CSCD84

UNIT 1 - SEARCH

LEC 01

INTELLIGENT AGENT

Device / Algorithm / System that has a specific Goal to achieve. Achieve goal by:

- senses the environment: using sensor / being provided information about environment as input
- takes action: some actions that affect state of environment.

e.g. - change value of parameter that control agent actions
: make choice between actions it could take in the future
- make decisions based on what is happening in the environment
- goal: usually a utility function to maximize / loