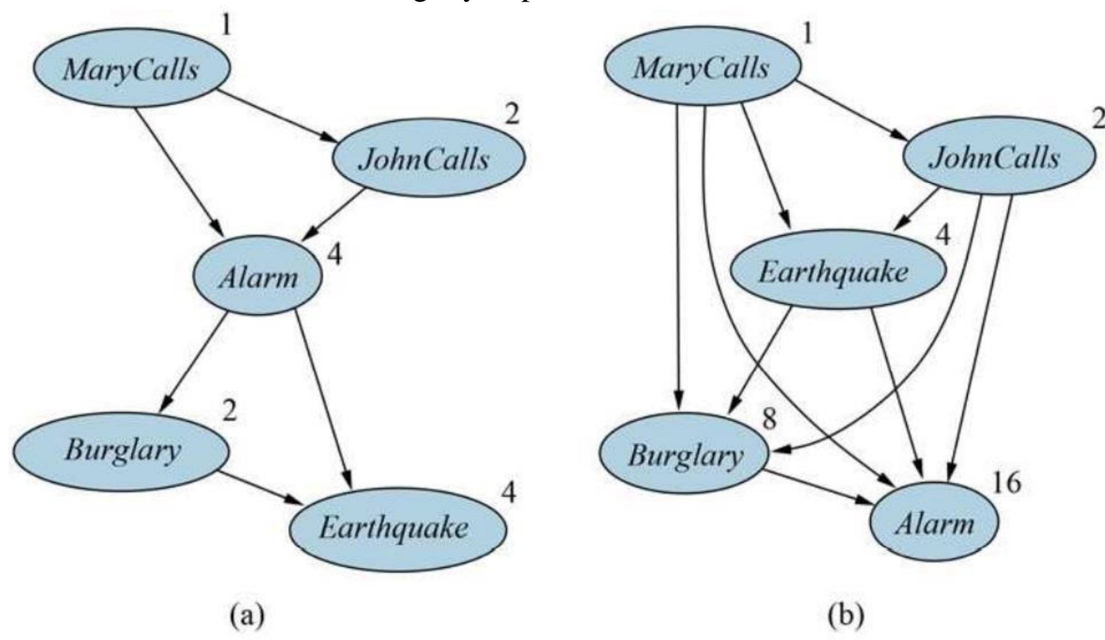
- The compactness of Bayesian network is an example of general property of locally constructed systems. (also called as spare systems, inside some components there, and those are communicated)
- In a locally structured system, each subcomponent interacts directly with only a bounded number of other components, regardless of the total number of components.

Downloaded by Viji Jai (nancymeena@gmail.com)

- Therefor the correct in which to add node is to add the ‘root causes’ first, then the variables they influenced and so on until we reach the leaves.
- Suppose, we decide to add the nodes in the order MaryCalls, JohnCalls,Alarm, Burglary, Earthquake.
- Adding MaryCalls: No parents
- Adding JohnCalls: If Mary calls, the probably means the alarm has gone off, which of course would make it more likely that John calls. Therfore, johnCalls needs MaryCalls as a parent.
- Adding Alarm: Clearly, if both call, it is more likely that the alarm has gone off than if just one or neither call, so we need both MaryCalls and JohnCalls as parents.
- Adding Burglary: If we know the alarm state, then the call from John or Mary might give us information about our phone ringing or Mary’s music, but not about burglary:

$$\mathbf{P}(Burglary|Alarm,JohnCalls,MaryCalls) = \mathbf{P}(Burglary|Alarm).$$

- Hence we need just Alarm as parent.
- Adding Earthquake: If the alarm is on, it is more likely that there has been an earthquake. But if we know that there has been a burglary, then that explains the alarm, and the probability of an earthquake would be only slightly above normal. Hence we need both Alarm and Burglary as parents.

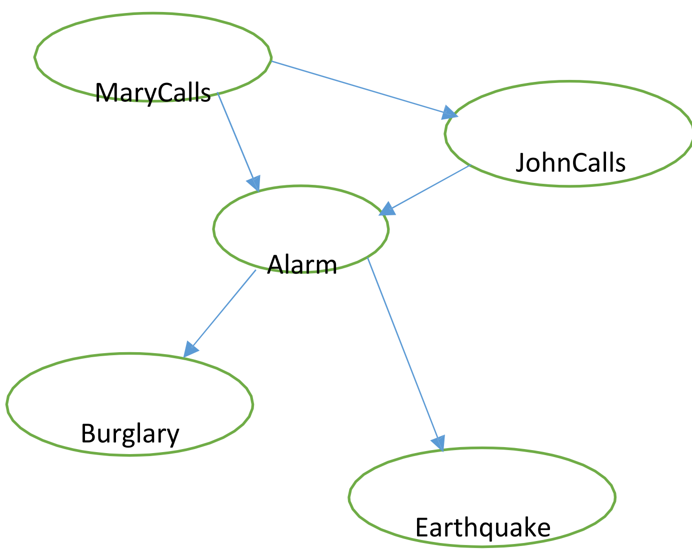


Network structure and number of parameters depends on order of introduction. (a) The structure obtained with ordering M,J,A,B,E. (b) The structure obtained with M,J,E,B,A. Each node is annotated with the number of parameters required; 13 in all for (a) and 31 for (b). In Figure 13.2, only 10 parameters were required.

Conditional independence relations in Bayesian networks

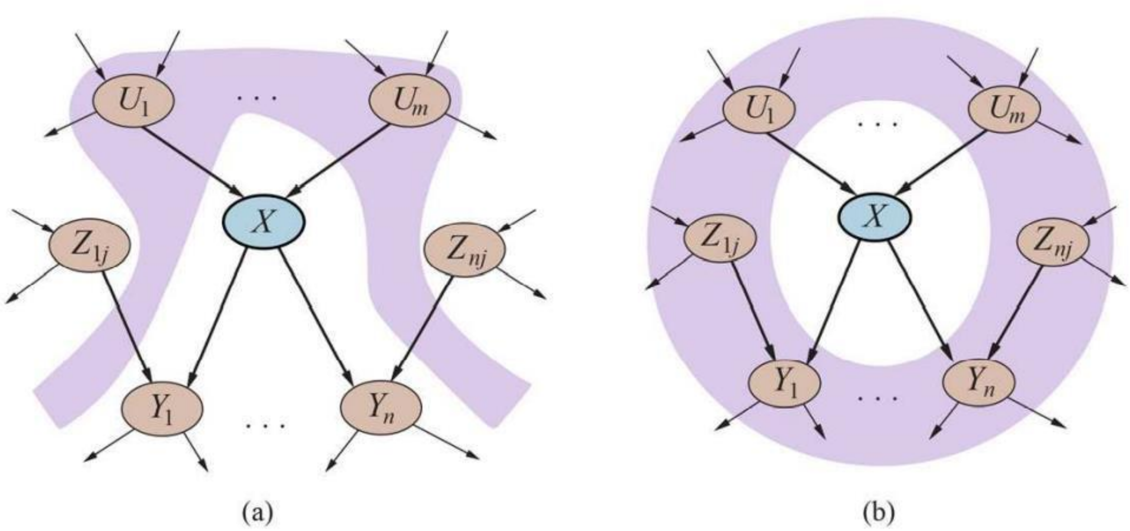
i. A node is conditionally independent of its non-descendants, given its parents.

- For example, JohnCalls is independent of Burglary and Earthquake, given the value of Alarm.



ii. A node is conditionally independent of all other nodes in the network, given its parents, children, and children's parents- that is, given its Markov blanket.

- For example, Burglary is independent of JohnCalls and MaryCalls, given Alarm and Earthquake.



(a) A node X is conditionally independent of its non-descendants (e.g., the Z_{ij}s) given its parents (the U_is shown in the gray area). (b) A node X is conditionally independent of all other nodes in the network given its Markov blanket (the gray area).

Exact Inference in Bayesian Network

- Probabilistic Inference System is to compute Posterior Probability Distribution for a set of query variables, given some observed events.
- That is, some assignment of values to a set of evidence variables.

<u>Notations</u>

- X denotes the query variable
- E set of evidence variables $\{E_1, \dots E_m\}$
- E particular observed event
- Y non-evidence, non-query variables, $Y_1, \dots Y_n$. (called the hidden variables)

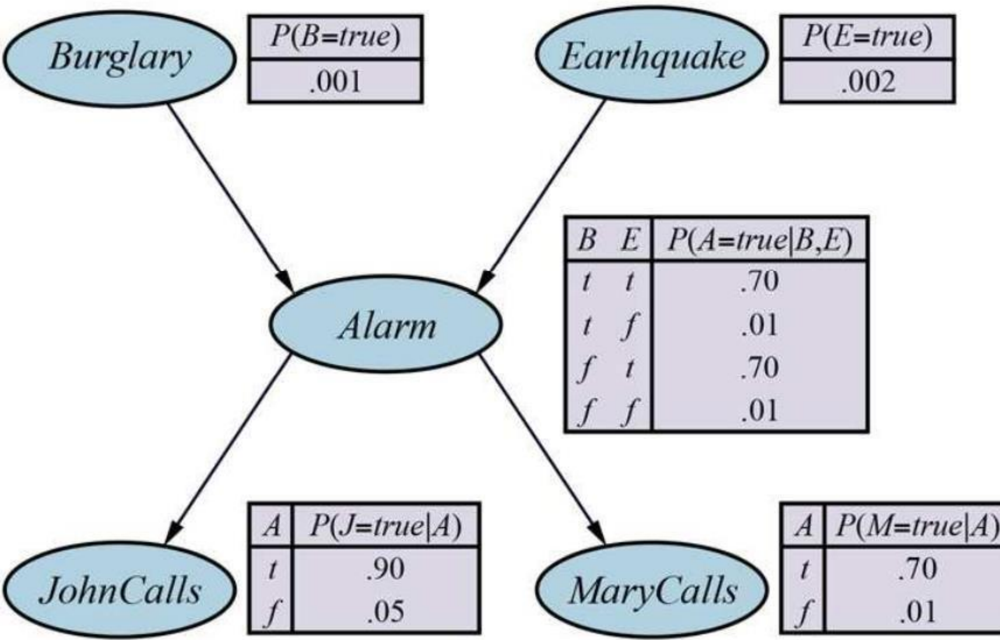
Downloaded by Viji Jai (nancymeena@gmail.com)

- The complete set of variables – $\boldsymbol{X} = \{\boldsymbol{X}\}\ \boldsymbol{U}\ \boldsymbol{E}\ \boldsymbol{U}\ \boldsymbol{Y}$
- A typical query asks for the Posterior Probability distribution **P(X | e)**
- In the burglary network, we might observe the event in which

$$JohnCalls = true \ and \ MaryCalls = true.$$

- We could then ask for, say, the probability that a burglary has occurred:

$$\mathbf{P}(Burglary|JohnCalls = true, MaryCalls = true) = \langle 0.284, 0.716 \rangle.$$



A typical Bayesian network, showing both the topology and the conditional probability tables (CPTs). In the CPTs, the letters *B*, *E*, *A*, *J*, and *M* stand for *Burglar*, *Earthquake*, *Alarm*, *JohnCalls*, and *MaryCalls*, respectively.

Types of Inferences

Inference by Enumeration (inference by listing or recording all variables)

- Any conditional probability can be computed by summing terms from the full joint distribution.
- More specifically, a query P(X | e) can be answered using equation.

$$\mathbf{P}(X|\mathbf{e}) = \alpha\, \mathbf{P}(X, \mathbf{e}) = \alpha \sum_{\mathbf{y}} \mathbf{P}(X, \mathbf{e}, \mathbf{y}).$$

- Where α is normalized constant
- X Query Variable
- e event
- y number of terms

Consider the query $\mathbf{P}(Burglary|JohnCalls = true, MaryCalls = true)$.

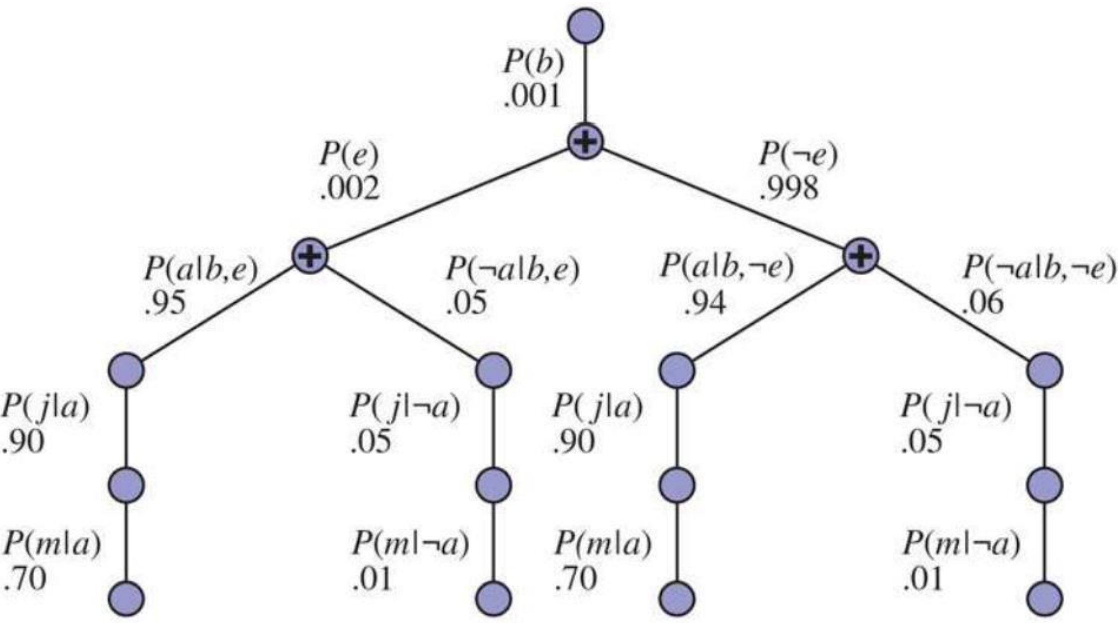
- Burglary query variable (X)
- JohnCalls Evidence variable 1 (E1)

 MaryCalls Evidence Variable 2 (E2)
 The hidden variables of this query are earthquake and alarm
 Using initial letter for the variables to shorten the expression we have

$$\mathbf{P}(B|j,m) = \alpha\,\mathbf{P}(B,j,m) = \alpha \sum_{e} \sum_{a} \mathbf{P}(B,j,m,e,a).$$

 The semantic of Bayesian network give us an expression, in terms of CPT entries, for simplicity we do this just for **Burglary** = **true**

$$P(b|j,m) = \alpha \sum_{e} \sum_{a} P(b)P(e)P(a|b,e)P(j|a)P(m|a).$$



The structure of the expression shown in Equation (13.5). The evaluation proceeds top down, multiplying values along each path and summing at the "+" nodes. Notice the repetition of the paths for j and m.

function ENUMERATION-ASK(X, $\mathbf{e}$, bn) **returns** a distribution over X
  inputs: X, the query variable
    $\mathbf{e}$, observed values for variables $\mathbf{E}$
    bn, a Bayes net with variables $vars$

  $\mathbf{Q}(X) \leftarrow$ a distribution over X, initially empty
  for each value x_i of X **do**
    $\mathbf{Q}(x_i) \leftarrow$ ENUMERATE-ALL($vars$, $\mathbf{e}_{x_i}$)
      where $\mathbf{e}_{x_i}$ is $\mathbf{e}$ extended with $X = x_i$
  return NORMALIZE($\mathbf{Q}(X)$)

function ENUMERATE-ALL($vars$, $\mathbf{e}$) **returns** a real number
  if EMPTY?($vars$) **then return** 1.0
  $V \leftarrow$ FIRST($vars$)
  if V is an evidence variable with value v in $\mathbf{e}$
    then return $P(v \mid parents(V)) \times$ ENUMERATE-ALL(REST($vars$), $\mathbf{e}$)
    else return $\sum_v P(v \mid parents(V)) \times$ ENUMERATE-ALL(REST($vars$), $\mathbf{e}_v$)
      where $\mathbf{e}_v$ is $\mathbf{e}$ extended with $V = v$

The enumeration algorithm for exact inference in Bayes nets.

Inference by Variable Elimination

- The enumeration algorithm can be improved substantially by elimination repeated calculations.
- The idea is simple: do the calculation once and solve the result for later use. This is a form of dynamic programming.
- Variable elimination works by evaluating expressions,
- Previous equation (derived in inference by enumeration)

$$P(b|j,m) = \alpha \sum_e \sum_a P(b)P(e)P(a|b,e)P(j|a)P(m|a).$$

- From this the repeated variables are separated

$$P(b|j,m) = \alpha\, P(b) \sum_e P(e) \sum_a P(a|b,e)P(j|a)P(m|a).$$

- Intermediate results are stored, and summations of each variable are done, for only those portion of the expression, that depends on the variable.
- Let us illustrate this process for the burglary network.
- We evaluate the expression

$$\mathbf{P}(B|j,m) = \alpha \underbrace{\mathbf{P}(B)}_{\mathbf{f}_1(B)} \sum_e \underbrace{P(e)}_{\mathbf{f}_2(E)} \sum_a \underbrace{\mathbf{P}(a|B,e)}_{\mathbf{f}_3(A,B,E)} \underbrace{P(j|a)}_{\mathbf{f}_4(A)} \underbrace{P(m|a)}_{\mathbf{f}_5(A)}.$$

- We have annotated each part of the expression with the same name of the **associated variable**, these parts are called **factors**
- For example, the factors $f4(A)$ and $f5(A)$ corresponding to $P(j \mid a)$ and $P(m \mid a)$ depending just on A because J and M are fixed by the query.
- They are therefore two element vectors.

$$\mathbf{f}_4(A) = \begin{pmatrix} P(j|a) \\ P(j|\neg a) \end{pmatrix} = \begin{pmatrix} 0.90 \\ 0.05 \end{pmatrix} \qquad \mathbf{f}_5(A) = \begin{pmatrix} P(m|a) \\ P(m|\neg a) \end{pmatrix} = \begin{pmatrix} 0.70 \\ 0.01 \end{pmatrix}.$$

- Given two factors $f(X, Y)$ and $g(Y, Z)$ with probability distributions shown below, the pointwise product $f \times g = h(X, Y, Z)$ has $2^{1+1+1} = 8$

X	Y	$\mathbf{f}(X,Y)$	Y	Z	$\mathbf{g}(Y,Z)$	X	Y	Z	$\mathbf{h}(X,Y,Z)$
t	t	.3	t	t	.2	t	t	t	$.3 \times .2 = .06$
t	f	.7	t	f	.8	t	t	f	$.3 \times .8 = .24$
f	t	.9	f	t	.6	t	f	t	$.7 \times .6 = .42$
f	f	.1	f	f	.4	t	f	f	$.7 \times .4 = .28$
						f	t	t	$.9 \times .2 = .18$
						f	t	f	$.9 \times .8 = .72$
						f	f	t	$.1 \times .6 = .06$
						f	f	f	$.1 \times .4 = .04$

Illustrating pointwise multiplication: $\mathbf{f}(X,Y) \times \mathbf{g}(Y,Z) = \mathbf{h}(X,Y,Z)$.

```
function ELIMINATION-ASK(X, e, bn) returns a distribution over X
  inputs: X, the query variable
          e, observed values for variables E
          bn, a Bayesian network with variables vars

  factors ← []
  for each V in ORDER(vars) do
     factors ← [MAKE-FACTOR(V, e)] + factors
     if V is a hidden variable then factors ← SUM-OUT(V, factors)
  return NORMALIZE(POINTWISE-PRODUCT(factors))
```

The variable elimination algorithm for exact inference in Bayes nets.

<u>Elimination</u>

- Summing out, or eliminating a variable from a factor is done by adding up the sub-arrays formed by fixing the variable to each of its values in turn.
- For example, to sum out a from $h(X, Y, Z)$, we write:

$$\mathbf{h}_2(Y,Z) = \sum_x \mathbf{h}(X,Y,Z) = \mathbf{h}(x,Y,Z) + \mathbf{h}(\neg x,Y,Z)$$
$$= \begin{pmatrix} .06 & .24 \\ .42 & .28 \end{pmatrix} + \begin{pmatrix} .18 & .72 \\ .06 & .04 \end{pmatrix} = \begin{pmatrix} .24 & .96 \\ .48 & .32 \end{pmatrix}.$$

Relevance

Let us consider one more query: $\mathbf{P}(JohnCalls|Burglary = true)$.

$$P(J|b)$$

$$\mathbf{P}(J|b) = \alpha\, P(b) \sum_e P(e) \sum_a P(a|b,e)\mathbf{P}(J|a) \sum_m P(m|a).$$

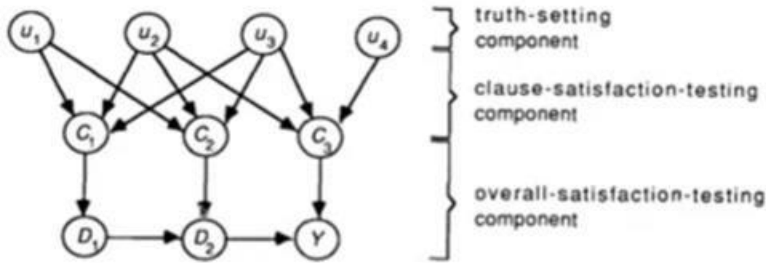
- $\Sigma_m P(m|a) = 1$, therefore M is irrelevant for the query.
- In other words, $P(J|b)$ remains unchanged if we remove M from the network.

Complexity

The computational and space complexity of variable elimination is determined by the largest factor.

- The elimination ordering can greatly affect the size of the largest factor.
- Does there always exist an ordering that only results in small factors? No!
  - Greedy heuristic: eliminate whichever variable minimizes the size of the factor to be constructed.
  - Singly connected networks (polytrees):
    - Any two nodes are connected by at most one (undirected path).
    - For these networks, time and space complexity of variable elimination are $O(nd^k)$.

Worst-case complexity?



3SAT is a special case of inference:

- CSP: $(u_1 \vee u_2 \vee u_3) \wedge (\neg u_1 \vee \neg u_2 \vee u_3) \wedge (u_2 \vee \neg u_3 \vee u_4)$
- $P(U_i = 0) = P(U_i = 1) = 0.5$
- $C_1 = U_1 \vee U_2 \vee U_3; C_2 = \neg U_1 \vee \neg U_2 \vee U_3; C_3 = U_2 \vee \neg U_3 \vee U_4$
- $D_1 = C_1; D_2 = D_1 \wedge C_2$
- $Y = D_2 \wedge C_3$

- If we answer whether P(Y=1)>0, then we answer whether 3SAT has a solution.
  - By reduction, inference in Bayesian networks is therefore NP-complete.
  - There is no known efficient probabilistic inference algorithm in general.

Approximate inference

- Exact inference is intractable for most probabilistic models of practical interest. e.g) involving many variables, continuous and discrete, undirected cycles, etc.

<u>Sampling Methods</u>

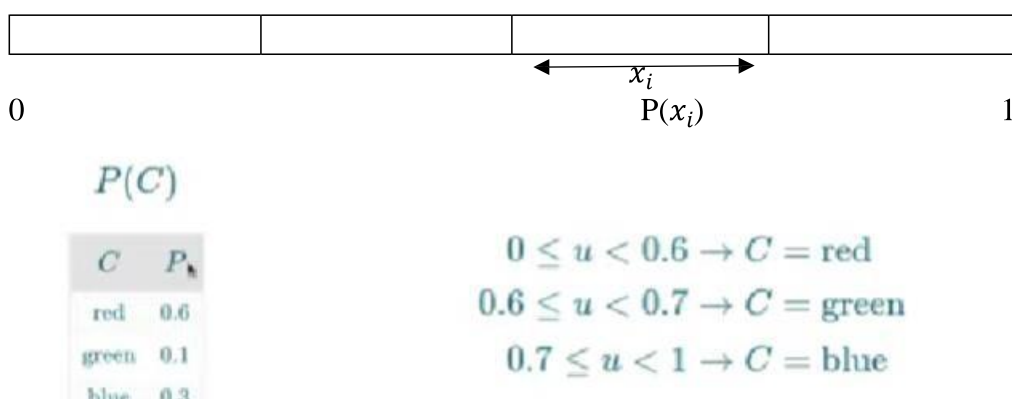
- Basic idea:
  - Draw N samples from a sampling distribution S.
  - Compute an approximate posterior probability P.
  - Show this approximate coverages to the true probability distribution P.

Why sampling

- Generating samples is often much faster than computing the right answer (e.g., with variable elimination)

Sampling

- How to sample from the distribution of a discrete variable X?
  - Assume k discrete outcomes $x_{1,...}x_k$ with probability P(x_i)
  - Assume sampling from the uniform $U(0,1)$ is possible. e.g) as enabled by a standard r and () function.
  - Divide the [0,1] interval into k regions, with region i having size P(x_i).
  - Sample $u \sim U(0,1)$ and return the value associated to the region in which u falls.



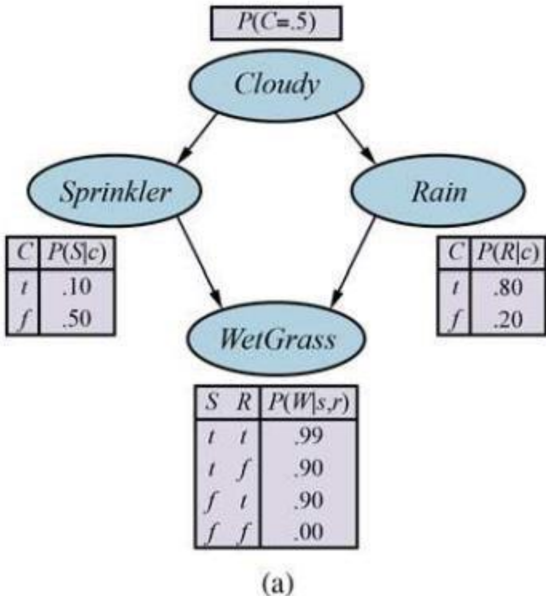
<u>Prior Sampling</u>

- Sampling from a Bayesian network, without observed evidence.
  - Sample each variable in turn, in topological order.
  - The probability distribution from which the value is sampled is conditioned on the values already assigned to the variable's parents.

function PRIOR-SAMPLE(bn) **returns** an event sampled from the prior specified by bn
  inputs: bn, a Bayesian network specifying joint distribution $\mathbf{P}(X_1,\ldots,X_n)$

  $\mathbf{x} \leftarrow$ an event with n elements
  for each variable X_i **in** $X_1,\ldots,X_n$ **do**
    $\mathbf{x}[i] \leftarrow$ a random sample from $\mathbf{P}(X_i \mid parents(X_i))$
  return x

A sampling algorithm that generates events from a Bayesian network. Each variable is sampled according to the conditional distribution given the values already sampled for the variable's parents.



(a)

Downloaded by Viji Jai (nancymeena@gmail.com)

Example

We will collect a bunch of samples from the Bayesian network:

$c, \neg s, r, w$
c, s, r, w
$\neg c, s, r, \neg w$
$c, \neg s, r, w$
$\neg c, \neg s, \neg r, w$

If we want to know $\mathbf{P}(W)$:

- We have counts $\langle w : 4, \neg w : 1 \rangle$
- Normalize to obtain $\hat{\mathbf{P}}(W) = \langle w : 0.8, \neg w : 0.2 \rangle$
- $\hat{\mathbf{P}}(W)$ will get closer to the true distribution $\mathbf{P}(W)$ as we generate more samples.

Analysis

- The probability that prior sampling generates a particular event is

$$S_{PS}(x_1 \ldots x_n) = \prod_{i=1}^{n} P(x_i | parents(X_i))$$

$$S_{PS}(x_1 \ldots x_n) = P(x_1 \ldots x_n).$$

i.e) the Bayesian network's joint probability

- Let $N_{PS}(x_{1,..}x_n)$ denote the number of samples of an event. We define the probability estimator

$$\hat{P}(x_1, .. x_n) = N_{PS}(x_1, \ldots x_n)/N$$

Then,

$$\lim_{N \to \infty} \frac{N_{PS}(x_1, \ldots, x_n)}{N} = S_{PS}(x_1, \ldots, x_n) = P(x_1, \ldots, x_n).$$

Therefore, prior sampling is consistent:

$$P(x_1, \ldots, x_m) \approx N_{PS}(x_1, \ldots, x_m)/N.$$

Rejection sampling in Bayesian networks

- Using prior sampling, an estimate $\hat{P}(x|e)$ can be formed from the proportion of samples x agreeing with the evidence e among all samples agreeing with the evidence.

```
function REJECTION-SAMPLING(X, e, bn, N) returns an estimate of P(X | e)
   inputs: X, the query variable
           e, observed values for variables E
           bn, a Bayesian network
           N, the total number of samples to be generated
   local variables: C, a vector of counts for each value of X, initially zero

   for j = 1 to N do
      x ← PRIOR-SAMPLE(bn)
      if x is consistent with e then
         C[j] ← C[j]+1 where x_j is the value of X in x
   return NORMALIZE(C)
```

The rejection-sampling algorithm for answering queries given evidence in a Bayesian network.

Analysis

- Let consider the posterior probability estimator $\hat{P}(X|e)$ formed by rejection sampling:

$$\hat{\mathbf{P}}(X|\mathbf{e}) = \alpha \mathbf{N}_{PS}(X,\mathbf{e}) = \frac{\mathbf{N}_{PS}(X,\mathbf{e})}{N_{PS}(\mathbf{e})}.$$

$$\begin{aligned} \hat{P}(x|e) &= N_{PS}(x,e)/N_{PS}(e) \\ &= \frac{N_{PS}(x,e)}{N} / \frac{N_{PS}(e)}{N} \\ &\approx P(x,e)/P(e) \\ &= P(x|e) \end{aligned}$$

this becomes

$$\hat{\mathbf{P}}(X|\mathbf{e}) \approx \frac{\mathbf{P}(X,\mathbf{e})}{P(\mathbf{e})} = \mathbf{P}(X|\mathbf{e}).$$

- Therefore, rejection sampling is consistent.
- The standard deviation of the error in each probability is $O(1/\sqrt{n})$, where n is the number of samples used to compute the estimate.
- Problem: many samples are rejected!
  - Hopelessly expensive if the evidence is unlikely. i.e if P(e) is small.
  - Evidence is not exploited when sampling.

Likelihood weighting

- Idea: clamp the evidence variables, sample the rest.
  - Problem: the resulting sampling distribution is not consistent.
  - Solution: weight by probability of evidence given parents.

Downloaded by Viji Jai (nancymeena@gmail.com)

```
function LIKELIHOOD-WEIGHTING(X, e, bn, N) returns an estimate of P(X | e)
  inputs: X, the query variable
          e, observed values for variables E
          bn, a Bayesian network specifying joint distribution P(X_1, ..., X_n)
          N, the total number of samples to be generated
  local variables: W, a vector of weighted counts for each value of X, initially zero

  for j = 1 to N do
     x, w ← WEIGHTED-SAMPLE(bn, e)
     W[j] ← W[j] + w where x_j is the value of X in x
  return NORMALIZE(W)

function WEIGHTED-SAMPLE(bn, e) returns an event and a weight
  w ← 1; x ← an event with n elements, with values fixed from e
  for i = 1 to n do
     if X_i is an evidence variable with value x_ij in e
        then w ← w × P(X_i = x_ij | parents(X_i))
        else x[i] ← a random sample from P(X_i | parents(X_i))
  return x, w
```

The likelihood-weighting algorithm for inference in Bayesian networks. In WEIGHTED-SAMPLE, each nonevidence variable is sampled according to the conditional distribution given the values already sampled for the variable's parents, while a weight is accumulated based on the likelihood for each evidence variable.

Analysis

- The sampling probability for an event with likelihood weighting is

$$S_{WS}(x, e) = \prod_{i=1}^{l} P(x_i|parents(X_i))$$

- Where the product is over the non-evidence variables. The weight for a given sample x, e is

$$w(x, e) = \prod_{i=1}^{m} P(e_i|parents(E_i))$$

- Where the product is over the evidence variables.
- The weighted sampling probability is

$$S_{WS}(x, e)w(x, e) = \prod_{i=1}^{l} P(x_i|parents(X_i)) \prod_{i=1}^{m} P(e_i|parents(E_i))$$

$$= P(x, e)$$

- The estimated posterior probability is computed as follows:

$$\hat{P}(x, e) = \alpha N_{WS}(x, e)\omega(x, e)$$

$$\approx \alpha' S_{WS}(x, e)\omega(x, e)$$

$$= \alpha' P(x, e)$$

$$= P(x|e)$$

- Where α and α' are normalization constants.
- Hence likelihood weighting returns consistent estimates.
- Likelihood weighting is helpful:
    - The evidence is taken into account to generate a sample.
    - More samples will reflect the state of the world suggested by the evidence.
- Likelihood weighting does not solve all problems:
    - Performance degrades as the number of evidence variable increases.
    - The evidence influences the choice of downstream variables, but not upstream ones.
        - Ideally, we would like to consider the evidence when we sample each and every variable.

Inference by Markov chain simulation

- Markov chain Monte Carlo (MCMC) algorithms are a family of sampling algorithms that generate samples through a Markov chain.
- They generate a sequence of samples by making random changes to a preceding sample, instead of generating each sample from scratch.
- Helpful to think of a Bayesian network as being in a particular current state specifying a value for each variable and generating a next state by making random changes to the current state.
- Metropolis-Hastings is one of the most famous MCMC methods, of which Gibbs sampling is a special case.

Gibbs sampling

- Start with an arbitrary instance x_1, ... x_n consistent with the evidence.
- Sample one variable at a time, conditioned on all the rest, but keep the evidence fixed.
- Keep repeating this for a long time.

function GIBBS-ASK(X, **e**, bn, N) **returns** an estimate of $\mathbf{P}(X \mid \mathbf{e})$
  local variables: **C**, a vector of counts for each value of X, initially zero
    Z, the nonevidence variables in bn
    x, the current state of the network, initialized from **e**

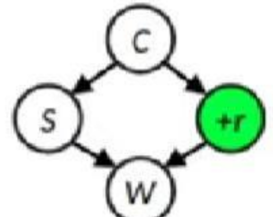
  initialize **x** with random values for the variables in **Z**
  for k = 1 **to** N **do**
    choose any variable Z_i from **Z** according to any distribution $\rho(i)$
    set the value of Z_i in **x** by sampling from $\mathbf{P}(Z_i \mid mb(Z_i))$
    $\mathbf{C}[j] \leftarrow \mathbf{C}[j] + 1$ where x_j is the value of X in **x**
  return NORMALIZE(**C**)

The Gibbs sampling algorithm for approximate inference in Bayes nets; this version chooses variables at random, but cycling through the variables but also works.

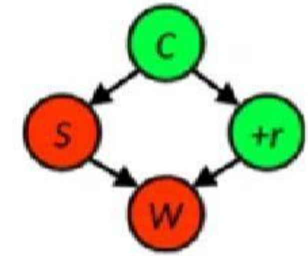
Downloaded by Viji Jai (nancymeena@gmail.com)

Example

1) Fix the evidence

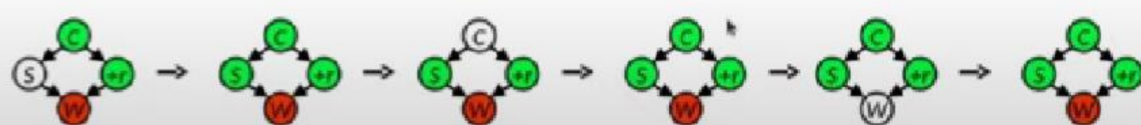


2) Randomly initialize the other variables



3) Repeat

- Choose a non-evidence variable X.
- Resample X from $\mathbf{P}(X|\text{all other variables})$.



UNIT IV ENSEMBLE TECHNIQUES AND UNSUPERVISED LEARNING

Combining multiple learners: Model combination schemes, Voting, Ensemble Learning - bagging, boosting, stacking, Unsupervised learning: K-means, Instance Based Learning: KNN, Gaussian mixture models and Expectation maximization

1. Combining multiple learners

- Though different learning algorithms are generally successful, no one single algorithm is always the most accurate.
- The models composed of multiple learners that complement each other so that by combining them, we attain higher accuracy.
- Each learning algorithm dictates a certain model that comes with a set of assumptions.
- By suitably combining multiple base learners then, accuracy can be improved.

There are basically two questions here:
1. How do we generate base-learners that complement each other?
2. How do we combine the outputs of base-learners for maximum accuracy?

1.1. Generating Diverse Learners

- Different Algorithms
- We can use different learning algorithms to train different base-learners.
- For example, one base-learner may be parametric and another may be nonparametric.

Different Hyperparameters

- We can use the same learning algorithm but use it with different hyperparameters.
- Examples are the number of hidden units in a multilayer perceptron, k in k-nearest neighbor, error threshold in decision trees, the kernel function in support vector machines, and so forth.

Different Input Representations

- Separate base-learners may be using different representations of the same input object or event, making it possible to integrate different types of sensors/measurements/modalities.
- Sensor fusion where the data from different sensors are integrated to extract more information for a specific application.
- To Combine more sources to find the right set of images; this is also sometimes called
- multi-view learning multi-view learning.
- Even if there is a single input representation, by choosing random subsets
- from it, we can have classifiers using different input features; this is called the random subspace method.

Different Training Sets

- Another possibility is to train different base-learners by different subsets of the training set.
- This can be done randomly by drawing random training sets from the given sample; this is called bagging.
- Or, the learners can be trained serially. Examples are boosting and cascading

2. Model Combination Schemes

- There are also different ways the multiple base-learners are combined to generate the final output:
- Multiexpert combination methods have base-learners that work in parallel. These methods can in turn be divided into two:

- Global Approach : In the global approach, also called learner fusion, given an input, all base-learners generate an output and all these outputs are used.
- Examples are voting and stacking.
- Local Approach : In the local approach, or learner selection, for example, in mixture of experts, there is a gating model, which looks at the input and
- chooses one (or very few) of the learners as responsible for generating the output.
- Multistage combination methods use a serial approach where the next combination base-learner is trained with or tested on only the instances where the previous base-learners are not accurate enough.
- Let us say that we have L base-learners.
- We denote by dj(x) the prediction of base-learner Mj given the arbitrary dimensional input x.

- In the case of multiple representations, each Mj uses a different input representation
- xj .
- The final prediction is calculated from the predictions of the base-learners:

$$y = f(d_1, d_2, \ldots, d_L | \Phi)$$

- where f (·) is the combining function with Φ denoting its parameters.
- When there are K outputs, for each learner there are dji(x), i = 1, . . . , K, j = 1, . . . , L, and, combining them, we also generate K values, yi, i =1, . . . , K and then for example in classification, we choose the class with the maximum yi value:

$$\text{Choose } C_i \text{ if } y_i = \max_{k=1}^{K} y_k$$

Figure 17.1 Base-learners are d_j and their outputs are combined using $f(\cdot)$. This is for a single output; in the case of classification, each base-learner has K outputs that are separately used to calculate y_i, and then we choose the maximum. Note that here all learners observe the same input; it may be the case that different learners observe different representations of the same input object or event.

3. Voting

- The simplest way to combine multiple classifiers is by voting, which corresponds
- to taking a linear combination of the learners.
- This is also known as ensembles and linear opinion pools.
- Linear Option Pools : In the simplest case, all learners are given equal weight and we have simple voting that corresponds to taking an average.

Table 17.1 Classifier combination rules

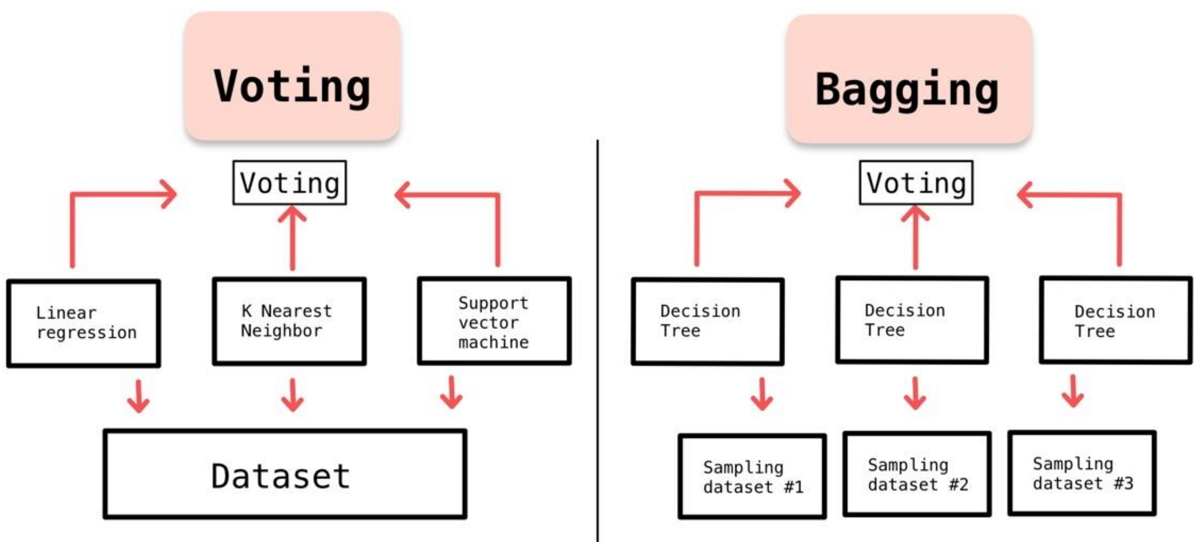
Rule	Fusion function $f(\cdot)$
Sum	$y_i = \frac{1}{L}\sum_{j=1}^{L} d_{ji}$
Weighted sum	$y_i = \sum_j w_j d_{ji}, w_j \geq 0, \sum_j w_j = 1$
Median	$y_i = \text{median}_j d_{ji}$
Minimum	$y_i = \min_j d_{ji}$
Maximum	$y_i = \max_j d_{ji}$
Product	$y_i = \prod_j d_{ji}$

Table 17.2 Example of combination rules on three learners and three classes

	C_1	C_2	C_3
d_1	0.2	0.5	0.3
d_2	0.0	0.6	0.4
d_3	0.4	0.4	0.2
Sum	0.2	**0.5**	0.3
Median	0.2	**0.5**	0.4
Minimum	0.0	**0.4**	0.2
Maximum	0.4	**0.6**	0.4
Product	0.0	**0.12**	0.032

Rules :

- Median rule is more robust to outliers; minimum and maximum rules are pessimistic and optimistic, respectively.
- With the product rule, each learner has veto power; regardless of the other ones, if one learner has an output of 0, the overall output goes to 0.
- Note that after the combination rules, yi do not necessarily sum up to 1.



- In weighted sum, dji is the vote of learner j for class Ci and wj is the weight of its vote. Simple voting is a special case where all voters have equal weight, namely, wj = 1/L.
- Plurality Voting: Plurality voting where the class having the maximum number of votes is the winner.
- When there are two classes, this is majority voting where the winning class gets more than half of the votes.
- Weighted Voting Scheme : If the voters can also supply the additional information of how much they vote for each class, then after normalization, these can be used as weights in a weighted voting scheme.
- In the case of regression, simple or weighted averaging or median can be used to fuse the outputs of base-regressors.

Bayesian combination model :

- Voting schemes can be seen as approximations under a Bayesian framework with weights approximating prior model probabilities, and model decisions approximating model-conditional likelihoods.

$$P(C_i|x) = \sum_{\text{all models } \mathcal{M}_j} P(C_i|x, \mathcal{M}_j)P(\mathcal{M}_j)$$

- We only choose a subset for which we believe P(Mj) is high, or we can have another Bayesian step and calculate P(Mj|X), the probability of a model given the sample, and sample high probable models from this density.

- The independent two class classifiers with success probability higher than 1/2, namely, better than random guessing, by taking a majority vote, the accuracy increases as the number of voting classifiers increases.
- Let us assume that dj are iid with expected value E[dj] and variance Var(dj), then when we take a simple average with wj = 1/L, the expected value and variance of the output are

$$E[y] = E\left[\sum_j \frac{1}{L} d_j\right] = \frac{1}{L} L E[d_j] = E[d_j]$$

$$\text{Var}(y) = \text{Var}\left(\sum_j \frac{1}{L} d_j\right) = \frac{1}{L^2}\text{Var}\left(\sum_j d_j\right) = \frac{1}{L^2} L \text{Var}(d_j) = \frac{1}{L}\text{Var}(d_j)$$

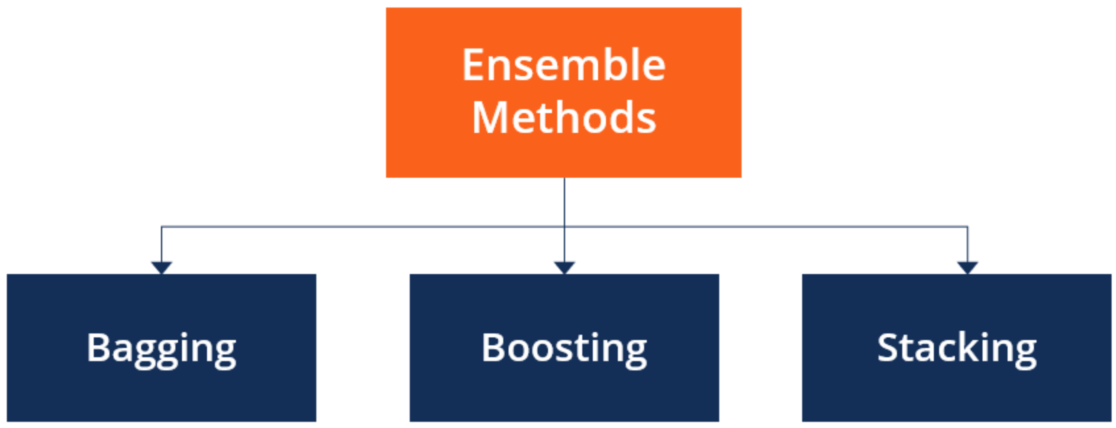
- We see that the expected value does not change, so the bias does not change.
- But variance, and therefore mean square error, decreases as the number of independent voters, L, increases.

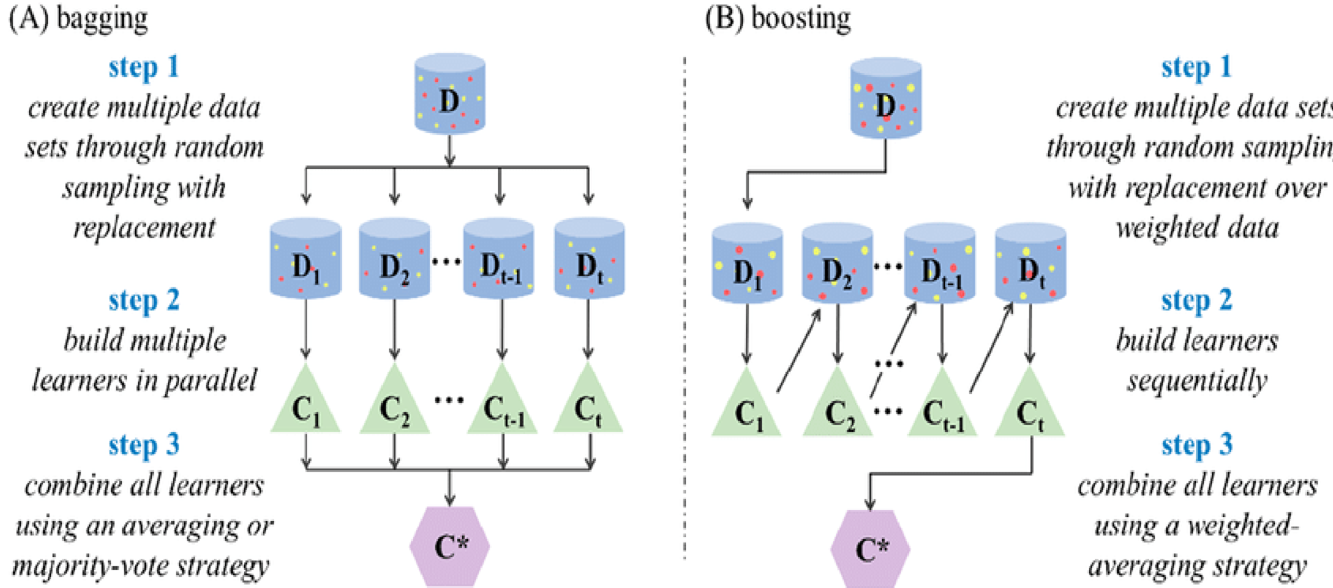
$$\mathrm{Var}(y) = \frac{1}{L^2}\mathrm{Var}\left(\sum_j d_j\right) = \frac{1}{L^2}\left[\sum_j \mathrm{Var}(d_j) + 2\sum_j \sum_{i<j} \mathrm{Cov}(d_j, d_i)\right]$$

- The further decrease in variance is possible if the voters are not independent but negatively correlated.
- The error then decreases if the accompanying increase in bias is not higher.
- Voting has the effect of smoothing in the functional space and can be thought of as a regularizer with a smoothness assumption on the true function.
- We vote over models with high variance and low bias so that after combination, the bias remains small and we reduce the variance by averaging.

4. Ensemble Learning

- Ensemble methods are techniques that aim at improving the accuracy of results in models by combining multiple models instead of using a single model.
- The combined models increase the accuracy of the results significantly.
- This has boosted the popularity of ensemble methods in machine learning.
- Ensemble methods fall into two broad categories, i.e., sequential ensemble techniques and parallel ensemble techniques.
- Sequential ensemble techniques generate base learners in a sequence, e.g., Adaptive Boosting (AdaBoost).
- The sequential generation of base learners promotes the dependence between the base learners.
- The performance of the model is then improved by assigning higher weights to previously misrepresented learners.
- In parallel ensemble techniques, base learners are generated in a parallel format, e.g., random forest.
- Parallel methods utilize the parallel generation of base learners to encourage independence between the base learners.
- The independence of base learners significantly reduces the error due to the application of averages.



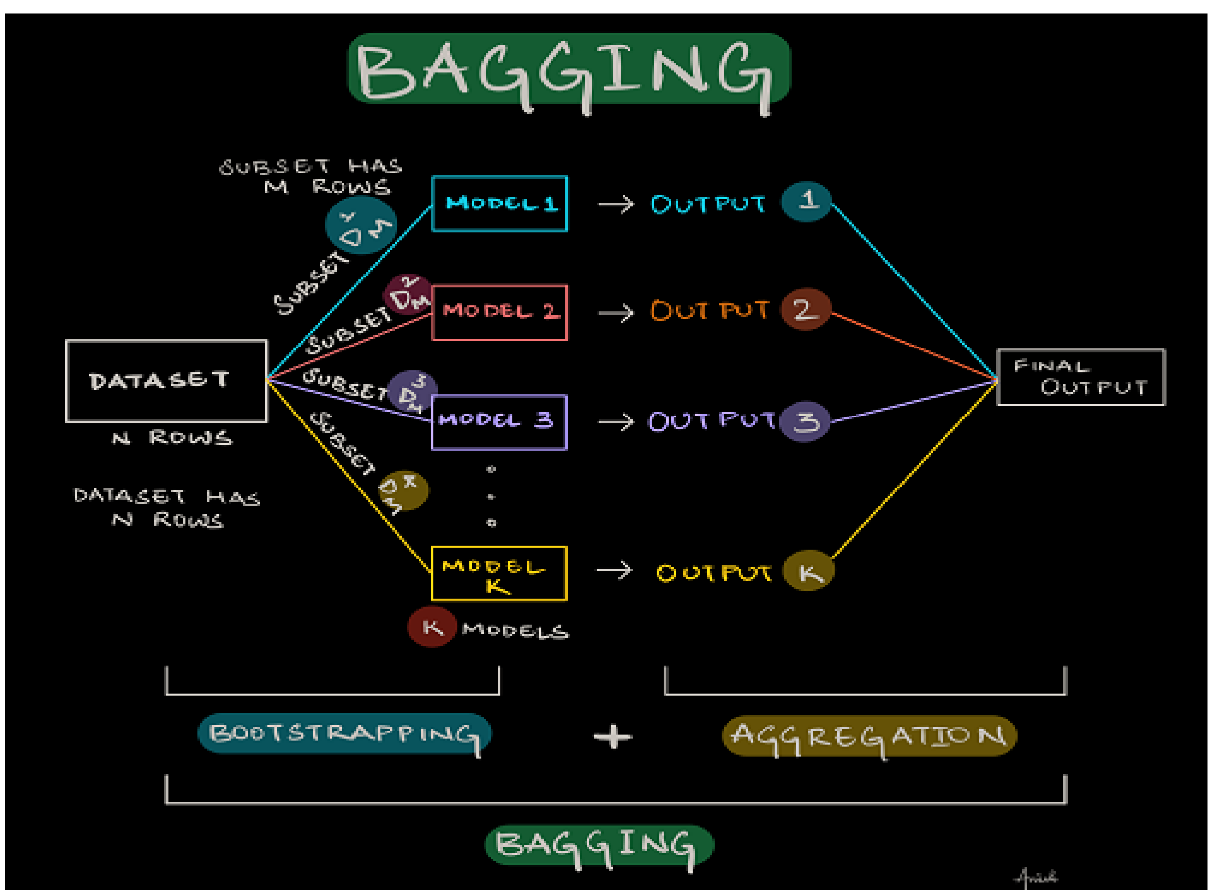


5. Bagging:

- Bagging is a voting method whereby base-learners are made different by training them over slightly different training sets.
- Generating L slightly different samples from a given sample is done by bootstrap, where given a training set X of size N, we draw N instances randomly from X with replacement.
- It is possible that some instances are drawn more than once and that certain instances
- are not drawn at all.
- The samples are similar because they are all drawn from the same original sample, but they are also slightly different due to chance.
- The base-learners dj are trained with these L samples Xj .

Unstable algorithm:

- A learning algorithm is an unstable algorithm if small changes in the training set causes a large difference in the generated learner.
- Bagging - bootstrap aggregating:
  - Uses bootstrap to generate L training sets, trains L base-learners using an unstable learning procedure,
  - During testing, takes an average.
- Bagging can be used both for classification and regression.
- Regression - Can take the median instead of the average when combining predictions.
- An algorithm is stable if different runs of the same algorithm on resampled versions of the same dataset lead to learners with
- high positive correlation.
- Algorithms such as decision trees and multilayer perceptrons are unstable.
- Nearest neighbor is stable, but condensed nearest neighbor is unstable.
- If the original training set is large, then we may want to generate smaller sets of size N' < N from them using bootstrap, since otherwise the bootstrap replicates Xj will be
- too similar, and dj will be highly correlated.



6. Boosting:

- In boosting, we try to generate complementary base-learners by training the next learner
- on the mistakes of the previous learners.
- The original boosting algorithm combines three weak learners to generate a strong
- learner.

Weak learner:

  - A weak learner has error probability less than 1/2, which makes it better than random guessing on a two-class problem, and a strong learner has arbitrarily small error probability.

- Given a large training set, we randomly divide it into three.
- We use X_1 and train d_1. We then take X_2 and feed it to d_1.
- We take all instances misclassified by d_1 and also as many instances on which d_1 is correct
- from X_2, and these together form the training set of d_2.
- We then take X_3 and feed it to d_1 and d_2.
- The instances on which d_1 and d_2 disagree form the training set of d_3.
- During testing, given an instance, we give it to d_1 and d_2; if they agree, that is the response, otherwise the response of d_3 is taken as the output.
- The overall system has reduced error rate.
- The error rate can arbitrarily be reduced by using systems recursively, that is, a boosting system of three models used as d_j in a higher system.

Disadvantage :

- It requires a very large training sample.

- The sample should be divided into three and furthermore, the second and third classifiers

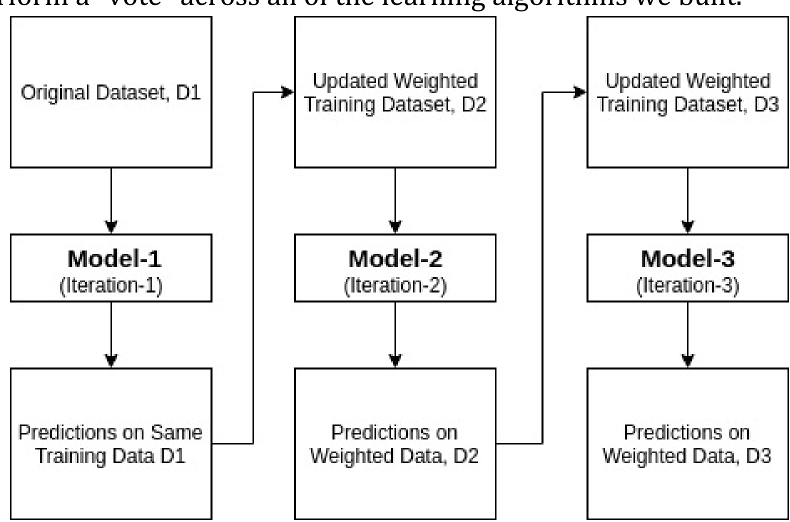
- are only trained on a subset on which the previous ones.

AdaBoost:

- Adaptive boosting - that uses the same training set over and over and thus need not be large, but the classifiers should be simple so that they do not overfit.
- AdaBoost can also combine an arbitrary number of base learners, not three.

It works in the following steps:

- Initially, Adaboost selects a training subset randomly.
- It iteratively trains the AdaBoost machine learning model by selecting the training set based on the accurate prediction of the last training.
- It assigns the higher weight to wrong classified observations so that in the next iteration these observations will get the high probability for classification.
- Also, It assigns the weight to the trained classifier in each iteration according to the accuracy of the classifier. The more accurate classifier will get high weight.
- This process iterate until the complete training data fits without any error or until reached to the specified maximum number of estimators.
- To classify, perform a "vote" across all of the learning algorithms we built.



Training:
  For all $\{x^t, r^t\}_{t=1}^N \in \mathcal{X}$, initialize $p_1^t = 1/N$
  For all base-learners $j = 1, \ldots, L$
    Randomly draw $\mathcal{X}_j$ from $\mathcal{X}$ with probabilities p_j^t
    Train d_j using $\mathcal{X}_j$
    For each (x^t, r^t), calculate $y_j^t \leftarrow d_j(x^t)$
    Calculate error rate: $\epsilon_j \leftarrow \sum_t p_j^t \cdot 1(y_j^t \neq r^t)$
    If $\epsilon_j > 1/2$, then $L \leftarrow j - 1$; stop
    $\beta_j \leftarrow \epsilon_j / (1 - \epsilon_j)$
    For each (x^t, r^t), decrease probabilities if correct:
      If $y_j^t = r^t$, then $p_{j+1}^t \leftarrow \beta_j p_j^t$ Else $p_{j+1}^t \leftarrow p_j^t$
    Normalize probabilities:
      $Z_j \leftarrow \sum_t p_{j+1}^t; \;\; p_{j+1}^t \leftarrow p_{j+1}^t / Z_j$
Testing:
  Given x, calculate $d_j(x), j = 1, \ldots, L$
  Calculate class outputs, $i = 1, \ldots, K$:
    $y_i = \sum_{j=1}^L \left(\log \frac{1}{\beta_j}\right) d_{ji}(x)$

Figure 17.2 AdaBoost algorithm.

Many variants of AdaBoost have been proposed; here, we discuss the original algorithm AdaBoost.M1 (see figure 17.2). The idea is to modify the probabilities of drawing the instances as a function of the error. Let us say p_j^t denotes the probability that the instance pair (x^t, r^t) is drawn to train the jth base-learner. Initially, all $p_1^t = 1/N$. Then we add new

base-learners as follows, starting from $j = 1$: ϵ_j denotes the error rate of d_j. AdaBoost requires that learners are weak, that is, $\epsilon_j < 1/2, \forall j$; if not, we stop adding new base-learners. Note that this error rate is not on the original problem but on the dataset used at step j. We define $\beta_j = \epsilon_j/(1-\epsilon_j) < 1$, and we set $p^t_{j+1} = \beta_j p^t_j$ if d_j correctly classifies x^t; otherwise, $p^t_{j+1} = p^t_j$. Because p^t_{j+1} should be probabilities, there is a normalization where we divide p^t_{j+1} by $\sum_t p^t_{j+1}$, so that they sum up to 1. This has the effect that the probability of a correctly classified instance is decreased, and the probability of a misclassified instance increases. Then a new sample of the same size is drawn from the original sample according to these modified probabilities, p^t_{j+1}, with replacement, and is used to train d_{j+1}.

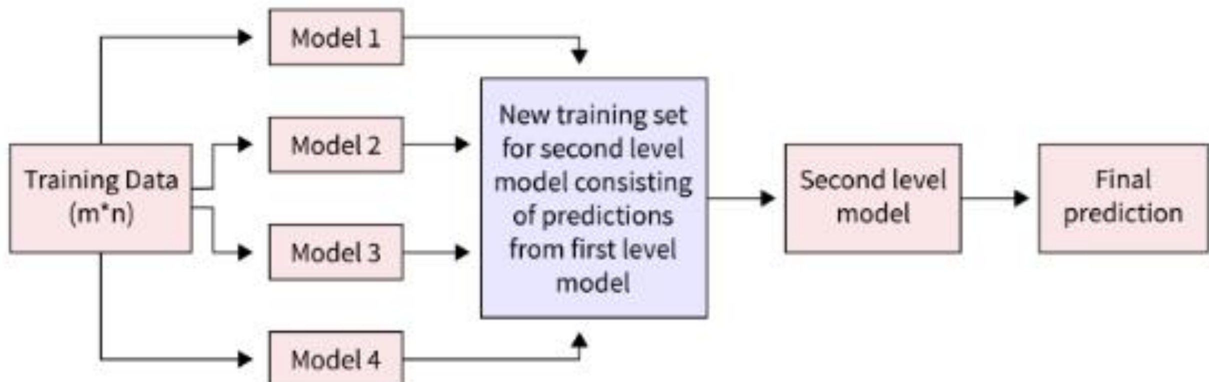
This has the effect that d_{j+1} focuses more on instances misclassified by d_j; that is why the base-learners are chosen to be simple and not accurate, since otherwise the next training sample would contain only a few outlier and noisy instances repeated many times over. For example, with decision trees, *decision stumps*, which are trees grown only one or two levels, are used. So it is clear that these would have bias but the decrease in variance is larger and the overall error decreases. An algorithm like the linear discriminant has low variance, and we cannot gain by AdaBoosting linear discriminants.

Once training is done, AdaBoost is a voting method. Given an instance, all d_j decide and a weighted vote is taken where weights are proportional to the base-learners' accuracies (on the training set): $w_j = \log(1/\beta_j)$. Freund and Schapire (1996) showed improved accuracy in twenty-two benchmark problems, equal accuracy in one problem, and worse accuracy in four problems.

- The success of AdaBoost is due to its property of increasing the margin.
- If the margin increases, the training instances are better separated and an error is less likely.

7. Stacking:

- The steps of Stacking are as follows:
- We use initial training data to train m-number of algorithms.
- Using the output of each algorithm, we create a new training set.
- Using the new training set, we create a meta-model algorithm.
- Using the results of the meta-model, we make the final prediction. The results are combined using weighted averaging.



- Stacking mainly differ from bagging and boosting on two points. First stacking often considers heterogeneous weak learners whereas bagging and boosting consider mainly homogeneous weak learners.
- Second, stacking learns to combine the base models using a meta-model whereas bagging and boosting combine weak learners following deterministic algorithms.

How stacking works?

- We split the training data into K-folds just like K-fold cross-validation.
- A base model is fitted on the K-1 parts and predictions are made for Kth part.
- We do for each part of the training data.
- The base model is then fitted on the whole train data set to calculate its performance on the test set.
- We repeat the last 3 steps for other base models.
- Predictions from the train set are used as features for the second level model.
- Second level model is used to make a prediction on the test set.
- **Data** - The Dataset to be used is split into training and testing data into n folds. This is achieved by repeated **n-fold** cross-validation, a technique to guarantee the efficient performance of the model.
- **Fitting data to base model** - Based on the above n-fold data, the first fold is assigned to the base model, and the output is generated. This is done for all n folds for all the base models.
- **Level 1 model** - Now that we have the results for the base models, we train the level 1 model.
- **Final prediction** - Predictions based on the level 1 model are used as the features for the model, and then they can be tested on the test data for the final results of the stack model

8. Unsupervised Learning:

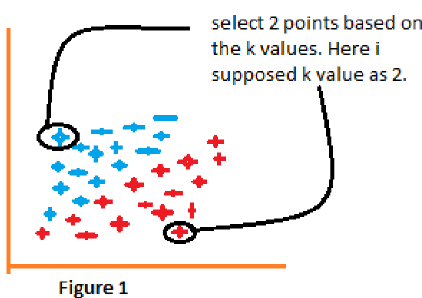
- Unsupervised Learning Algorithms allow users to perform more complex processing tasks compared to supervised learning.
- Although, unsupervised learning can be more unpredictable compared with other natural learning methods.
- Unsupervised learning algorithms include clustering, anomaly detection, neural networks, etc.

Supervised Learning	Unsupervised Learning
It uses known and labeled data as input	It uses unlabeled data as input

It has a feedback mechanism	It has no feedback mechanism
The most commonly used supervised learning algorithms are: • Decision tree • Logistic regression • Support vector machine	The most commonly used unsupervised learning algorithms are: • K-means clustering • Hierarchical clustering • Apriori algorithm

9. K-means Clustering:

K-means is a data clustering approach for unsupervised machine learning that can separate unlabeled data into a predetermined number of disjoint groups of equal variance – clusters – based on their similarities.



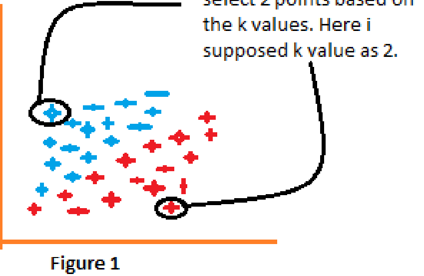
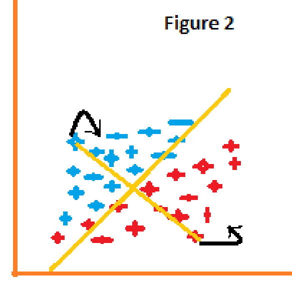


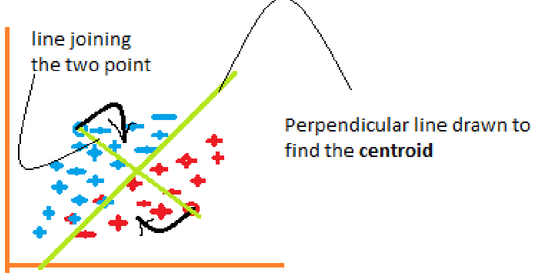
Figure 1



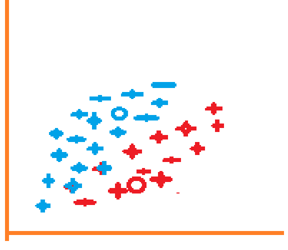
= original points

= the original points moved to centroid.

F2: Find the average of all the blue points and red points and move the selected points to **centroid.**



F3: Some of the red points changed to blue points, that means they belong to the group blue now. Again the repeat the same process.



F4: The same process has been applied here. This process will be continued until we get the **two complete different cluster.**

- Step 1: First, we need to provide the number of clusters, K, that need to be generated by this algorithm.
- Step 2: Next, choose K data points at random and assign each to a cluster. Briefly, categorize the data based on the number of data points.
- Step 3: The cluster centroids will now be computed.
- Step 4: Iterate the steps below until we find the ideal centroid, which is the assigning of data points to clusters that do not vary.
- 4.1 The sum of squared distances between data points and centroids would be calculated first.
- 4.2 At this point, we need to allocate each data point to the cluster that is closest to the others (centroid).
- 4.3 Finally, compute the centroids for the clusters by averaging all of the cluster's data points.

- K-means implements the Expectation-Maximization strategy to solve the problem.

- Let us say we have an image that is stored with 24 bits/pixel and can have up to 16 million colors.
- Assume we have a color screen with 8 bits/pixel that can display only 256 colors.
- We want to find the best 256 colors among all 16 million colors such that the image using only the 256 colors color quantization in the palette looks as close as possible to the original image. This is color quantization where we map from high to lower resolution.
- The aim is to map from a continuous space to a discrete space; this process is called vector quantization.
- If we quantize uniformly, it wastes the colormap by assigning entries to colors not existing in the image.
- Eg: If the image is a seascape, we expect to see many shades of blue and maybe no red.
- The colormap entries should reflect the original density.

Reference Vectors:

Let us say we have a sample of $\mathcal{X} = \{\boldsymbol{x}^t\}_{t=1}^N$. We have k *reference vectors*, $\boldsymbol{m}_j, j = 1, \ldots, k$. In our example of color quantization, $\boldsymbol{x}^t$ are the image pixel values in 24 bits and $\boldsymbol{m}_j$ are the color map entries also in 24 bits, with $k = 256$.

m_j – values
x^t – pixels
m_i - colour map

$$\|\boldsymbol{x}^t - \boldsymbol{m}_i\| = \min_j \|\boldsymbol{x}^t - \boldsymbol{m}_j\|$$

Codebook vectors:

- Instead of the original data value, we use the closest value we have in the alphabet of reference vectors. m_i are also called codebook vectors or code words, because this is a process of encoding/decoding.

- Going from x_t to i is a process of encoding the data using the codebook of m_i, i = 1, . . . , k and, on the receiving end, generating m_i from i is decoding. Quantization also allows compression.
- Eg: 24 bits to 8 bits

Let us see how we can calculate $\boldsymbol{m}_i$: When $\boldsymbol{x}^t$ is represented by $\boldsymbol{m}_i$, there is an error that is proportional to the distance, $\|\boldsymbol{x}^t - \boldsymbol{m}_i\|$. For the new image to look like the original image, we should have these distances as small as possible for all pixels. The total *reconstruction error* is defined as

RECONSTRUCTION ERROR

$$E(\{\boldsymbol{m}_i\}_{i=1}^k|\mathcal{X}) = \sum_t \sum_i b_i^t \|\boldsymbol{x}^t - \boldsymbol{m}_i\|^2 \quad (7.3)$$

where

$$b_i^t = \begin{cases} 1 & \text{if } \|\boldsymbol{x}^t - \boldsymbol{m}_i\| = \min_j \|\boldsymbol{x}^t - \boldsymbol{m}_j\| \\ 0 & \text{otherwise} \end{cases} \quad (7.4)$$

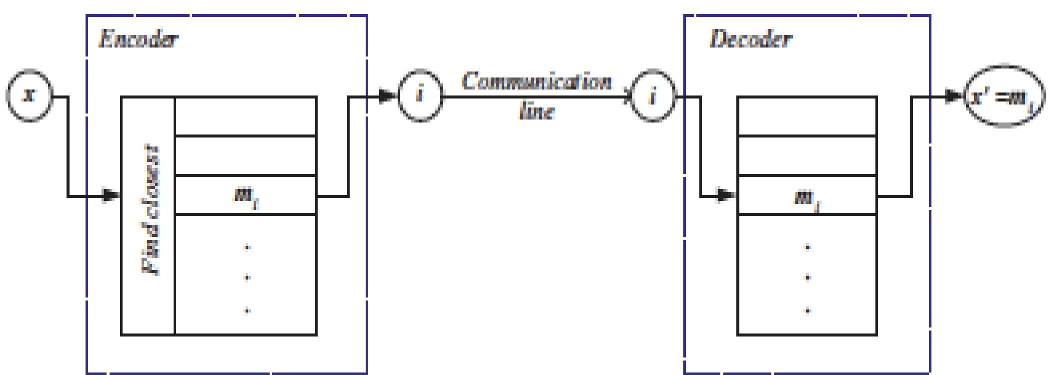


Figure 7.1 Given $\boldsymbol{x}$, the encoder sends the index of the closest code word and the decoder generates the code word with the received index as $\boldsymbol{x}'$. Error is $\|\boldsymbol{x}' - \boldsymbol{x}\|^2$.

The best reference vectors are those that minimize the total reconstruction error. b_i^t also depend on $\boldsymbol{m}_i$, and we cannot solve this optimization problem analytically. We have an iterative procedure named *k-means clustering* for this: First, we start with some $\boldsymbol{m}_i$ initialized randomly. Then at each iteration, we first use equation 7.4 and calculate b_i^t for all $\boldsymbol{x}^t$, which are the *estimated labels*; if b_i^t is 1, we say that $\boldsymbol{x}^t$ belongs to the group of $\boldsymbol{m}_i$. Then, once we have these labels, we minimize equation 7.3. Taking its derivative with respect to $\boldsymbol{m}_i$ and setting it to 0, we get

k-MEANS CLUSTERING

$$\boldsymbol{m}_i = \frac{\sum_t b_i^t \boldsymbol{x}^t}{\sum_t b_i^t} \quad (7.5)$$

The reference vector is set to the mean of all the instances that it represents. Note that this is the same as the formula for the mean in equation 7.2, except that we place the estimated labels b_i^t in place of the labels r_i^t. This is an iterative procedure because once we calculate the new $\boldsymbol{m}_i$, b_i^t change and need to be recalculated, which in turn affect $\boldsymbol{m}_i$. These two steps are repeated until $\boldsymbol{m}_i$ stabilize (see figure 7.2). The pseudocode of the k-means algorithm is given in figure 7.3.

One disadvantage is that this is a local search procedure, and the final $\boldsymbol{m}_i$ highly depend on the initial $\boldsymbol{m}_i$. There are various methods for initialization:

- We can simply take randomly selected k instances as the initial $\boldsymbol{m}_i$.
- The mean of all data can be calculated and small random vectors may be added to the mean to get the k initial $\boldsymbol{m}_i$.

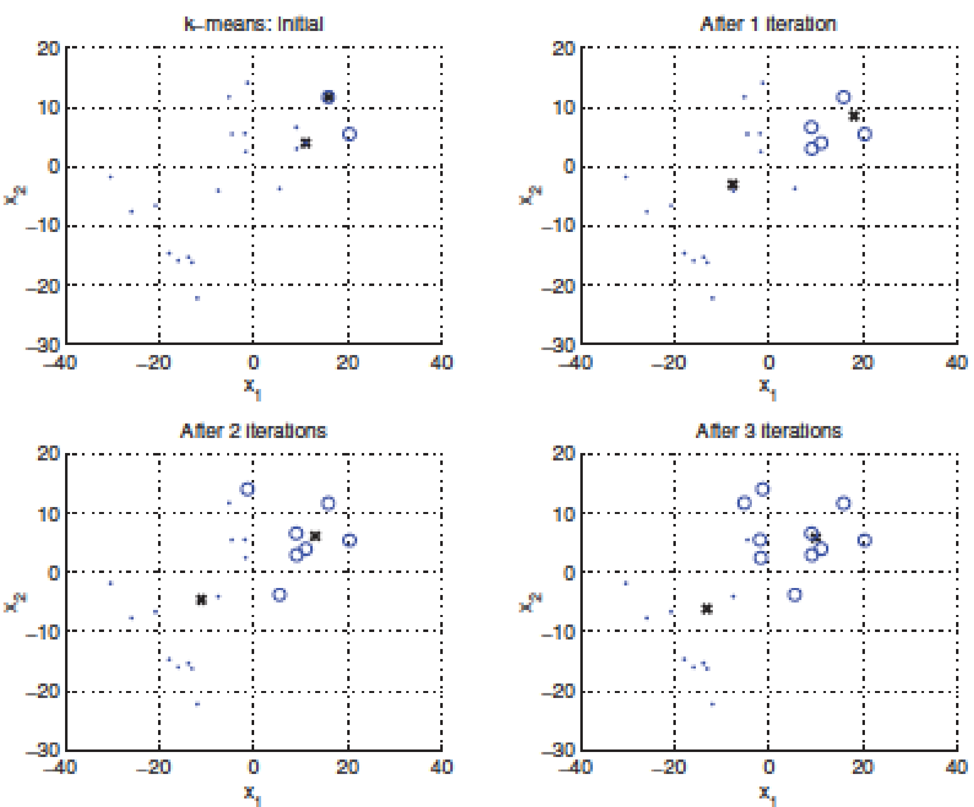


Figure 7.2 Evolution of k-means. Crosses indicate center positions. Data points are marked depending on the closest center.

- We can calculate the principal component, divide its range into k equal intervals, partitioning the data into k groups, and then take the means of these groups as the initial centers.
- There are also algorithms for adding new centers incrementally or deleting empty leader cluster ones.
- In leader cluster algorithm, an instance that is far away algorithm from existing centers causes the creation of a new center at that point

Initialize $\boldsymbol{m}_i, i = 1, \ldots, k$, for example, to k random $\boldsymbol{x}^t$
Repeat
  For all $\boldsymbol{x}^t \in \mathcal{X}$

$$b_i^t \leftarrow \begin{cases} 1 & \text{if } \|\boldsymbol{x}^t - \boldsymbol{m}_i\| = \min_j \|\boldsymbol{x}^t - \boldsymbol{m}_j\| \\ 0 & \text{otherwise} \end{cases}$$

  For all $\boldsymbol{m}_i, i = 1, \ldots, k$

$$\boldsymbol{m}_i \leftarrow \sum_t b_i^t \boldsymbol{x}^t / \sum_t b_i^t$$

Until $\boldsymbol{m}_i$ converge

Figure 7.3 k-means algorithm.

10. Instance Based Learning : KNN

- Instance based learning is a supervised classification learning algorithm that performs operation after comparing the current instances with the previously trained instances, which have been stored in memory.
- Its name is derived from the fact that it creates assumption from the training data instances.
- Components of Instance Based Learning Framework
- The framework requires three components: Similarity function, Classification function, and Concept Description Updater.

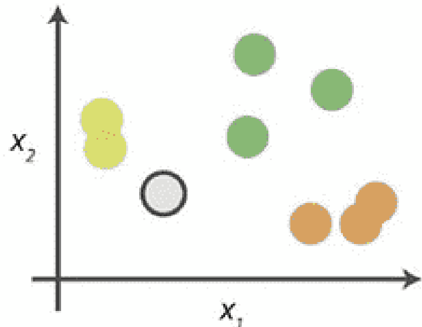
• Similarity Function: This computes similarity between a training instance the instances in the CD at a certain point in time. Similarities are numeric-valued.
• Classification Function: Given a new instance, it gives us the classification for that instance, based on the values coming from the similarity function, instances in the CD and performance of these instances in classification so far.
• Concept Description Updater: Maintains record of classification performance and decides which instances to include in the CD.
Some of the instance-based learning algorithms are :

- K Nearest Neighbor (KNN)
- Self-Organizing Map (SOM)
- Learning Vector Quantization (LVQ)
- Locally Weighted Learning (LWL)
- Case-Based Reasoning

The Nearest Neighbor Classifier

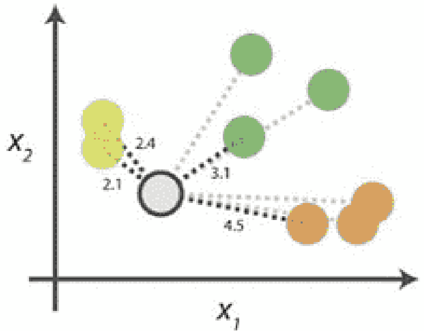
- The most commonly used instance-based classification method is the nearest neighbor method.
- In this method, the nearest k instances to the test instance are determined.
- Then, a simple model is constructed on this set of k nearest neighbors in order to determine the class label.
    - Binary class
    - Multi label class

0. Look at the data



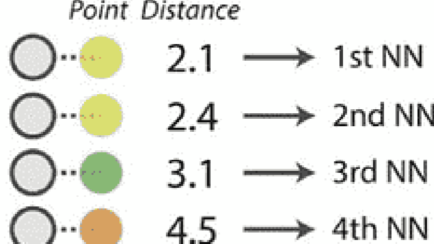
Say you want to classify the grey point into a class. Here, there are three potential classes - lime green, green and orange.

1. Calculate distances



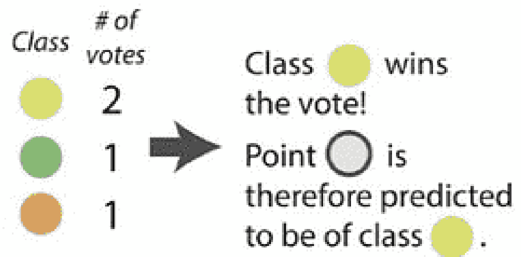
Start by calculating the distances between the grey point and all other points.

2. Find neighbours

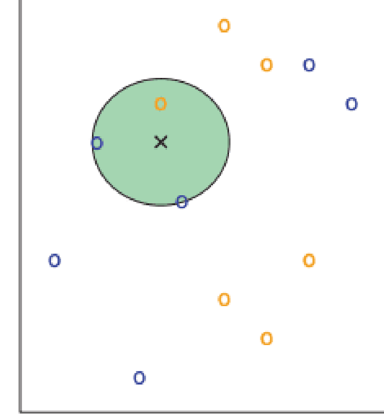


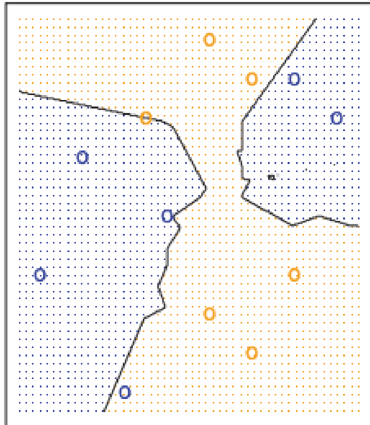
Next, find the nearest neighbours by ranking points by increasing distance. The nearest neighbours (NNs) of the grey point are the ones closest in dataspace.

3. Vote on labels



Vote on the predicted class labels based on the classes of the k nearest neighbours. Here, the labels were predicted based on the k=3 nearest neighbours.



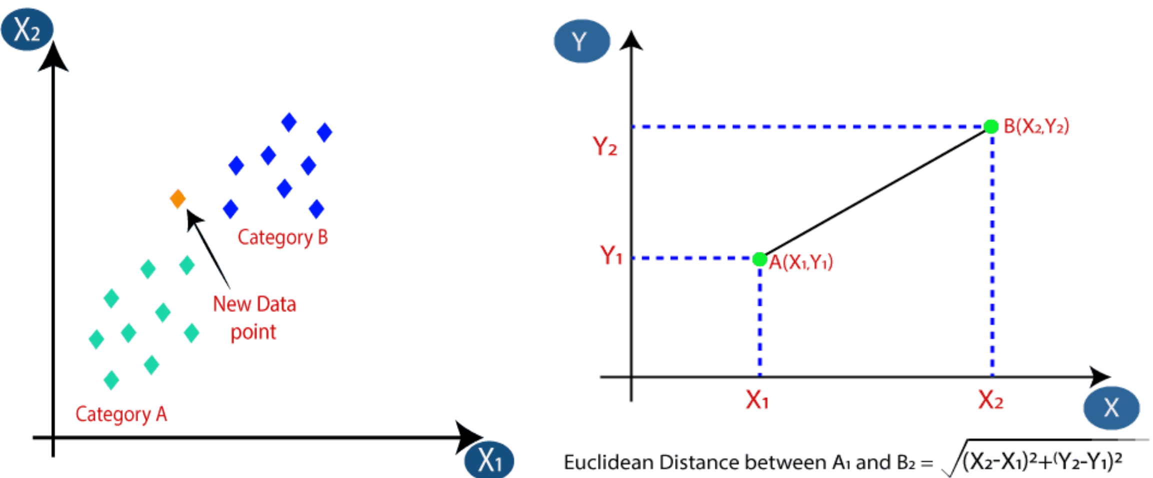


The K-NN working can be explained on the basis of the below algorithm:

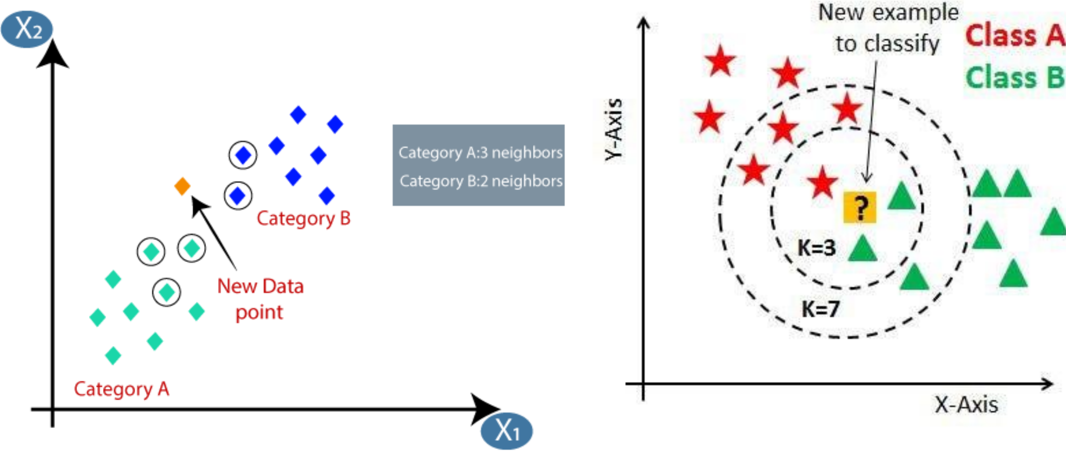
Step-1: Select the number K of the neighbors
Step-2: Calculate the Euclidean distance of K number of neighbors
Step-3: Take the K nearest neighbors as per the calculated Euclidean distance.

Step-4: Among these k neighbors, count the number of the data points in each category.
Step-5: Assign the new data points to that category for which the number of the neighbor is maximum.
Step-6: The model is ready.

- Firstly, we will choose the number of neighbors, so we will choose the k=5.
- Next, we will calculate the Euclidean distance between the data points.
- The Euclidean distance is the distance between two points, which we have already studied in geometry. It can be calculated as:



- By calculating the Euclidean distance we got the nearest neighbors, as three nearest neighbors in category A and two nearest neighbors in category B.
- Consider the below image:
- As we can see the 3 nearest neighbors are from category A, hence this new data point must belong to category A.



- Kvalue indicates the count of the nearest neighbors.
- We have to compute distances between test points and trained labels points.
- Updating distance metrics with every iteration is computationally expensive, and so KNN is a lazy learning algorithm.
- If we proceed with K=3, then we predict that test input belongs to class B, and if we continue with K=7, then we predict that test input belongs to class A.

Optimal K-Value:

- There are no pre-defined statistical methods to find the most favorable value of K.

- Initialize a random K value and start computing.
- Choosing a small value of K leads to unstable decision boundaries.
- The substantial K value is better for classification as it leads to smoothening the decision boundaries.
- Derive a plot between error rate and K denoting values in a defined range.
- Then choose the K value as having a minimum error rate.

- Calculating distance: The first step is to calculate the distance between the new point and each training point. There are various methods for calculating this distance, of which the most commonly known methods are — Euclidian, Manhattan (for continuous) and Hamming distance (for categorical).

- Euclidean Distance: Euclidean distance is calculated as the square root of the sum of the squared differences between a new point (x) and an existing point (y).

- Manhattan Distance: This is the distance between real vectors using the sum of their absolute difference.

- Hamming Distance: It is used for categorical variables. If the value (x) and the value (y) are the same, the distance D will be equal to 0 . Otherwise D=1.

11. Gaussian mixture models and Expectation maximization

A Generative View of Clustering

- Last time: hard and soft k-means algorithm
- Today: statistical formulation of clustering $\rightarrow$ principled, justification for updates
- We need a sensible measure of what it means to cluster the data well
  - This makes it possible to judge different methods
  - It may help us decide on the number of clusters
- An obvious approach is to imagine that the data was produced by a generative model
  - Then we adjust the model parameters to maximize the probability that it would produce exactly the data we observed

- We model the joint distribution as,

$$p(\mathbf{x}, z) = p(\mathbf{x}|z)p(z)$$

- But in unsupervised clustering we do not have the class labels z.
- What can we do instead?

$$p(\mathbf{x}) = \sum_z p(\mathbf{x}, z) = \sum_z p(\mathbf{x}|z)p(z)$$

- This is a mixture model

Gaussian Mixture Model (GMM)

Most common mixture model: Gaussian mixture model (GMM)

- A GMM represents a **distribution** as

$$p(\mathbf{x}) = \sum_{k=1}^{K} \pi_k \mathcal{N}(\mathbf{x}|\mu_k, \Sigma_k)$$

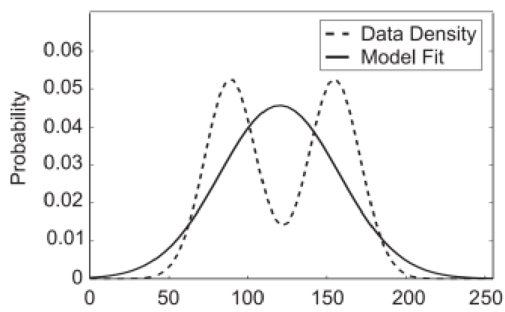
with π_k the mixing coefficients, where:

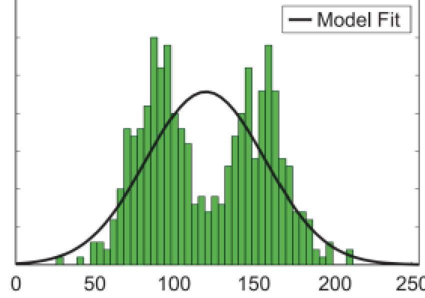
$$\sum_{k=1}^{K} \pi_k = 1 \quad \text{and} \quad \pi_k \geq 0 \quad \forall k$$

- GMM is a density estimator
- GMMs are **universal approximators of densities** (if you have enough Gaussians). Even diagonal GMMs are universal approximators.
- In general mixture models are very powerful, but harder to optimize

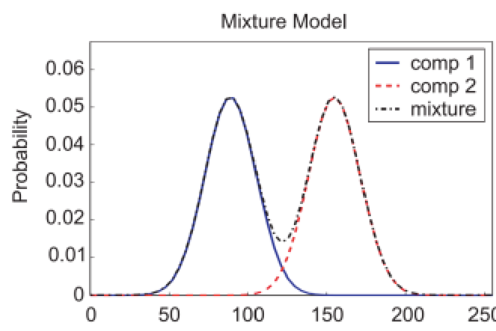
Visualizing a Mixture of Gaussians – 1D Gaussians

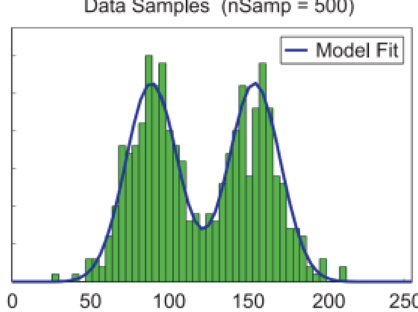
- If you fit a Gaussian to data:



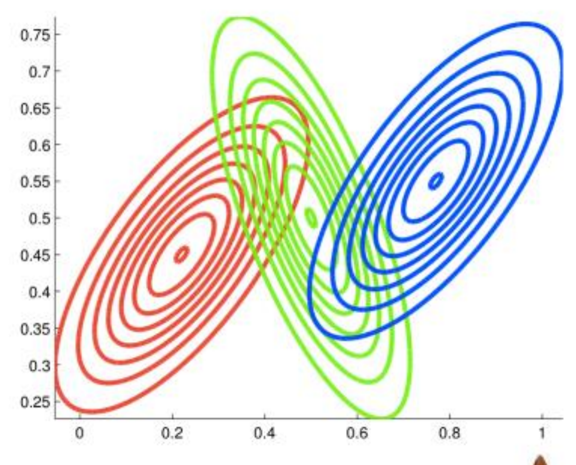


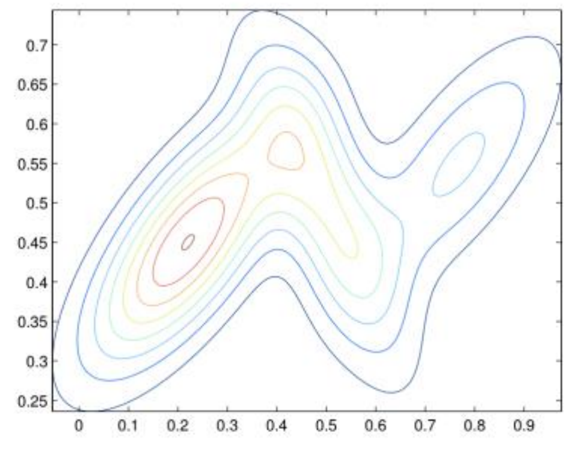
- Now, we are trying to fit a GMM (with $K = 2$ in this example):





Visualizing a Mixture of Gaussians – 2D Gaussians





Fitting GMMs: Maximum Likelihood

- Maximum likelihood maximizes

$$\ln p(\mathbf{X}|\pi, \mu, \Sigma) = \sum_{n=1}^{N} \ln\left(\sum_{k=1}^{K} \pi_k \mathcal{N}(\mathbf{x}^{(n)}|\mu_k, \Sigma_k)\right)$$

w.r.t $\Theta = \{\pi_k, \mu_k, \Sigma_k\}$

- Problems:
  - Singularities: Arbitrarily large likelihood when a Gaussian explains a single point
  - Identifiability: Solution is invariant to permutations
  - Non-convex

Latent Variable

- Our original representation had a hidden (latent) variable z which would represent which Gaussian generated our observation $\mathbf{x}$, with some probability
- Let $z \sim \text{Categorical}(\boldsymbol{\pi})$ (where $\pi_k \geq 0$, $\sum_k \pi_k = 1$)
- Then:

$$p(\mathbf{x}) = \sum_{k=1}^{K} p(\mathbf{x}, z = k)$$

$$= \sum_{k=1}^{K} \underbrace{p(z = k)}_{\pi_k} \underbrace{p(\mathbf{x}|z = k)}_{\mathcal{N}(\mathbf{x}|\mu_k, \Sigma_k)}$$

- This breaks a complicated distribution into simple components - the price is the hidden variable.

Latent Variable Models

- Some model variables may be unobserved, either at training or at test time, or both
- If occasionally unobserved they are missing, e.g., undefined inputs, missing class labels, erroneous targets
- Variables which are always unobserved are called latent variables, or sometimes hidden variables

- We may want to intentionally introduce latent variables to model complex dependencies between variables – this can actually simplify the model
- Form of divide-and-conquer: use simple parts to build complex models

- In a mixture model, the identity of the component that generated a given datapoint is a latent variable

Back to GMM

- A Gaussian mixture distribution:

$$p(\mathbf{x}) = \sum_{k=1}^{K} \pi_k \mathcal{N}(\mathbf{x}|\mu_k, \Sigma_k)$$

- We had: $z \sim \mathrm{Categorical}(\boldsymbol{\pi})$ (where $\pi_k \geq 0$, $\sum_k \pi_k = 1$)
- Joint distribution: $p(\mathbf{x}, \mathbf{z}) = p(\mathbf{z})p(\mathbf{x}|\mathbf{z})$
- Log-likelihood:

$$\begin{aligned} \ell(\boldsymbol{\pi}, \mu, \Sigma) = \ln p(\mathbf{X}|\pi, \mu, \Sigma) &= \sum_{n=1}^{N} \ln p(\mathbf{x}^{(n)}|\pi, \mu, \Sigma) \\ &= \sum_{n=1}^{N} \ln \sum_{z^{(n)}=1}^{K} p(\mathbf{x}^{(n)}|\, z^{(n)}; \mu, \Sigma) p(z^{(n)}|\, \boldsymbol{\pi}) \end{aligned}$$

- Note: We have a hidden variable $z^{(n)}$ for every observation
- General problem: sum inside the log
- How can we optimize this?

Maximum Likelihood

- **If we knew** $z^{(n)}$ for every $x^{(n)}$, the maximum likelihood problem is easy:

$$\ell(\boldsymbol{\pi}, \mu, \Sigma) = \sum_{n=1}^{N} \ln p(x^{(n)}, z^{(n)}|\pi, \mu, \Sigma) = \sum_{n=1}^{N} \ln p(\mathbf{x}^{(n)}|\, z^{(n)}; \mu, \Sigma) + \ln p(z^{(n)}|\, \boldsymbol{\pi})$$

- We have been optimizing something similar for Gaussian bayes classifiers
- We would get this (check old slides):

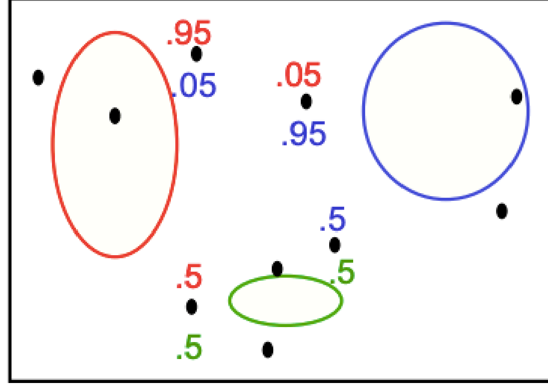
$$\mu_k = \frac{\sum_{n=1}^{N} 1_{[z^{(n)}=k]}\, \mathbf{x}^{(n)}}{\sum_{n=1}^{N} 1_{[z^{(n)}=k]}}$$

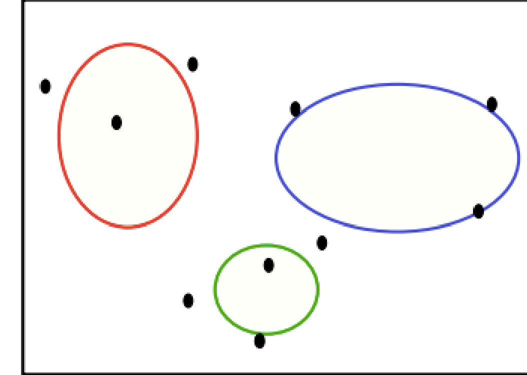
$$\Sigma_k = \frac{\sum_{n=1}^{N} 1_{[z^{(n)}=k]}\, (\mathbf{x}^{(n)} - \mu_k)(\mathbf{x}^{(n)} - \mu_k)^T}{\sum_{n=1}^{N} 1_{[z^{(n)}=k]}}$$

$$\pi_k = \frac{1}{N} \sum_{n=1}^{N} 1_{[z^{(n)}=k]}$$

Intuitively, How Can We Fit a Mixture of Gaussians?

- Optimization uses the Expectation Maximization algorithm, which alternates between two steps:
  1. E-step: Compute the posterior probability over z given our current model - i.e. how much do we think each Gaussian generates each datapoint.
  2. M-step: Assuming that the data really was generated this way, change the parameters of each Gaussian to maximize the probability that it would generate the data it is currently responsible for.





Relation to k-Means

- The K-Means Algorithm:
  1. Assignment step: Assign each data point to the closest cluster
  2. Refitting step: Move each cluster center to the center of gravity of the data assigned to it
- The EM Algorithm:
  1. E-step: Compute the posterior probability over z given our current model
  2. M-step: Maximize the probability that it would generate the data it is currently responsible for.

- Elegant and powerful method for finding maximum likelihood solutions for models with latent variables
  1. E-step:
     - In order to adjust the parameters, we must first solve the inference problem: Which Gaussian generated each datapoint?
     - We cannot be sure, so it's a distribution over all possibilities.

$$\gamma_k^{(n)} = p(z^{(n)} = k|\mathbf{x}^{(n)}; \pi, \mu, \Sigma)$$

  2. M-step:
     - Each Gaussian gets a certain amount of posterior probability for each datapoint.
     - We fit each Gaussian to the weighted datapoints
     - We can derive closed form updates for all parameters

- Remember that optimizing the likelihood is hard because of the sum inside of the log. Using Θ to denote all of our parameters:

$$\ell(\mathbf{X}, \Theta) = \sum_i \log(P(\mathbf{x}^{(i)}; \Theta)) = \sum_i \log\left(\sum_j P(\mathbf{x}^{(i)}, z^{(i)} = j; \Theta)\right)$$

- We can use a common trick in machine learning, introduce a new distribution, q:

$$\ell(\mathbf{X}, \Theta) = \sum_i \log\left(\sum_j q_j \frac{P(\mathbf{x}^{(i)}, z^{(i)} = j; \Theta)}{q_j}\right)$$

- Now we can swap them! Jensen's inequality - for concave function (like log)

$$f(\mathbb{E}[x]) = f\left(\sum_i p_i x_i\right) \geq \sum_i p_i f(x_i) = \mathbb{E}[f(x)]$$

- Applying Jensen's,

$$\sum_i \log\left(\sum_j q_j \frac{P(\mathbf{x}^{(i)}, z^{(i)} = j; \Theta)}{q_j}\right) \geq \sum_i \sum_j q_j \log\left(\frac{P(\mathbf{x}^{(i)}, z^{(i)} = j; \Theta)}{q_j}\right)$$

- Maximizing this lower bound will force our likelihood to increase.
- But how do we pick a q_i that gives a good bound?

EM derivation

- We got the sum outside but we have an inequality.

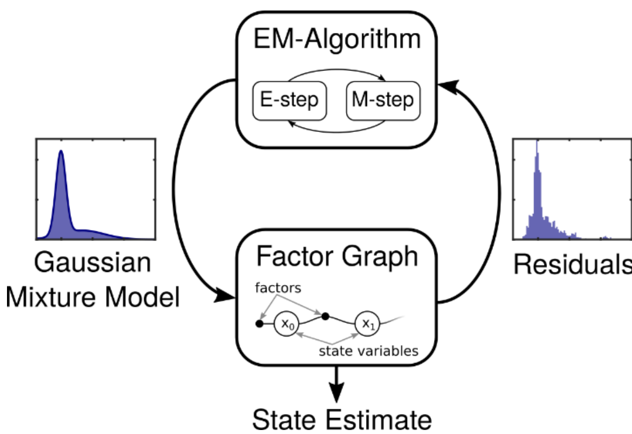
$$\ell(\mathbf{X}, \Theta) \geq \sum_i \sum_j q_j \log \left(\frac{P(\mathbf{x}^{(i)}, z^{(i)} = j; \Theta)}{q_j} \right)$$

- Lets fix the current parameters to Θ^{old} and try to find a good q_i
- What happens if we pick $q_j = p(z^{(i)} = j | x^{(i)}, \Theta^{old})$?
  - $\frac{P(\mathbf{x}^{(i)}, z^{(i)}; \Theta)}{p(z^{(i)} = j | x^{(i)}, \Theta^{old})} = P(\mathbf{x}^{(i)}; \Theta^{old})$ and the inequality becomes an equality!
- We can now define and optimize

$$Q(\Theta) = \sum_i \sum_j p(z^{(i)} = j | x^{(i)}, \Theta^{old}) \log \left(P(\mathbf{x}^{(i)}, z^{(i)} = j; \Theta) \right)$$

$$= \mathbb{E}_{P(z^{(i)} | \mathbf{x}^{(i)}, \Theta^{old})} [\log \left(P(\mathbf{x}^{(i)}, z^{(i)}; \Theta) \right)]$$

- We ignored the part that doesn't depend on Θ

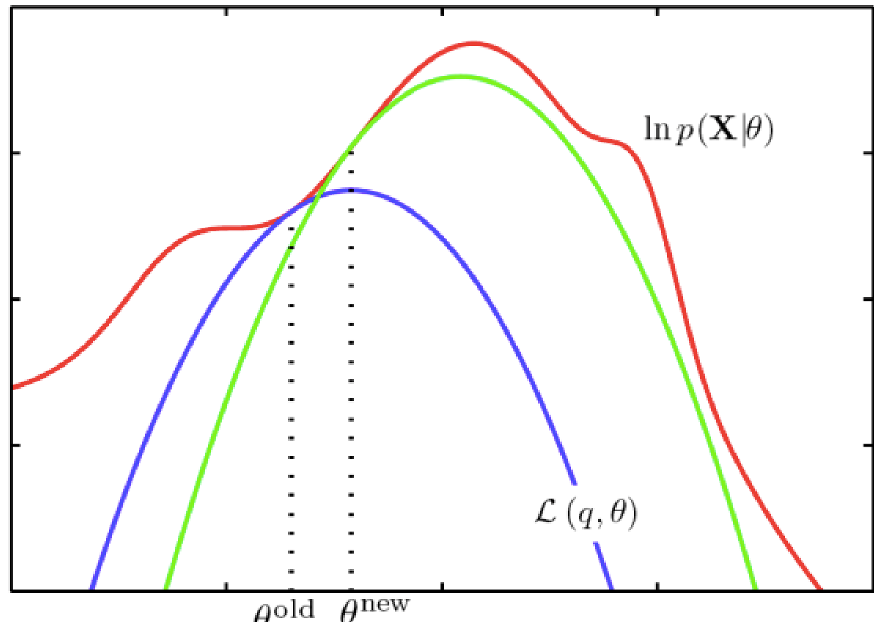


- Conceptually: We don't know $z^{(i)}$ so we average them given the current model.
- Practically: We define a function $Q(\Theta) = \mathbb{E}_{P(z^{(i)}|\mathbf{x}^{(i)},\Theta^{old})}[\log\left(P(\mathbf{x}^{(i)}, z^{(i)}; \Theta)\right)]$ that lower bounds the desired function and is equal at our current guess.
- If we now optimize Θ we will get a better lower bound!

$$\log(P(\mathbf{X}|\Theta^{old})) = Q(\Theta^{old}) \leq Q(\Theta^{new}) \leq P(P(\mathbf{X}|\Theta^{new}))$$

- We can iterate between expectation step and maximization step and the lower bound will always improve (or we are done)

Visualization of the EM Algorithm



- The EM algorithm involves alternately computing a lower bound on the log likelihood for the current parameter values and then maximizing this bound to obtain the new parameter values.

General EM Algorithm

1. Initialize Θ^{old}
2. E-step: Evaluate $p(\mathbf{Z}|\mathbf{X}, \Theta^{old})$ and compute

$$Q(\Theta, \Theta^{old}) = \sum_{z} p(\mathbf{Z}|\mathbf{X}, \Theta^{old}) \ln p(\mathbf{X}, \mathbf{Z}|\Theta)$$

3. M-step: Maximize

$$\Theta^{new} = arg \max_{\Theta} Q(\Theta, \Theta^{old})$$

4. Evaluate log likelihood and check for convergence (or the parameters). If not converged, $\Theta^{old} = \Theta^{new}$, Go to step 2

GMM E-Step: Responsibilities

Lets see how it works on GMM:

- Conditional probability (using Bayes rule) of $\mathbf{z}$ given $\mathbf{x}$

$$\begin{aligned}
\gamma_k = p(z = k|\mathbf{x}) &= \frac{p(z = k)p(\mathbf{x}|z = k)}{p(\mathbf{x})} \\
&= \frac{p(z = k)p(\mathbf{x}|z = k)}{\sum_{j=1}^{K} p(z = j)p(\mathbf{x}|z = j)} \\
&= \frac{\pi_k \mathcal{N}(\mathbf{x}|\mu_k, \Sigma_k)}{\sum_{j=1}^{K} \pi_j \mathcal{N}(\mathbf{x}|\mu_j, \Sigma_j)}
\end{aligned}$$

- γ_k can be viewed as the responsibility of cluster k towards $\mathbf{x}$

GMM E-Step

- Once we computed $\gamma_k^{(i)} = p(z^{(i)} = k|\mathbf{x}^{(i)})$ we can compute the expected likelihood

$$
\begin{aligned}
& \mathbb{E}_{P(z^{(i)}|\mathbf{x}^{(i)})}\left[\sum_i \log(P(\mathbf{x}^{(i)}, z^{(i)}|\Theta))\right] \\
= & \sum_i \sum_k \left(\gamma_k^{(i)} \log(P(z^i = k|\Theta)) + \log(P(\mathbf{x}^{(i)}|z^{(i)} = k, \Theta))\right) \\
= & \sum_i \sum_k \gamma_k^{(i)} \left(\log(\pi_k) + \log(\mathcal{N}(\mathbf{x}^{(i)}; \mu_k, \Sigma_k))\right) \\
= & \sum_k \sum_i \gamma_k^{(i)} \log(\pi_k) + \sum_k \sum_i \gamma_k^{(i)} \log(\mathcal{N}(\mathbf{x}^{(i)}; \mu_k, \Sigma_k))
\end{aligned}
$$

- We need to fit k Gaussians, just need to weight examples by γ_k

GMM M-Step

- Need to optimize

$$\sum_k \sum_i \gamma_k^{(i)} \log(\pi_k) + \sum_k \sum_i \gamma_k^{(i)} \log(\mathcal{N}(\mathbf{x}^{(i)}; \mu_k, \Sigma_k))$$

- Solving for μ_k and Σ_k is like fitting k separate Gaussians but with weights $\gamma_k^{(i)}$.
- Solution is similar to what we have already seen:

$$
\begin{aligned}
\mu_k &= \frac{1}{N_k} \sum_{n=1}^{N} \gamma_k^{(n)} \mathbf{x}^{(n)} \\
\Sigma_k &= \frac{1}{N_k} \sum_{n=1}^{N} \gamma_k^{(n)} (\mathbf{x}^{(n)} - \mu_k)(\mathbf{x}^{(n)} - \mu_k)^T \\
\pi_k &= \frac{N_k}{N} \quad \text{with} \quad N_k = \sum_{n=1}^{N} \gamma_k^{(n)}
\end{aligned}
$$

EM Algorithm for GMM

- Initialize the means μ_k, covariances Σ_k and mixing coefficients π_k
- Iterate until convergence:
  - E-step: Evaluate the responsibilities given current parameters

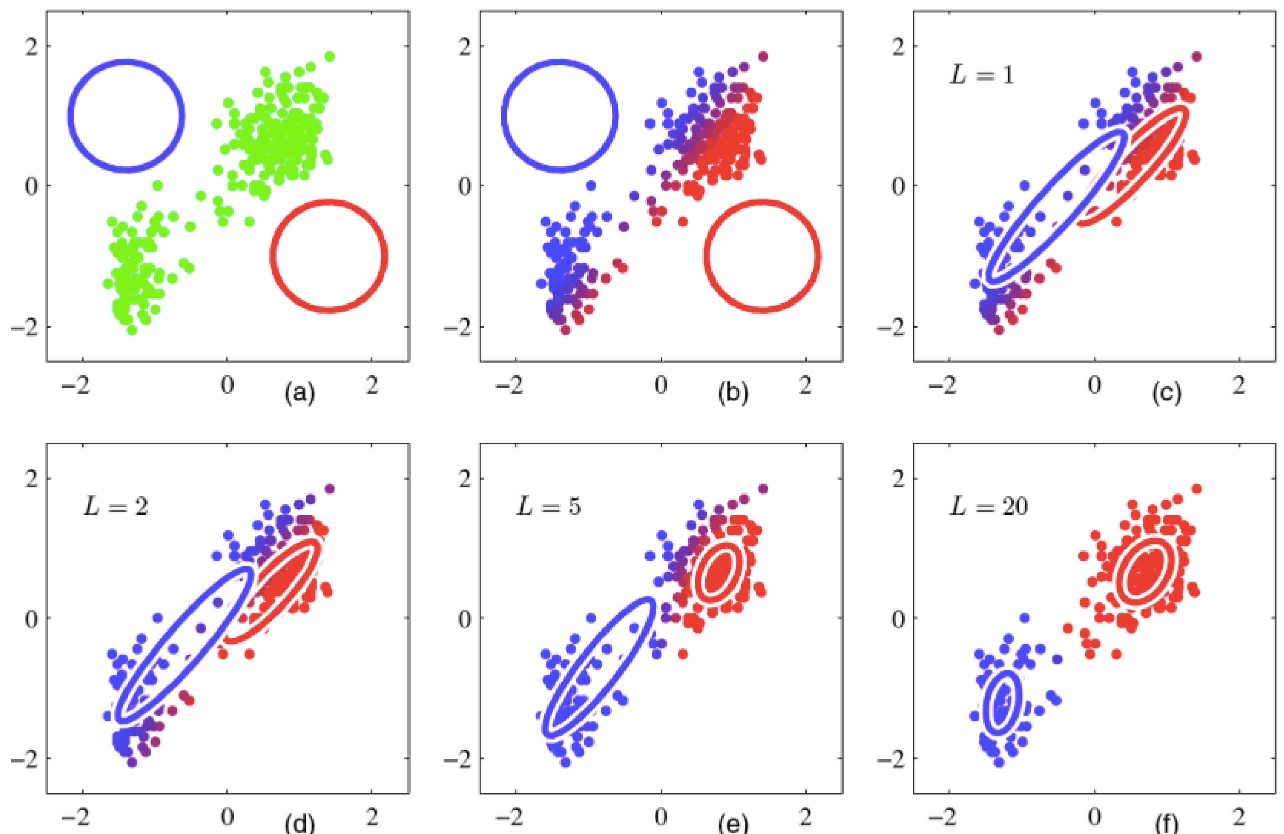
$$\gamma_k^{(n)} = p(z^{(n)}|\mathbf{x}) = \frac{\pi_k \mathcal{N}(\mathbf{x}^{(n)}|\mu_k, \Sigma_k)}{\sum_{j=1}^K \pi_j \mathcal{N}(\mathbf{x}^{(n)}|\mu_j, \Sigma_j)}$$

  - M-step: Re-estimate the parameters given current responsibilities

$$\begin{aligned}
\mu_k &= \frac{1}{N_k} \sum_{n=1}^N \gamma_k^{(n)} \mathbf{x}^{(n)} \\
\Sigma_k &= \frac{1}{N_k} \sum_{n=1}^N \gamma_k^{(n)} (\mathbf{x}^{(n)} - \mu_k)(\mathbf{x}^{(n)} - \mu_k)^T \\
\pi_k &= \frac{N_k}{N} \quad \text{with} \quad N_k = \sum_{n=1}^N \gamma_k^{(n)}
\end{aligned}$$

  - Evaluate log likelihood and check for convergence

$$\ln p(\mathbf{X}|\pi, \mu, \Sigma) = \sum_{n=1}^N \ln \left(\sum_{k=1}^K \pi_k \mathcal{N}(\mathbf{x}^{(n)}|\mu_k, \Sigma_k) \right)$$

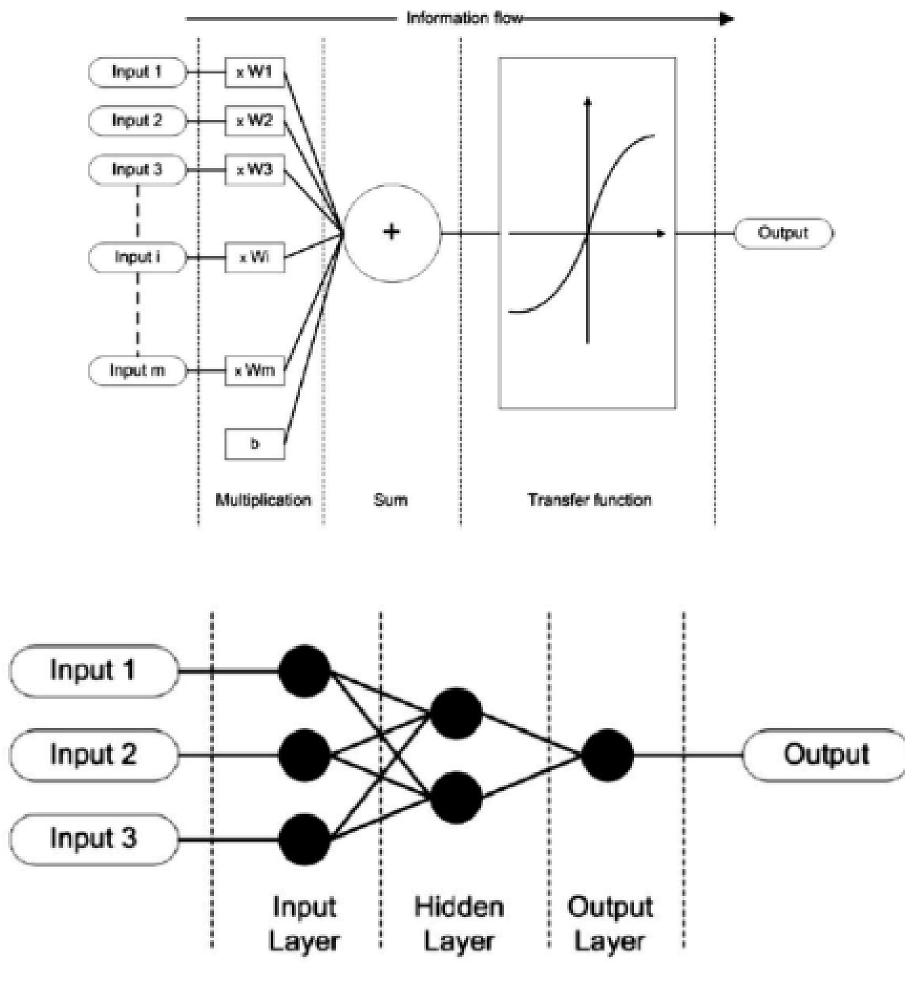


UNIT V **NEURAL NETWORKS**

Perceptron - Multilayer perceptron, activation functions, network training – gradient descent optimization – stochastic gradient descent, error backpropagation, from shallow networks to deep networks –Unit saturation (aka the vanishing gradient problem) – ReLU, hyperparameter tuning, batch normalization, regularization, dropout.

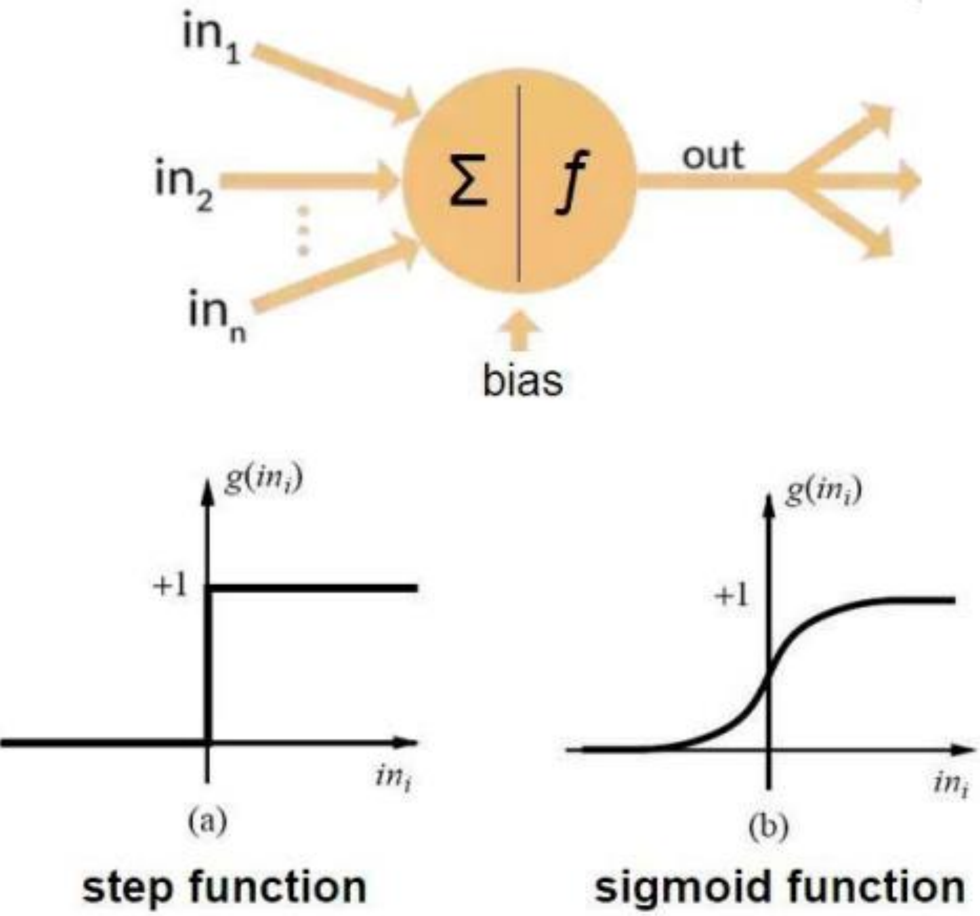
WHAT IS ARTIFICIAL NEURAL NETWORK?

An Artificial Neural Network (ANN) is a mathematical model that tries to simulate the structure and functionalities of biological neural networks. Basic building block of every artificial neural network is artificial neuron, that is, a simple mathematical model (function). Such a model has three simple sets of rules: multiplication, summation and activation. At the entrance of artificial neuron the inputs are weighted what means that every input value is multiplied with individual weight. In the middle section of artificial neuron is sum function that sums all weighted inputs and bias. At the exit of artificial neuron the sum of previously weighted inputs and bias is passing through activation function that is also called transfer function.



STRUCTURE AND FUNCTIONS OF ARTIFICIAL NEURON.

An artificial neuron is a mathematical function conceived as a model of biological neurons, a neural network. Artificial neurons are elementary units in an artificial neural network. The artificial neuron receives one or more inputs (representing excitatory postsynaptic potentials and inhibitory postsynaptic potentials at neural dendrites) and sums them to produce an output (or activation, representing a neuron's action potential which is transmitted along its axon). Usually each input is separately weighted, and the sum is passed through a non-linear function known as an activation function or transfer function. The transfer functions usually have a sigmoid shape, but they may also take the form of other non-linear functions, piecewise linear functions, or step functions. They are also often monotonically increasing continuous, differentiable and bounded. The thresholding function has inspired building logic gates referred to as threshold logic; applicable to building logic circuits resembling brain processing. For example, new devices such as memristors have been extensively used to develop such logic in recent times.



Perceptron

Whereas a computer generally has one processor, the brain is composed of a very large (10^{11}) number of processing units, namely, neurons, operating in parallel.
Neurons in the brain have connections, called synapses, to around 104 other neurons, all operating in parallel.
In a computer, the processor is active and the memory is separate and passive,
but it is believed that in the brain, both the processing and memory are distributed together over the network;
processing is done by the neurons, and the memory is in the synapses between the neurons.

The *perceptron* is the basic processing element. It has inputs that may come from the environment or may be the outputs of other perceptrons. Associated with each input, $x_j \in \Re, j = 1, \ldots, d$, is a *connection weight*, or *synaptic weight* $w_j \in \Re$, and the output, y, in the simplest case is a weighted sum of the inputs (see figure 11.1):

$$y = \sum_{j=1}^{d} w_j x_j + w_0$$

w_0 is the intercept value to make the model more general; it is generally modeled as the weight coming from an extra *bias unit*, x_0, which is always

+1. We can write the output of the perceptron as a dot product

$$y = \mathbf{w}^T \mathbf{x}$$

where $\mathbf{w} = [w_0, w_1, \ldots, w_d]^T$ and $\mathbf{x} = [1, x_1, \ldots, x_d]^T$ are *augmented* vectors to include also the bias weight and input.

During testing, with given weights, $\mathbf{w}$, for input $\mathbf{x}$, we compute the output y. To implement a given task, we need to *learn* the weights $\mathbf{w}$, the parameters of the system, such that correct outputs are generated given the inputs.

When $d = 1$ and x is fed from the environment through an input unit, we have

$$y = wx + w_0$$

The perceptron as defined in equation 11.1 defines a hyperplane and as such can be used to divide the input space into two: the half-space where it is positive and the half-space where it is negative (see chapter 10). By using it to implement a linear discriminant function, the perceptron can separate two classes by checking the sign of the output. If we define $s(\cdot)$ as the *threshold function*

$$s(a) = \begin{cases} 1 & \text{if } a > 0 \\ 0 & \text{otherwise} \end{cases}$$

then we can

$$\text{choose} \begin{cases} C_1 & \text{if } s(\mathbf{w}^T \mathbf{x}) > 0 \\ C_2 & \text{otherwise} \end{cases}$$

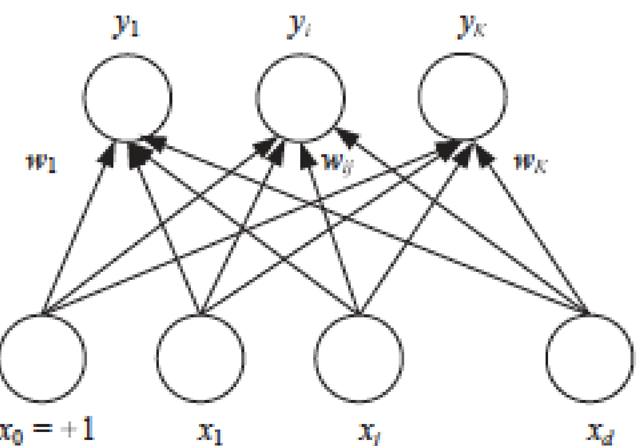
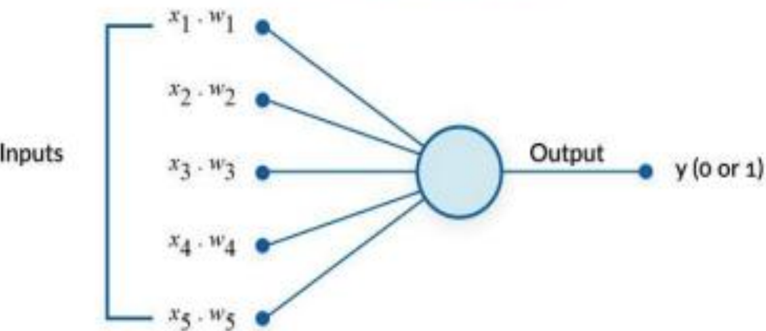


Figure 11.2 K parallel perceptrons. $x_j, j = 0, \ldots, d$ are the inputs and $y_i, i = 1, \ldots, K$ are the outputs. w_{ij} is the weight of the connection from input x_j to output y_i. Each output is a weighted sum of the inputs. When used for K-class classification problem, there is a postprocessing to choose the maximum, or softmax if we need the posterior probabilities.

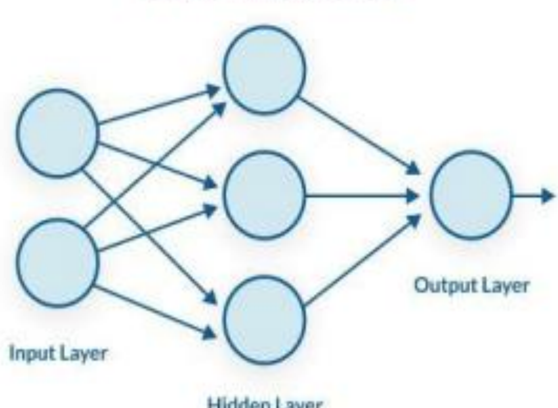
WHAT IS A PERCEPTRON?

A perceptron is a binary classification algorithm modeled after the functioning of the human brain—it was intended to emulate the neuron. The perceptron, while it has a simple structure, has the ability to learn a



What is Multilayer Perceptron?

A multilayer perceptron (MLP) is a group of perceptrons, organized in multiple layers, that can accurately answer complex questions. Each perceptron in the first layer (on the left) sends signals to all the perceptrons in the second layer, and so on. An MLP contains an input layer, at least one hidden layer, and an output layer.



The perceptron learns as follows:

1. Takes the inputs which are fed into the perceptrons in the input layer, multiplies them by their weights, and computes the sum.
2. Adds the number one, multiplied by a "bias weight". This is a technical step that makes it possible to move the output function of each perceptron (the activation function) up, down, left and right on the number graph.
3. Feeds the sum through the activation function—in a simple perceptron system, the activation function is a step function.
4. The result of the step function is the output.

A multilayer perceptron is quite similar to a modern neural network. By adding a few ingredients, the perceptron architecture becomes a full-fledged deep learning system:

- **Activation functions and other** hyperparameters**:** a full neural network uses a variety of activation functions which output real values, not boolean values like in the classic perceptron. It is more flexible in terms of other details of the learning process, such as the number of training iterations (iterations and epochs), weight initialization schemes, regularization, and so on. All these can be tuned as hyperparameters.
- **Backpropagation**: a full neural network uses the backpropagation algorithm, to perform iterative backward passes which try to find the optimal values of perceptron weights, to generate the most accurate prediction.
- **Advanced architectures**: full neural networks can have a variety of architectures that can help solve specific problems. A few examples are Recurrent Neural Networks (RNN), Convolutional Neural Networks (CNN), and Generative Adversarial Networks (GAN).

Training a Perceptron

The perceptron defines a hyperplane, and the neural network perceptron is just a way of implementing the hyperplane.
Given a data sample, the weight values can be calculated offline and then when they are plugged in, the perceptron can be used to calculate the output values.

ONLINE LEARNING

In *online learning*, we do not write the error function over the whole sample but on individual instances. Starting from random initial weights, at each iteration we adjust the parameters a little bit to minimize the error, without forgetting what we have previously learned. If this error function is differentiable, we can use gradient descent.

For example, in regression the error on the single instance pair with index t, (x^t, r^t), is

$$E^t(\boldsymbol{w}|\boldsymbol{x}^t, r^t) = \frac{1}{2}(r^t - y^t)^2 = \frac{1}{2}[r^t - (\boldsymbol{w}^T\boldsymbol{x}^t)]^2$$

and for $j = 0, \ldots, d$, the online update is

$$\Delta w_j^t = \eta(r^t - y^t)x_j^t \quad (11.7)$$

where η is the learning factor, which is gradually decreased in time for convergence. This is known as *stochastic gradient descent*.

STOCHASTIC GRADIENT DESCENT

Similarly, update rules can be derived for classification problems using logistic discrimination where updates are done after each pattern, instead of summing them and doing the update after a complete pass over the training set. With two classes, for the single instance $(\boldsymbol{x}^t, \boldsymbol{r}^t)$ where $r_i^t = 1$ if $\boldsymbol{x}^t \in C_1$ and $r_i^t = 0$ if $\boldsymbol{x}^t \in C_2$, the single output is

$$y^t = \text{sigmoid}(\boldsymbol{w}^T\boldsymbol{x}^t)$$

and the cross-entropy is

$$E^t(\boldsymbol{w}|\boldsymbol{x}^t, \boldsymbol{r}^t) = -r^t \log y^t - (1 - r^t)\log(1 - y^t)$$

Update = LearningFactor ° (DesiredOutput – ActualOutput) ° Input

```
For i = 1,...,K
    For j = 0,...,d
        w_ij ← rand(−0.01,0.01)
Repeat
    For all (x^t, r^t) ∈ X in random order
        For i = 1,...,K
            o_i ← 0
            For j = 0,...,d
                o_i ← o_i + w_ij x_j^t
        For i = 1,...,K
            y_i ← exp(o_i)/ Σ_k exp(o_k)
        For i = 1,...,K
            For j = 0,...,d
                w_ij ← w_ij + η(r_i^t − y_i)x_j^t
Until convergence
```

Figure 11.3 Perceptron training algorithm implementing stochastic online gradient descent for the case with $K > 2$ classes. This is the online version of the algorithm given in figure 10.8.

11.5 Multilayer Perceptrons

A perceptron that has a single layer of weights can only approximate linear functions of the input and cannot solve problems like the XOR, where the discrimininant to be estimated is nonlinear. Similarly, a perceptron cannot be used for nonlinear regression. This limitation does not apply to feedforward networks with intermediate or *hidden layers* between the input and the output layers. If used for classification, such *multilayer perceptrons* (MLP) can implement nonlinear discriminants and, if used for regression, can approximate nonlinear functions of the input.

HIDDEN LAYERS

MULTILAYER PERCEPTRONS

Input $\boldsymbol{x}$ is fed to the input layer (including the bias), the "activation" propagates in the forward direction, and the values of the hidden units z_h are calculated (see figure 11.6). Each hidden unit is a perceptron by itself and applies the nonlinear sigmoid function to its weighted sum:

$$z_h = \text{sigmoid}(\boldsymbol{w}_h^T \boldsymbol{x}) = \frac{1}{1 + \exp\left[-\left(\sum_{j=1}^d w_{hj} x_j + w_{h0}\right)\right]},\ h = 1,\ldots,H \tag{11.11}$$

The output y_i are perceptrons in the second layer taking the hidden units as their inputs

$$y_i = \boldsymbol{v}_i^T \boldsymbol{z} = \sum_{h=1}^{H} v_{ih} z_h + v_{i0} \tag{11.12}$$

where there is also a bias unit in the hidden layer, which we denote by z_0, and v_{i0} are the bias weights. The input layer of x_j is not counted since no computation is done there and when there is a hidden layer, this is a two-layer network.

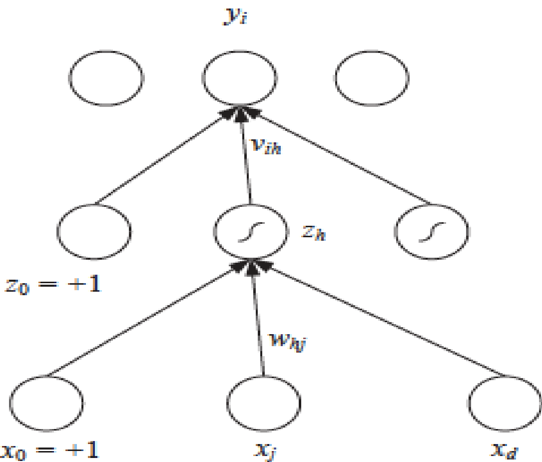
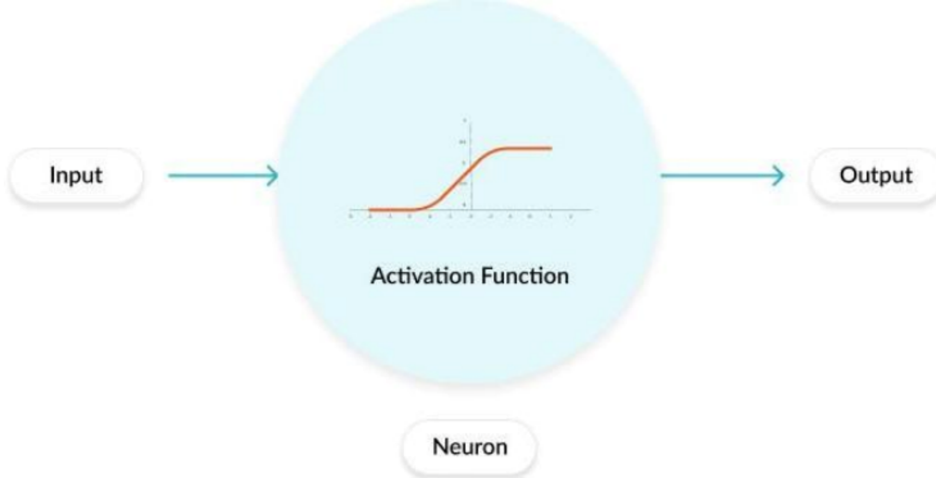


Figure 11.6 The structure of a multilayer perceptron. $x_j, j = 0, \ldots, d$ are the inputs and $z_h, h = 1, \ldots, H$ are the hidden units where H is the dimensionality of this hidden space. z_0 is the bias of the hidden layer. $y_i, i = 1, \ldots, K$ are the output units. w_{hj} are weights in the first layer, and v_{ih} are the weights in the second layer.

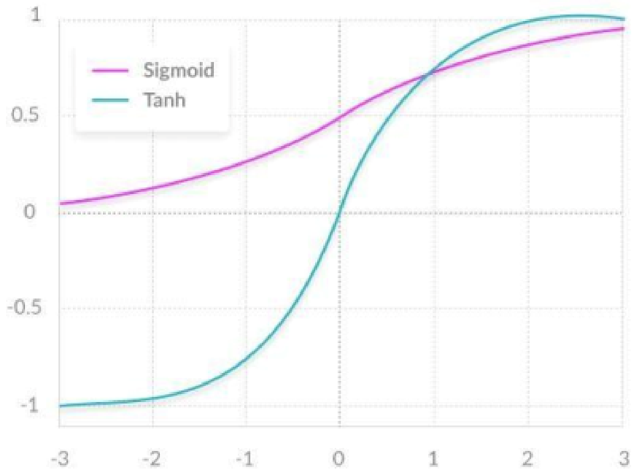
WHAT IS A NEURAL NETWORK ACTIVATION FUNCTION?

In a neural network, inputs, which are typically real values, are fed into the neurons in the network. Each neuron has a weight, and the inputs are multiplied by the weight and fed into the activation function. Each neuron's output is the input of the neurons in the next layer of the network, and so the inputs cascade through multiple activation functions until eventually, the output layer generates a prediction. Neural networks rely on nonlinear activation functions—the derivative of the activation function helps the network learn through the backpropagation process.



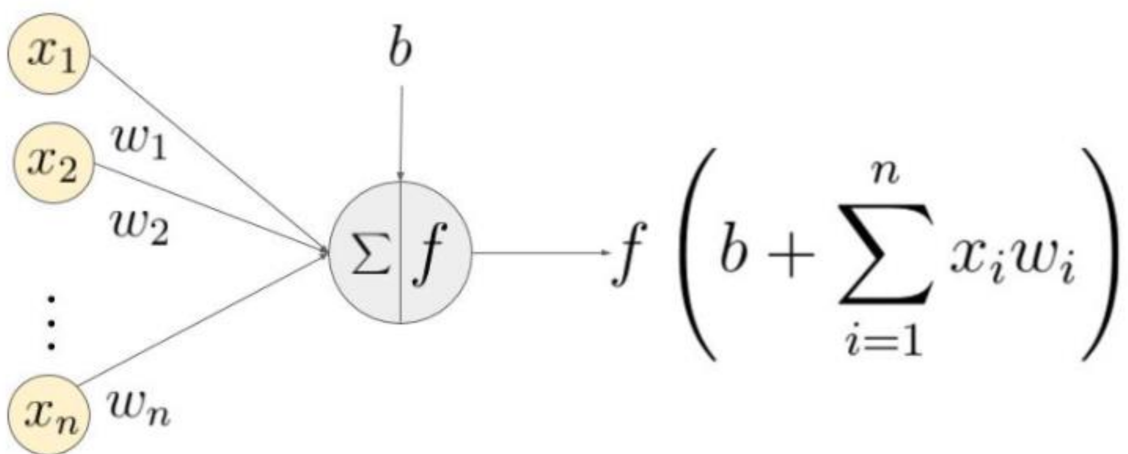
SOME COMMON ACTIVATION FUNCTIONS INCLUDE THE FOLLOWING:

1. **The sigmoid function** has a smooth gradient and outputs values between zero and one. For very high or low values of the input parameters, the network can be very slow to reach a prediction, called the *vanishing gradient* problem.
2. **The TanH function** is zero-centered making it easier to model inputs that are strongly negative strongly positive or neutral.
3. **The ReLu function** is highly computationally efficient but is not able to process inputs that approach zero or negative.
4. **The Leaky ReLu** function has a small positive slope in its negative area, enabling it to process zero or negative values.
5. **The Parametric ReLu** function allows the negative slope to be learned, performing backpropagation to learn the most effective slope for zero and negative input values.
6. **Softmax** is a special activation function use for output neurons. It normalizes outputs for each class between 0 and 1, and returns the probability that the input belongs to a specific class.
7. **Swish** is a new activation function discovered by Google researchers. It performs better than ReLu with a similar level of computational efficiency.



Two common neural network activation functions - Sigmoid and Tanh

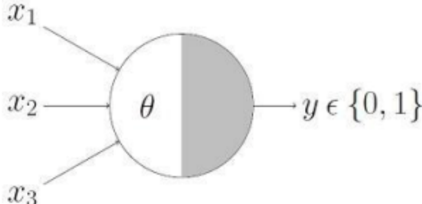
Name	Plot	Equation	Derivative
Identity		$f(x) = x$	$f'(x) = 1$
Binary step		$f(x) = \begin{cases} 0 & \text{for } x < 0 \\ 1 & \text{for } x \geq 0 \end{cases}$	$f'(x) = \begin{cases} 0 & \text{for } x \neq 0 \\ ? & \text{for } x = 0 \end{cases}$
Logistic (a.k.a Soft step)		$f(x) = \frac{1}{1 + e^{-x}}$	$f'(x) = f(x)(1 - f(x))$
TanH		$f(x) = \tanh(x) = \frac{2}{1 + e^{-2x}} - 1$	$f'(x) = 1 - f(x)^2$
ArcTan		$f(x) = \tan^{-1}(x)$	$f'(x) = \frac{1}{x^2 + 1}$
Rectified Linear Unit (ReLU)		$f(x) = \begin{cases} 0 & \text{for } x < 0 \\ x & \text{for } x \geq 0 \end{cases}$	$f'(x) = \begin{cases} 0 & \text{for } x < 0 \\ 1 & \text{for } x \geq 0 \end{cases}$
Parameteric Rectified Linear Unit (PReLU)[2]		$f(x) = \begin{cases} \alpha x & \text{for } x < 0 \\ x & \text{for } x \geq 0 \end{cases}$	$f'(x) = \begin{cases} \alpha & \text{for } x < 0 \\ 1 & \text{for } x \geq 0 \end{cases}$
Exponential Linear Unit (ELU)[3]		$f(x) = \begin{cases} \alpha(e^x - 1) & \text{for } x < 0 \\ x & \text{for } x \geq 0 \end{cases}$	$f'(x) = \begin{cases} f(x) + \alpha & \text{for } x < 0 \\ 1 & \text{for } x \geq 0 \end{cases}$
SoftPlus		$f(x) = \log_e(1 + e^x)$	$f'(x) = \frac{1}{1 + e^{-x}}$



An example of a neuron showing the input (x_1 - x_n), their corresponding weights (w_1 - w_n), a bias (b) and the activation function f applied to the weighted sum of the inputs.

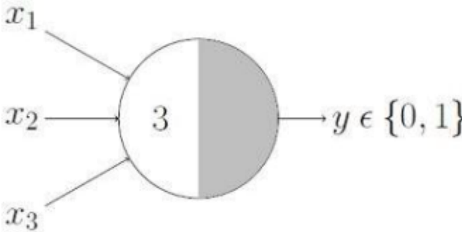
BOOLEAN FUNCTIONS USING McCULLOGH-PITTS NEURON

In any Boolean function, all inputs are Boolean and the output is also Boolean. So essentially, the neuron is just trying to learn a Boolean function.



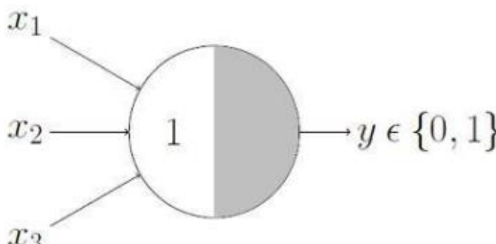
This representation just denotes that, for the boolean inputs x_1, x_2 and x_3 if the g(x) i.e., sum ≥ theta, the neuron will fire otherwise, it won't.

AND Function



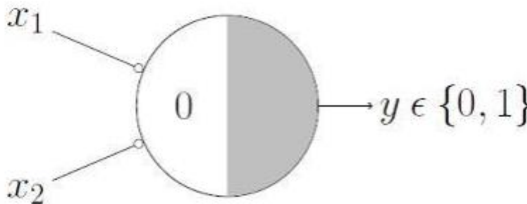
An AND function neuron would only fire when ALL the inputs are ON i.e., g(x) ≥ 3 here.

OR Function



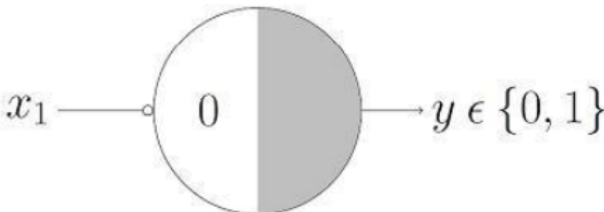
For an OR function neuron would fire if ANY of the inputs is ON i.e., g(x) ≥ 1 here.

NOR Function



For a NOR neuron to fire, we want ALL the inputs to be 0 so the thresholding parameter should also be 0 and we take them all as inhibitory input.

NOT Function



For a NOT neuron, 1 outputs 0 and 0 outputs 1. So we take the input as an inhibitory input and set the thresholding parameter to 0.

We can summarize these rules with the McCullough-Pitts output rule as:

The McCulloch-Pitts model of a neuron is simple yet has substantial computing potential. It also has a precise mathematical definition. However, this model is so simplistic that it only generates a binary output and also the weight and threshold values are fixed. The neural computing algorithm has diverse features for various applications. Thus, we need to obtain the neural model with more flexible computational features.

Gradient Descent Optimization:

The simplest approach to using gradient information is to choose the weight update in (5.27) to comprise a small step in the direction of the negative gradient, so that

$$\mathbf{w}^{(\tau+1)} = \mathbf{w}^{(\tau)} - \eta \nabla E(\mathbf{w}^{(\tau)}) \tag{5.41}$$

where the parameter $\eta > 0$ is known as the *learning rate*. After each such update, the gradient is re-evaluated for the new weight vector and the process repeated. Note that the error function is defined with respect to a training set, and so each step requires that the entire training set be processed in order to evaluate ∇E. Techniques that use the whole data set at once are called *batch* methods. At each step the weight vector is moved in the direction of the greatest rate of decrease of the error function, and so this approach is known as *gradient descent* or *steepest descent*. Although such an approach might intuitively seem reasonable, in fact it turns out to be a poor algorithm, for reasons discussed in Bishop and Nabney (2008).

For batch optimization, there are more efficient methods, such as *conjugate gradients* and *quasi-Newton* methods, which are much more robust and much faster than simple gradient descent (Gill *et al.*, 1981; Fletcher, 1987; Nocedal and Wright, 1999). Unlike gradient descent, these algorithms have the property that the error function always decreases at each iteration unless the weight vector has arrived at a local or global minimum.

In order to find a sufficiently good minimum, it may be necessary to run a gradient-based algorithm multiple times, each time using a different randomly chosen starting point, and comparing the resulting performance on an independent validation set.

There is, however, an on-line version of gradient descent that has proved useful in practice for training neural networks on large data sets (Le Cun *et al.*, 1989). Error functions based on maximum likelihood for a set of independent observations comprise a sum of terms, one for each data point

$$E(\mathbf{w}) = \sum_{n=1}^{N} E_n(\mathbf{w}). \tag{5.42}$$

On-line gradient descent, also known as *sequential gradient descent* or *stochastic gradient descent*, makes an update to the weight vector based on one data point at a time, so that

$$\mathbf{w}^{(\tau+1)} = \mathbf{w}^{(\tau)} - \eta \nabla E_n(\mathbf{w}^{(\tau)}). \tag{5.43}$$

This update is repeated by cycling through the data either in sequence or by selecting points at random with replacement. There are of course intermediate scenarios in which the updates are based on batches of data points.

One advantage of on-line methods compared to batch methods is that the former handle redundancy in the data much more efficiently. To see, this consider an extreme example in which we take a data set and double its size by duplicating every data point. Note that this simply multiplies the error function by a factor of 2 and so is equivalent to using the original error function. Batch methods will require double the computational effort to evaluate the batch error function gradient, whereas on-line methods will be unaffected. Another property of on-line gradient descent is the possibility of escaping from local minima, since a stationary point with respect to the error function for the whole data set will generally not be a stationary point for each data point individually.

Error Backpropagation:

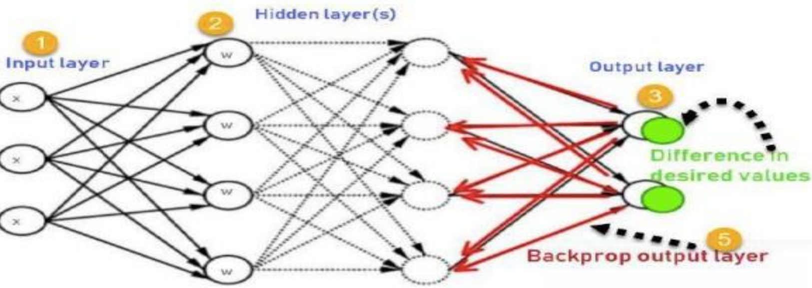
WHAT IS BACKPROPAGATION AND WHY IS IT IMPORTANT?

After a neural network is defined with initial weights, and a forward pass is performed to generate the initial prediction, there is an error function which defines how far away the model is from the true prediction. There are many possible algorithms that can minimize the error function—for example, one could do a brute force search to find the weights that generate the smallest error. However, for large neural networks, a training algorithm is needed that is very computationally efficient. Backpropagation is that algorithm—it can discover the optimal weights relatively quickly, even for a network with millions of weights.

HOW BACKPROPAGATION WORKS?

1. **Forward pass**—weights are initialized and inputs from the training set are fed into the network. The forward pass is carried out and the model generates its initial prediction.
2. **Error function**—the error function is computed by checking how far away the prediction is from the known true value.
3. **Backpropagation with gradient descent**—the backpropagation algorithm calculates how much the output values are affected by each of the weights in the model. To do this, it calculates partial derivatives, going back from the error function to a specific neuron and its weight. This provides complete traceability from total errors, back to a specific weight which contributed to that error. The result of backpropagation is a set of weights that minimize the error function.
4. **Weight update**—weights can be updated after every sample in the training set, but this is usually not practical. Typically, a batch of samples is run in one big forward pass, and then backpropagation performed on the aggregate result. The *batch size* and number of batches used in training, called *iterations*, are important hyperparameters that are tuned to get the best results. Running the entire training set through the backpropagation process is called an *epoch*.

an *epoch*.



Training algorithm of BPNN:

1. Inputs X, arrive through the pre connected path
2. Input is modeled using real weights W. The weights are usually randomly selected.
3. Calculate the output for every neuron from the input layer, to the hidden layers, to the output layer.
4. Calculate the error in the outputs

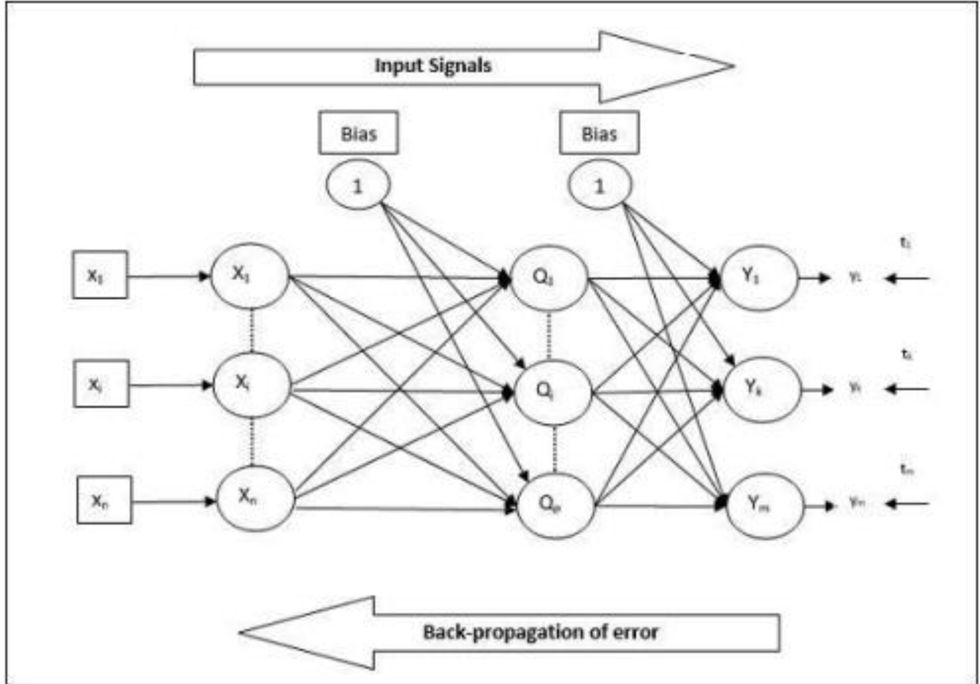
$Error_B$= Actual Output – Desired Output

5. Travel back from the output layer to the hidden layer to adjust the weights such that the error is decreased.

Keep repeating the process until the desired output is achieved

Architecture of back propagation network:

As shown in the diagram, the architecture of BPN has three interconnected layers having weights on them. The hidden layer as well as the output layer also has bias, whose weight is always 1, on them. As is clear from the diagram, the working of BPN is in two phases. One phase sends the signal from the input layer to the output layer, and the other phase back propagates the error from the output layer to the input layer.



Our goal in this section is to find an efficient technique for evaluating the gradient of an error function $E(\mathbf{w})$ for a feed-forward neural network. We shall see that this can be achieved using a local message passing scheme in which information is sent alternately forwards and backwards through the network and is known as *error backpropagation*, or sometimes simply as *backprop*.

The term backpropagation is also used to describe the training of a multilayer perceptron using gradient descent applied to a sum-of-squares error function.
It is important to recognize that the two stages are distinct. Thus, the first stage, namely the propagation of errors backwards through the network in order to evaluate derivatives, can be applied to many other kinds of network and not just the multilayer perceptron.

5.3.1 Evaluation of error-function derivatives

We now derive the backpropagation algorithm for a general network having arbitrary feed-forward topology, arbitrary differentiable nonlinear activation functions, and a broad class of error function. The resulting formulae will then be illustrated using a simple layered network structure having a single layer of sigmoidal hidden units together with a sum-of-squares error.

Many error functions of practical interest, for instance those defined by maximum likelihood for a set of i.i.d. data, comprise a sum of terms, one for each data point in the training set, so that

$$E(\mathbf{w}) = \sum_{n=1}^{N} E_n(\mathbf{w}). \tag{5.44}$$

Here we shall consider the problem of evaluating $\nabla E_n(\mathbf{w})$ for one such term in the error function. This may be used directly for sequential optimization, or the results can be accumulated over the training set in the case of batch methods.

Consider first a simple linear model in which the outputs y_k are linear combinations of the input variables x_i so that

$$y_k = \sum_i w_{ki} x_i \tag{5.45}$$

together with an error function that, for a particular input pattern n, takes the form

$$E_n = \frac{1}{2} \sum_k (y_{nk} - t_{nk})^2 \tag{5.46}$$

where $y_{nk} = y_k(\mathbf{x}_n, \mathbf{w})$. The gradient of this error function with respect to a weight w_{ji} is given by

$$\frac{\partial E_n}{\partial w_{ji}} = (y_{nj} - t_{nj}) x_{ni} \tag{5.47}$$

which can be interpreted as a 'local' computation involving the product of an 'error signal' $y_{nj} - t_{nj}$ associated with the output end of the link w_{ji} and the variable x_{ni} associated with the input end of the link. In Section 4.3.2, we saw how a similar formula arises with the logistic sigmoid activation function together with the cross entropy error function, and similarly for the softmax activation function together with its matching cross-entropy error function. We shall now see how this simple result extends to the more complex setting of multilayer feed-forward networks.

In a general feed-forward network, each unit computes a weighted sum of its inputs of the form

$$a_j = \sum_i w_{ji} z_i \tag{5.48}$$

where z_i is the activation of a unit, or input, that sends a connection to unit j, and w_{ji} is the weight associated with that connection. In Section 5.1, we saw that biases can be included in this sum by introducing an extra unit, or input, with activation fixed at $+1$. We therefore do not need to deal with biases explicitly. The sum in (5.48) is transformed by a nonlinear activation function $h(\cdot)$ to give the activation z_j of unit j in the form

$$z_j = h(a_j). \tag{5.49}$$

Note that one or more of the variables z_i in the sum in (5.48) could be an input, and similarly, the unit j in (5.49) could be an output.

For each pattern in the training set, we shall suppose that we have supplied the corresponding input vector to the network and calculated the activations of all of the hidden and output units in the network by successive application of (5.48) and (5.49). This process is often called *forward propagation* because it can be regarded as a forward flow of information through the network.

Now consider the evaluation of the derivative of E_n with respect to a weight w_{ji}. The outputs of the various units will depend on the particular input pattern n. However, in order to keep the notation uncluttered, we shall omit the subscript n from the network variables. First we note that E_n depends on the weight w_{ji} only via the summed input a_j to unit j. We can therefore apply the chain rule for partial derivatives to give

$$\frac{\partial E_n}{\partial w_{ji}} = \frac{\partial E_n}{\partial a_j}\frac{\partial a_j}{\partial w_{ji}}. \tag{5.50}$$

We now introduce a useful notation

$$\delta_j \equiv \frac{\partial E_n}{\partial a_j} \tag{5.51}$$

where the δ's are often referred to as *errors* for reasons we shall see shortly. Using (5.48), we can write

$$\frac{\partial a_j}{\partial w_{ji}} = z_i. \tag{5.52}$$

Substituting (5.51) and (5.52) into (5.50), we then obtain

$$\frac{\partial E_n}{\partial w_{ji}} = \delta_j z_i. \tag{5.53}$$

Equation (5.53) tells us that the required derivative is obtained simply by multiplying the value of δ for the unit at the output end of the weight by the value of z for the unit at the input end of the weight (where $z = 1$ in the case of a bias). Note that this takes the same form as for the simple linear model considered at the start of this section. Thus, in order to evaluate the derivatives, we need only to calculate the value of δ_j for each hidden and output unit in the network, and then apply (5.53).

As we have seen already, for the output units, we have

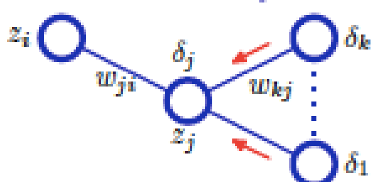
$$\delta_k = y_k - t_k \tag{5.54}$$

For batch methods, the derivative of the total error E can then be obtained by repeating the above steps for each pattern in the training set and then summing over all patterns:

$$\frac{\partial E}{\partial w_{ji}} = \sum_n \frac{\partial E_n}{\partial w_{ji}}. \tag{5.57}$$

In the above derivation we have implicitly assumed that each hidden or output unit in the network has the same activation function $h(\cdot)$. The derivation is easily generalized, however, to allow different units to have individual activation functions, simply by keeping track of which form of $h(\cdot)$ goes with which unit.

Figure 5.7 Illustration of the calculation of δ_j for hidden unit j by backpropagation of the δ's from those units k to which unit j sends connections. The blue arrow denotes the direction of information flow during forward propagation, and the red arrows indicate the backward propagation of error information.



provided we are using the canonical link as the output-unit activation function. To evaluate the δ's for hidden units, we again make use of the chain rule for partial derivatives,

$$\delta_j \equiv \frac{\partial E_n}{\partial a_j} = \sum_k \frac{\partial E_n}{\partial a_k}\frac{\partial a_k}{\partial a_j} \tag{5.55}$$

where the sum runs over all units k to which unit j sends connections. The arrangement of units and weights is illustrated in Figure 5.7. Note that the units labelled k could include other hidden units and/or output units. In writing down (5.55), we are making use of the fact that variations in a_j give rise to variations in the error function only through variations in the variables a_k. If we now substitute the definition of δ given by (5.51) into (5.55), and make use of (5.48) and (5.49), we obtain the following *backpropagation* formula

$$\delta_j = h'(a_j)\sum_k w_{kj}\delta_k \tag{5.56}$$

which tells us that the value of δ for a particular hidden unit can be obtained by propagating the δ's backwards from units higher up in the network, as illustrated in Figure 5.7. Note that the summation in (5.56) is taken over the first index on w_{kj} (corresponding to backward propagation of information through the network), whereas in the forward propagation equation (5.10) it is taken over the second index. Because we already know the values of the δ's for the output units, it follows that by recursively applying (5.56) we can evaluate the δ's for all of the hidden units in a feed-forward network, regardless of its topology.

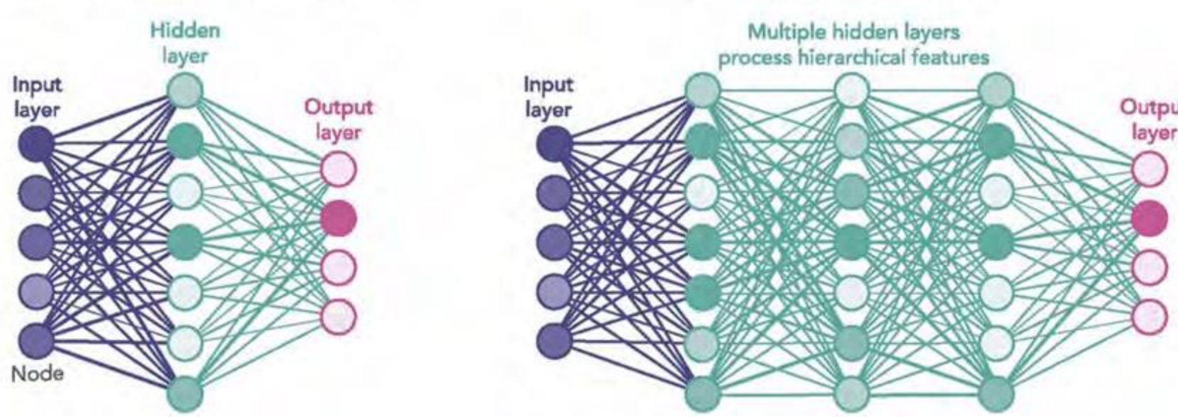
The backpropagation procedure can therefore be summarized as follows.

Error Backpropagation

1. Apply an input vector $\mathbf{x}_n$ to the network and forward propagate through the network using (5.48) and (5.49) to find the activations of all the hidden and output units.
2. Evaluate the δ_k for all the output units using (5.54).
3. Backpropagate the δ's using (5.56) to obtain δ_j for each hidden unit in the network.
4. Use (5.53) to evaluate the required derivatives.

From shallow networks to deep networks :

- A shallow neural network has only one hidden layer between the input and output layers, while a deep neural network has multiple hidden layers.
- A deep neural network learns to make decisions by processing information through multiple hidden layers.
- On the other hand, a shallow network is like having just one layer of decision-making, which might not be enough to capture the complexity of the problem at hand.
- A shallow network might be used for simple tasks like image classification, while a deep network might be used for more complex tasks like image segmentation or natural language processing.
- For instance, a shallow neural network with a single hidden layer can be used to recognize handwritten digits from the MNIST dataset.
- The main advantage of a shallow network is that it is computationally less expensive to train, and can be sufficient for simple tasks.
- However, it may not be powerful enough to capture complex patterns in the data.
- A deep network, on the other hand, can capture more complex patterns in the data and potentially achieve higher accuracy, but it is more computationally expensive to train and may require more data to avoid overfitting.
- Additionally, deep networks can be more challenging to design and optimize than shallow networks.



Unit saturation – Vanishing Gradient Problem:

- Neural network models are trained by the optimization algorithm of **gradient descent**.
- The input training data helps these models learn, and the loss function gauges how accurate the prediction performance is for each iteration when parameters get updated.
- As training goes, the goal is to reduce the loss function/prediction error by adjusting the parameters iteratively.

- Specifically, the gradient descent algorithm has a forward step and a backward step, which lets it do this.
- In forward propagation, input vectors/data move forward through the network using a formula to compute each neuron in the next layer. The formula consists of input/output, activation function *f*, weight *W* and bias *b*:
- $$O^l = f(WO^{l-1} + b)$$
- This computation iterates forward until it reaches an output or prediction.
- We then calculate the difference defined by a loss function, e.g., Mean Squared Error MSE, between the target variable y (in the output layer) and each prediction, y cap:
- $$Loss = \frac{1}{n}\sum_{i=1}^{n}(yi - \hat{yi})^2$$
- With this initial evaluation, we go through a backward pass (a.k.a. backpropagation) to adjust the weights and biases for each neuron in each layer. To update our neural nets, we first calculate the gradients, which is nothing but the derivatives of the loss function *w.r.t.* weights and biases. Then we nudge our algorithm to take a gradient descent step to minimize the loss function (where alpha is the learning rate):
- $$Wnew = Wold - \alpha(\frac{\partial Loss}{\partial Wold})$$
- Two opposite scenarios could happen in this case: the derivative term gets extremely small, i.e., approaches zero vs. this term gets extremely large and overflows.
- These issues are referred to as the Vanishing and Exploding Gradients, respectively.
- When you train your model for a while and the performance doesn't seem to get better, chances are your model is suffering from either **vanishing or exploding gradients**.

- **Vanishing or exploding gradients – intuition behind the problem Vanishing**
- During backpropagation, the calculation of (partial) *derivatives/gradients* in the weight update formula follows the Chain Rule, where gradients in earlier layers are the multiplication of gradients of later layers:
- $$\frac{\partial Loss}{\partial W^l} = \frac{\partial Loss}{\partial O^l}\frac{\partial O^l}{\partial z^l}\frac{\partial z^l}{\partial W^l}$$
- where
- $$z^l = W^l * O + b$$
- As the gradients frequently become SMALLER until they are close to zero, the new model weights (of the initial layers) will be virtually identical to the old weights without any updates.
- As a result, the gradient descent algorithm never converges to the optimal solution.
- This is known as the problem of vanishing gradients, and it's one example of unstable behaviors of neural nets.

- **Exploding**
- On the contrary, if the gradients get LARGER or even NaN as our backpropagation progresses, we would end up with exploding gradients having big weight updates, leading to the divergence of the gradient descent algorithm.

Vanishing

- Simply put, the vanishing gradients issue occurs when we use the Sigmoid or Tanh **activation functions** in the hidden layer; these functions squish a large input space into a small space.
- Take the Sigmoid as an example, we have the following p.d.f.:

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

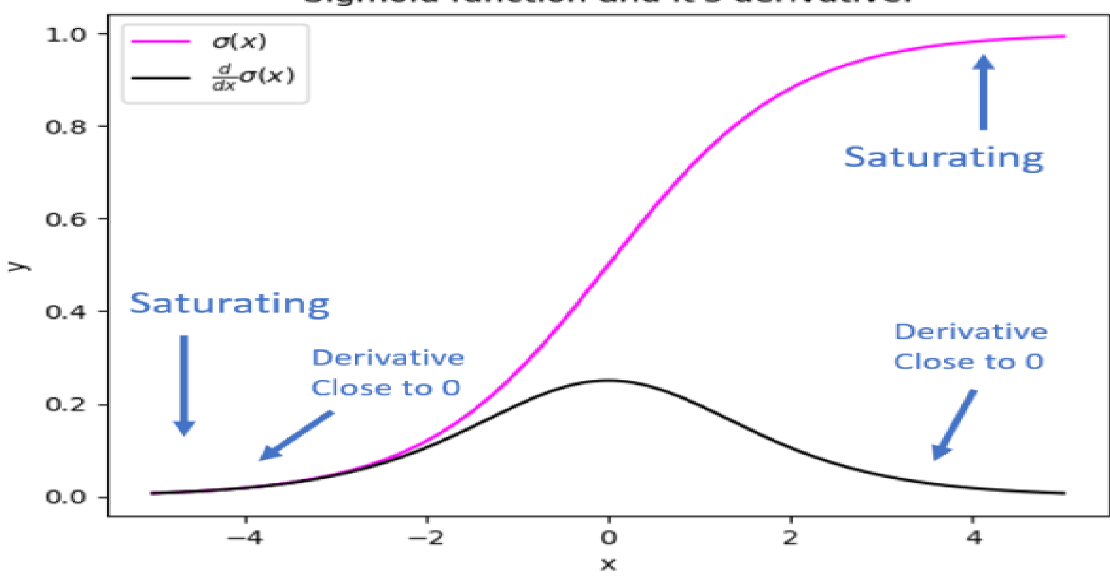
- Taking the derivative *w.r.t.* the parameter x, we get:

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

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

$$= \frac{1}{1+e^{-x}}\left(1 - \frac{1}{1+e^{-x}}\right) = y(1-y)$$

and if we visualize the Sigmoid function and its derivative:



Sigmoid function and its derivative

- We can see that the Sigmoid function squeezes our input space into a range between [0,1], and when the inputs become fairly small or fairly large, this function ***saturates*** at 0 or 1.
- These regions are referred to as 'saturating regions', whose derivatives become extremely close to zero.
- The same applies to the Tanh function that ***saturates*** at -1 and 1.
- Suppose that we have inputs that lie in any of the saturating regions, we would essentially have no gradient values to propagate back, leading to a zero update in earlier layer weights.
- Usually, this is no big of a concern for shallow networks with just a couple of layers, however, when we add more layers, vanishing gradients in initial layers will result in model training or convergence failure.
- This is due to the effect of multiplying *n* of these small numbers to compute gradients of the early layers in an *n*-layer network, meaning that the gradient decreases exponentially with *n* while the early layers train very slowly and thus the performance of the entire network degrades.

Exploding

- Moving on to the exploding gradients, in a nutshell, this problem is due to the **initial weights** assigned to the neural nets creating large losses.
- Big gradient values can accumulate to the point where large parameter updates are observed, causing gradient descents to oscillate without coming to global minima.

How to identify a vanishing or exploding gradients problem?

Acknowledging that the gradients' issues are something we need to avoid or fix when they do happen, how should we know that a model is suffering from vanishing or exploding gradients issues? Following are the few signs.

Vanishing

- Large changes are observed in parameters of later layers, whereas parameters of earlier layers change slightly or stay unchanged
- In some cases, weights of earlier layers can become 0 as the training goes

- The model learns slowly and often times, training stops after a few iterations
- Model performance is poor

Exploding

- Contrary to the vanishing scenario, exploding gradients shows itself as unstable, large parameter changes from batch/iteration to batch/iteration
- Model weights can become NaN very quickly
- Model loss also goes to NaN

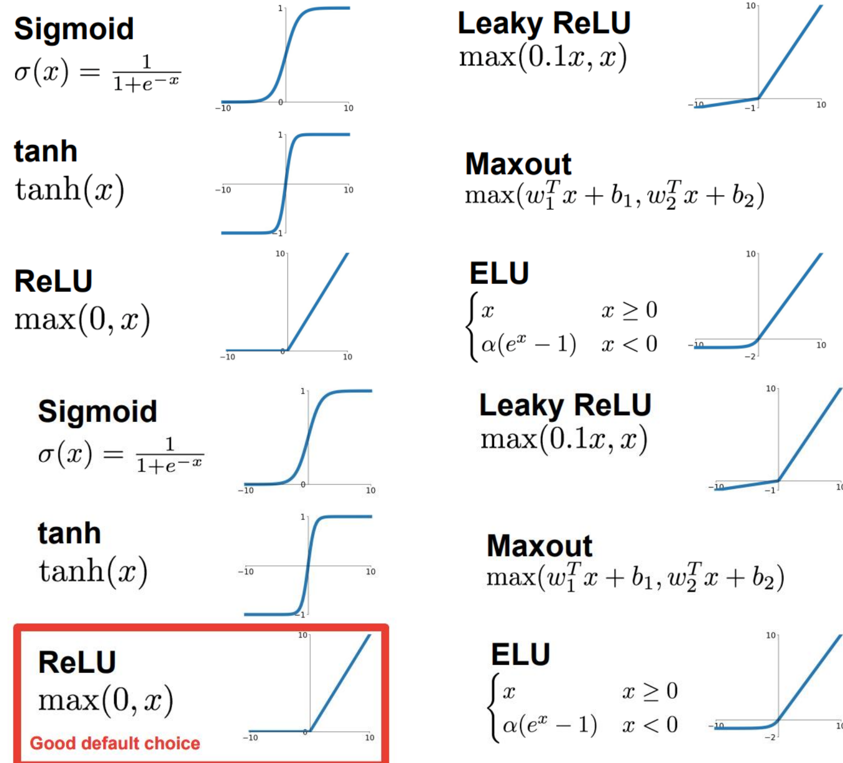
Practices to fix a vanishing or exploding gradients problem

With these indicators of the gradients problems in mind, let's explore the potential remedies to fix them.

- First, we will be focusing on the **vanishing scenario**: simulating a **binary classification** network model that suffers from this issue, and then demonstrating various solutions to fix it
- By the same token, we will be addressing the **exploding scenario** with a **regression** network model later

ReLU:

Activation Functions:



- An activation function is basically just a simple function that transforms its inputs into outputs that have a certain range.
- There are various types of activation functions that perform this task in a different manner.
- For example, the sigmoid activation function takes input and maps the resulting values in between 0 to 1.
- ReLU stands for rectified linear activation unit and is considered one of the few milestones in the deep learning revolution.
- It is simple yet really better than its predecessor activation functions such as sigmoid or tanh.

ReLU activation function formula

$$f(x)=max(0,x)$$

- ReLU function is its derivative both are monotonic.
- The function returns 0 if it receives any negative input, but for any positive value x, it returns that value back.
- Thus it gives an output that has a range from 0 to infinity.
- ReLU is used as a default activation function and nowadays and it is the most commonly used activation function in neural networks, especially in CNNs.
- The model can, therefore, take less time to train or run.
- One more important property that we consider the advantage of using ReLU activation function is sparsity.
- Since ReLU gives output zero for all negative inputs, it's likely for any given unit to not activate at all which causes the network to be sparse.
- ReLU does not face vainishing gradient problem as its slope doesn't plateau, or "saturate," when the input gets large.
- Due to this reason models using ReLU activation function converge faster.
- There are some problems with ReLU activation function such as exploding gradient
- Also, there is a downside for being zero for all negative values and this problem is called "dying ReLU."
- A ReLU neuron is "dead" if it's stuck in the negative side and always outputs 0.
- Because the slope of ReLU in the negative range is also 0, once a neuron gets negative, it's unlikely for it to recover.

Hyperparameter tuning:

Classification of Neural Network Hyperparameters

Classification by Network Structure

Directly affect the structure of the neural network

- Number of hidden layers
- Number of nodes in each layer
- Type of activation function

Classification by Learning & Optimization

Directly control the training process of the neural network

- Type of optimizer
- Learning rate-α
- Type of loss function
- Type of model evaluation metric
- Batch size
- Epochs
- Training steps (iterations) per epoch

Classification by Regularization Effect

Directly control the overfitting in the neural network

- Lambda-λ in L1 and L2 regularization
- Dropout rate in dropout regularization

How to choose a number of hidden layers

- One of the hyperparameters that change the fundamental structure of a neural network is the number of hidden layers, and we can divide them into 3 situations: 0, 1 or 2, many.
- We don't need to use the neural network at all if all we need is a linear boundary since the neural network is for solving complex problems.
- Second, if a data set isn't linearly separable, then we need a hidden layer.
- And normally, a single hidden layer is sufficient because the amount that a model improves by adding hidden layers isn't significant compared to the additional work you need to do.
- So in many simple applications, one or two hidden layers do their job.
- If we are trying to solve a complex problem such as object classification, then we need multiple hidden layers that apply different modifications to their inputs.

# of hidden layer	*When*
0	Linearly separable data
1~2	Nonlinear boundaries
many	Complex problem

- Finding an appropriate number is critical because too few neurons can lead to underfitting whereas too many can lead to overfitting plus longer training time.
- Empirically, it's best to use a number between input and output sizes, and the number changes based on how complex your problem is.
- If a problem is simple and the input and output relationship is clear, then about ⅔ of the input size can be a good starting point.

- But if the relationship is complex, the number can vary from the input size to less than twice the input size.

# of neurons in a hidden layer	*When*
⅔ of the input size	Simple input and output relationship
input size ~ 2 x input size - 1	Complex input and output relationship

Batch size, learning rate, epoch

- When batch size increases, each batch naturally becomes similar to the full data set because each batch starts to contain more observations.
- This means that each batch will not differ too much from others.
- Therefore, its noise will decrease, so it's logical to use a large learning rate for faster training time. In contrast, when we use a small batch size, noise increases.
- Thus, we use a small learning rate to offset the noise.
- Empirically, it has been shown that a large batch size could lead to poor generalization. In contrast, when we use a small batch size, the noise helps a network to escape a local minimum and leads to higher accuracy.
- It also tends to converge to a reasonable solution faster than a network with large batch size.
- So, in general, a batch size of 32 could be a good starting point, but this number really depends on your sample size, the complexity of a problem, and your computational environment.
- Therefore, using a grid search could also be appropriate.
- For the learning rate, we usually start with 0.1 or 0.01 or we can use a grid search from 0.1 to 1e-5.
- And when the learning rate is small, you need more iteration to find a minimum point.
- Epochs: Depending on a problem and random initialization, the number of epochs we need for convergence varies.
- We often set the number of epochs high and use early stopping so that the neural network stops training when an improvement coming from updating its weights does not pass a threshold.

Hyperparameter	*General starting point*
Learning rate	0.1 or 0.01 or grid search
Epoch	Early stopping
Batch size	32 or grid search

Batch Normalization:

1. Even after the model has been fully trained such that its training error is small, it exhibits a high test error rate. This is known as the problem of Overfitting.
2. The training error fails to come down in-spite of several epochs of training. This is known as the problem of Underfitting.

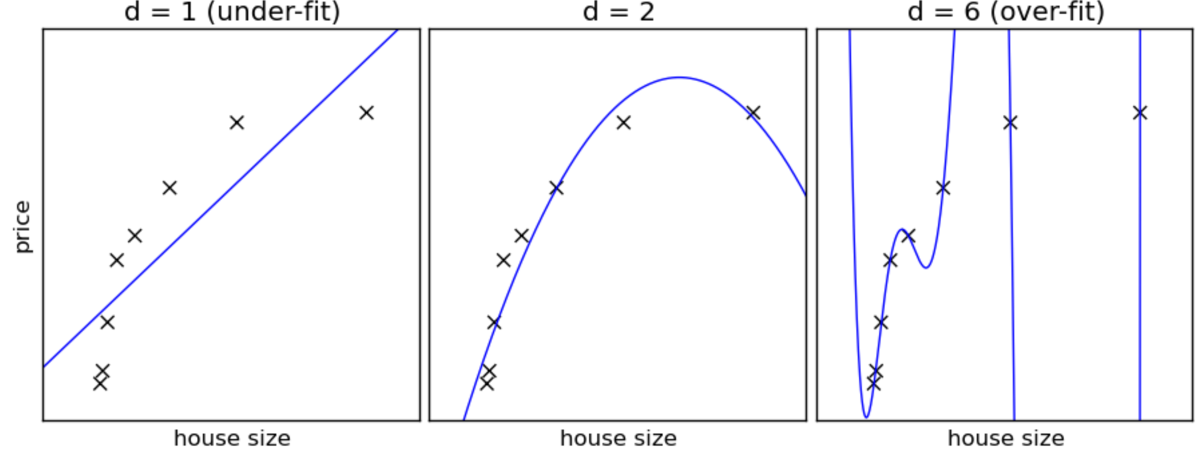
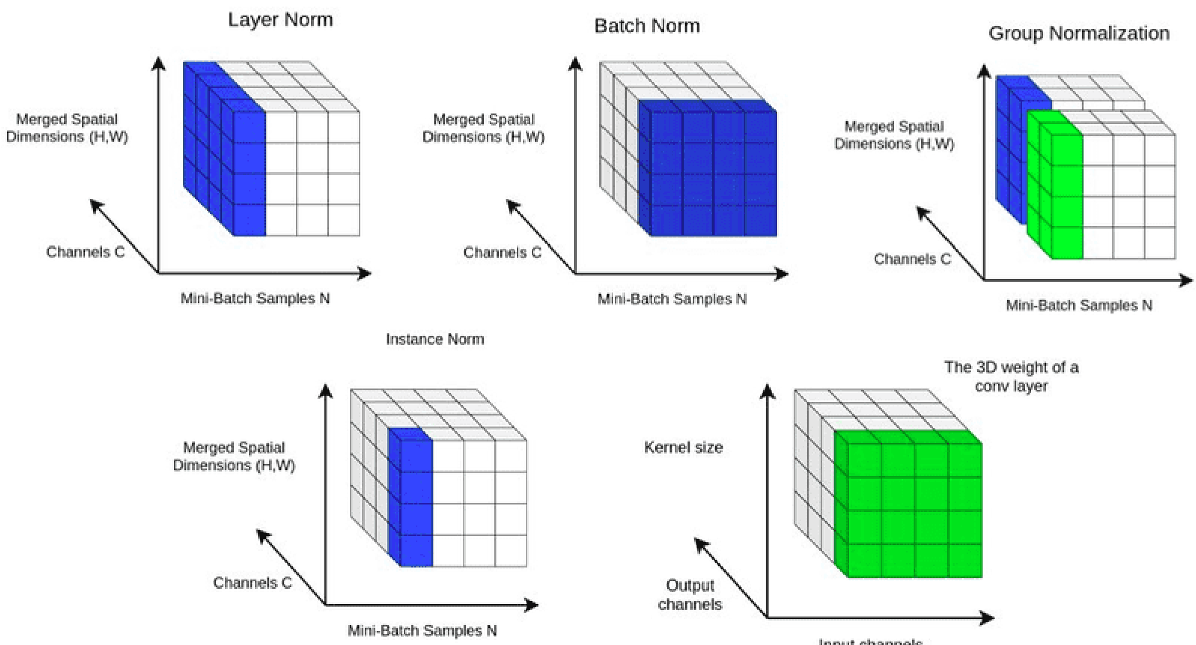


Illustration of Underfitting and Overfitting

Types of Normalization:



- Batch normalization is a deep learning approach that has been shown to significantly improve the efficiency and reliability of neural network models.
- It is particularly useful for training very deep networks, as it can help to reduce the internal covariate shift that can occur during training.
- Batch normalization is a technique used to improve the performance of a deep learning network by first removing the batch mean and then splitting it by the batch standard deviation.
- When applied to a layer, batch normalization multiplies its output by a standard deviation parameter (gamma) and adds a mean parameter (beta) to it as a secondary trainable parameter.
- The goal of batch normalization is to stabilize the training process and improve the generalization ability of the model.
- It can also help to reduce the need for careful initialization of the model's weights and can allow the use of higher learning rates, which can speed up the training process.

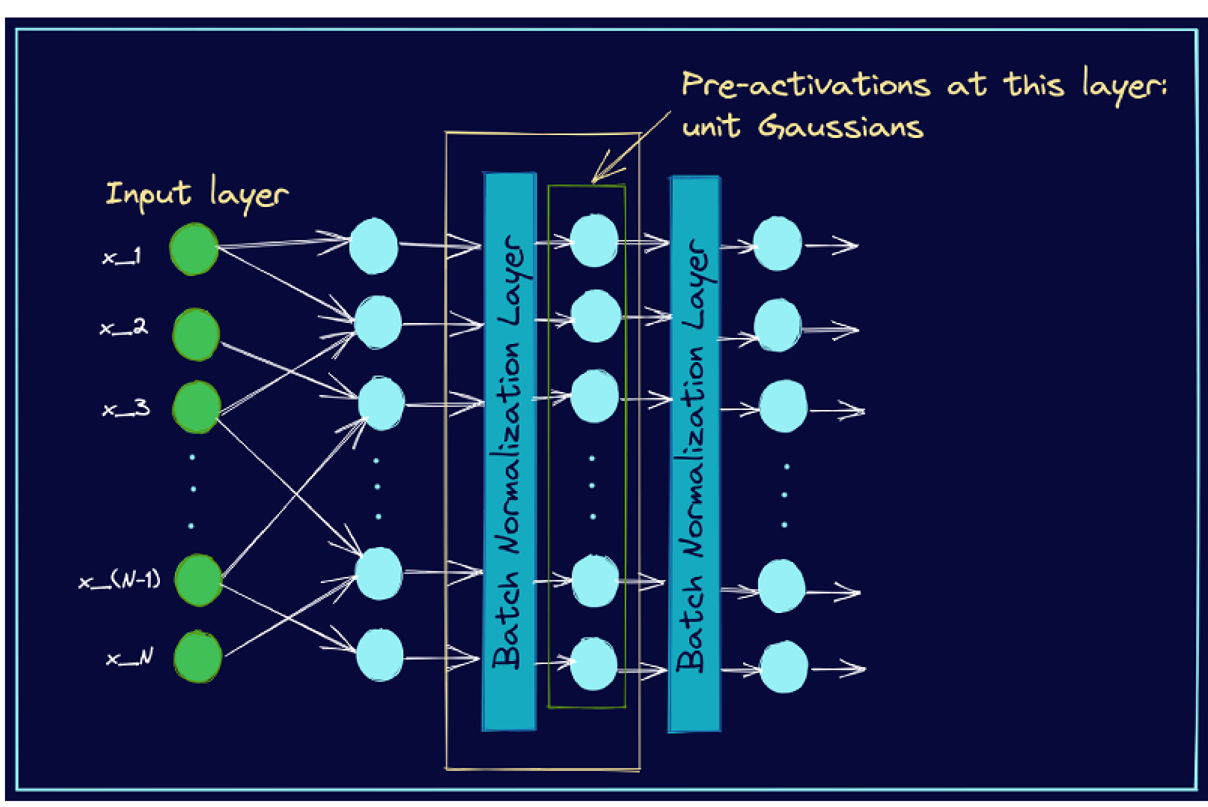
Batch normalization equations

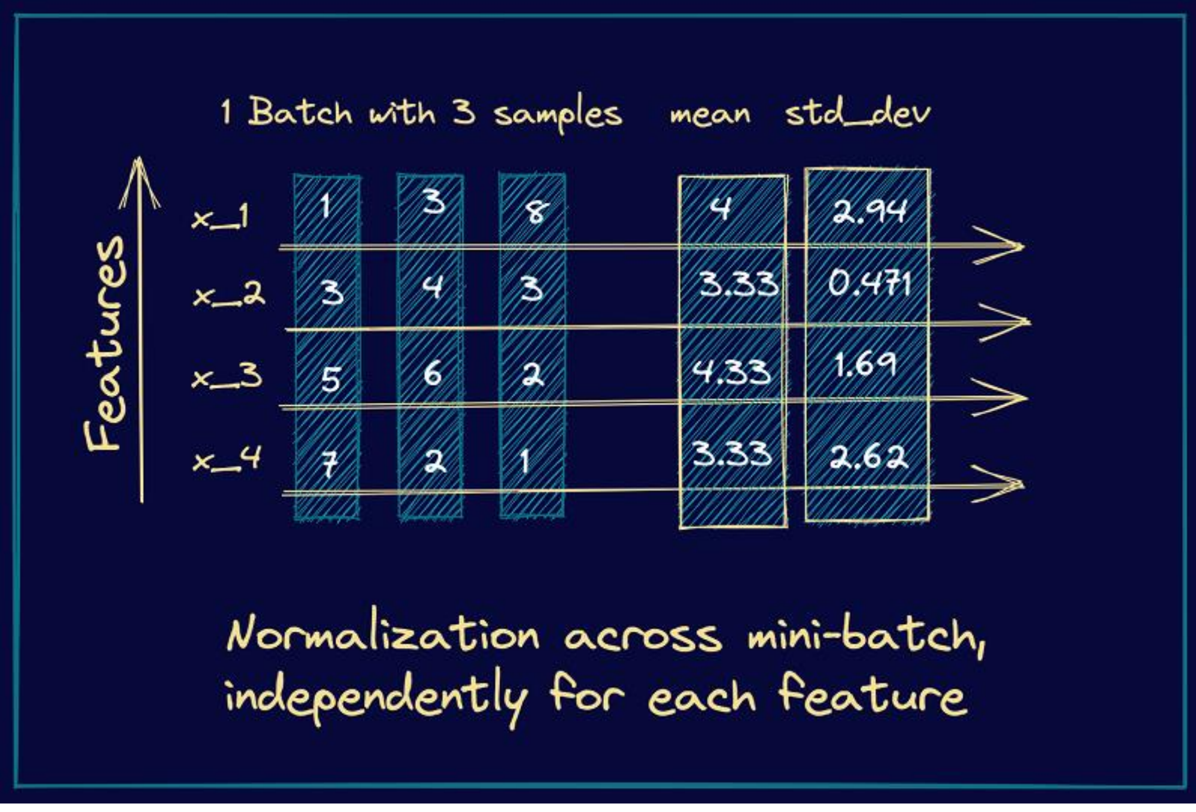
During training, the activations of a layer are normalized for each mini-batch of data using the following equations:

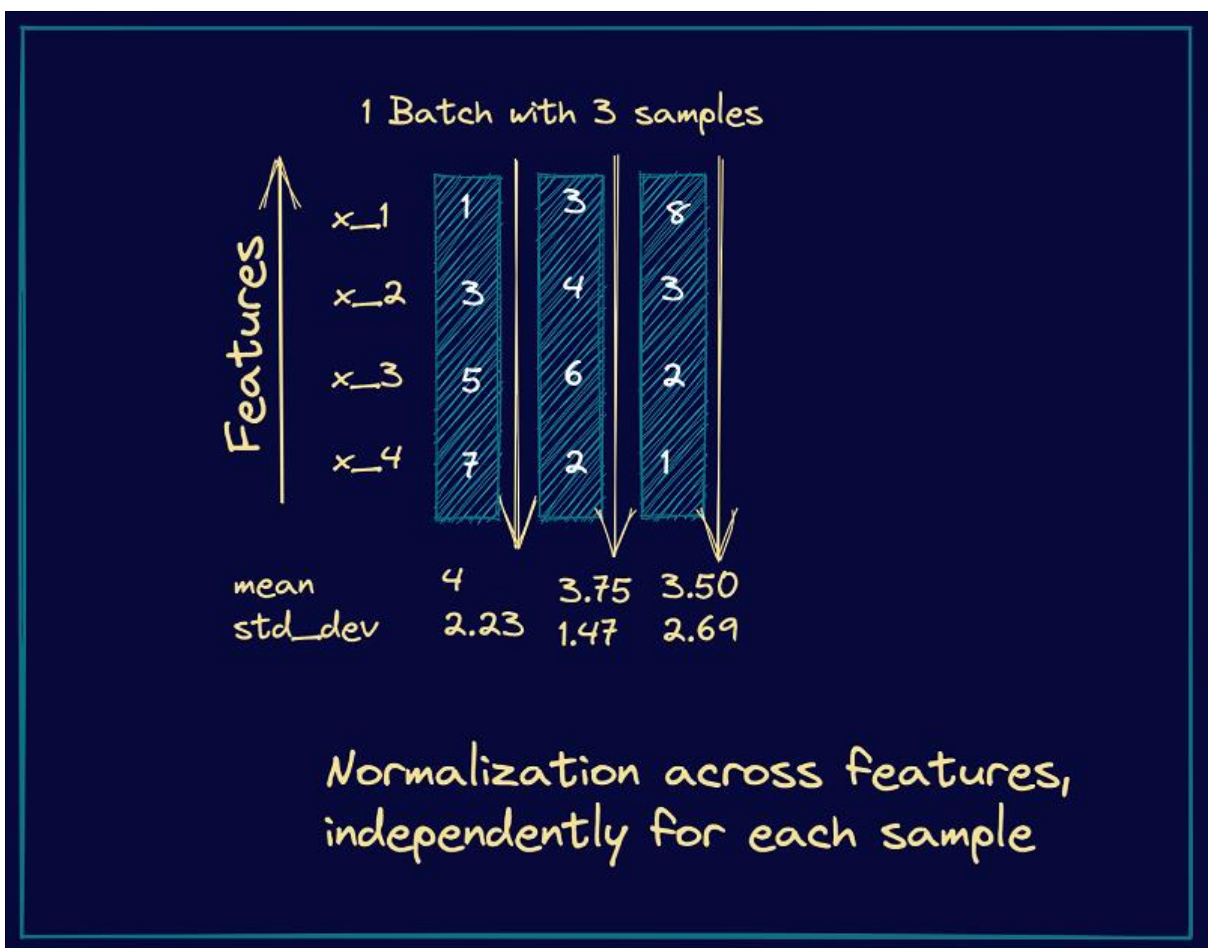
- Mean: $\text{mean} = 1/m \sum_{i=1}^{m} x_i$
- Variance: $\text{variance} = 1/m \sum_{i=1}^{m} (x_i - \text{mean})^2$
- Normalized activations: $y_i = (x_i - \text{mean}) / \text{sqrt}(\text{variance} + \varepsilon)$
- Scaled and shifted activations: $z_i = \gamma y_i + \beta$, where γ and β have learned parameters

During inference, the activations of a layer are normalized using the mean and variance of the activations calculated during training, rather than using the mean and variance of the mini-batch:

- Normalized activations: $y_i = (x_i - \text{mean}) / \text{sqrt}(\text{variance} + \varepsilon)$
- Scaled and shifted activations: $z_i = \gamma y_i + \beta$


Pre-activations at this layer:
unit Gaussians
Input layer
x_1
x_2
x_3
x_(N-1)
x_N
Batch Normalization Layer
Batch Normalization Layer


1 Batch with 3 samples
mean
std_dev
Features
x_1
x_2
x_3
x_4
1
3
8
4
2.94
3
4
3
3.33
0.471
5
6
2
4.33
1.69
7
2
1
3.33
2.62
Normalization across mini-batch,
independently for each feature



Advantages of batch normalization

- Stabilize the training process. Batch normalization can help to reduce the internal covariate shift that occurs during training, which can improve the stability of the training process and make it easier to optimize the model.
- Improves generalization. By normalizing the activations of a layer, batch normalization can help to reduce overfitting and improve the generalization ability of the model.
- Reduces the need for careful initialization. Batch normalization can help reduce the sensitivity of the model to the initial weights, making it easier to train the model.
- Allows for higher learning rates. Batch normalization can allow the use of higher learning rates that can speed up the training process.

Dropout Regularization

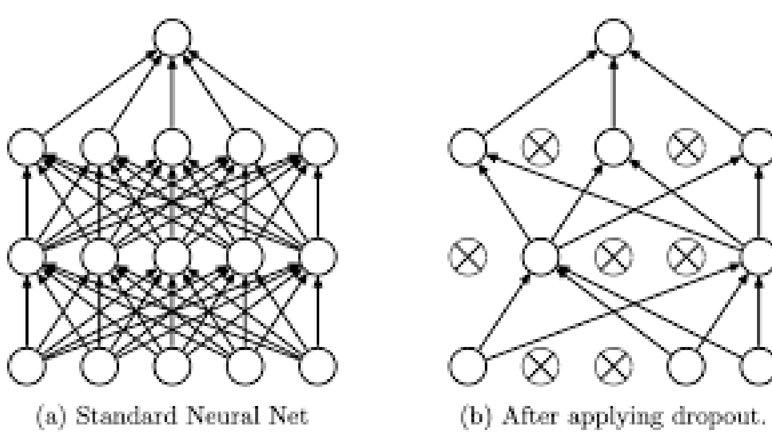
- When you have training data, if you try to train your model too much, it might overfit, and when you get the actual test data for making predictions, it will not probably perform well. Dropout regularization is one technique used to tackle overfitting problems in deep learning.
- DLNs also exhibit a little understood feature called Self Regularization.
- For example for a given amount of Training Set data, if we increase the complexity of the model by adding additional Hidden Layers for example, then we should start to see overfitting, as per the arguments that we just presented.
- However, interestingly enough, increased model complexity leads to higher test data classification accuracy, i.e., the increased complexity somehow self-regularizes the model.
- Hence when using DLN models, it is a good idea to start with a more complex model that the problem may warrant, and then add Regularization techniques if overfitting is detected.

Some commonly used Regularization techniques include:

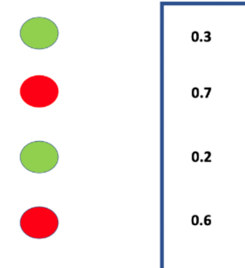
- Early Stopping
- L1 Regularization
- L2 Regularization
- Dropout Regularization
- Training Data Augmentation
- Batch Normalization

Training with Drop-Out Layers

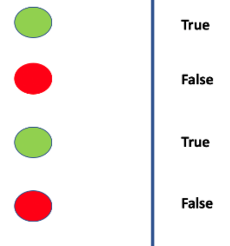
- Dropout is a regularization method approximating concurrent training of many neural networks with various designs.
- During training, some layer outputs are ignored or dropped at random.
- Dropout is an extremely versatile technique that can be applied to most neural network architectures.
- It is useful especially when your network is very big or when you train for a very long time, both of which put a network at a higher risk of overfitting.



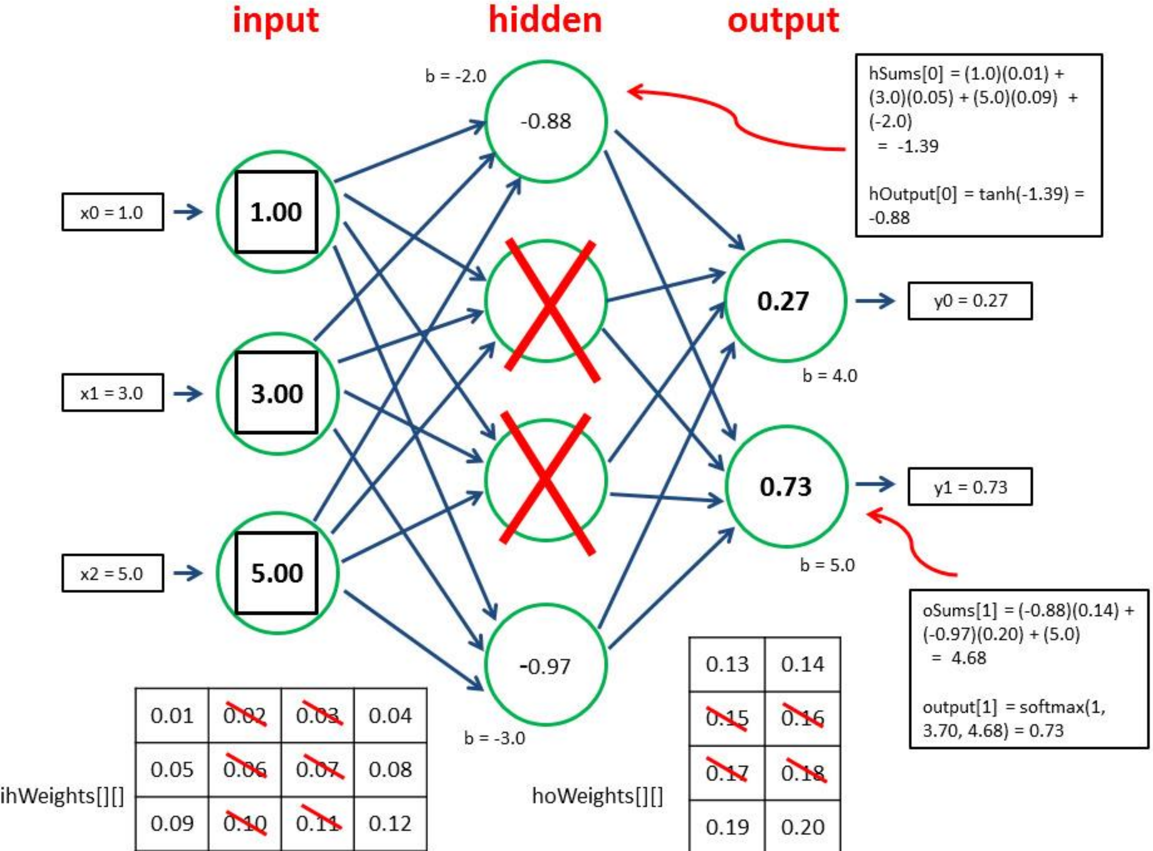
- To apply dropout, we need to set a retention probability for each layer.
- The retention probability specifies the probability that a unit is not dropped.
- For example, if we set the retention probability to 0.8, the units in that layer have an 80% chance of remaining active and a 20% chance of being dropped.
- Standard practice is to set the retention probability to 0.5 for hidden layers and to something close to 1, like 0.8 or 0.9 on the input layer.
- Output layers generally do not apply dropout.
- Dropout is applied by creating a mask for each layer and filling it with values between 0 and 1 generated by a random number generator according to the retention probability.



- We could also fill the mask with random boolean values according to the retention probability.
- Neurons with a corresponding "True" entry are kept while those with a "False" value are discarded.



True
False
True
False



input
hidden
output
x0 = 1.0
x1 = 3.0
x2 = 5.0
1.00
3.00
5.00
b = -2.0
-0.88
-0.97
b = -3.0
0.27
0.73
b = 4.0
b = 5.0
y0 = 0.27
y1 = 0.73
hSums[0] = (1.0)(0.01) + (3.0)(0.05) + (5.0)(0.09) + (-2.0) = -1.39
hOutput[0] = tanh(-1.39) = -0.88
oSums[1] = (-0.88)(0.14) + (-0.97)(0.20) + (5.0) = 4.68
output[1] = softmax(1, 3.70, 4.68) = 0.73
ihWeights[][]
0.01 0.02 0.03 0.04
0.05 0.06 0.07 0.08
0.09 0.10 0.11 0.12
hoWeights[][]
0.13 0.14
0.15 0.16
0.17 0.18
0.19 0.20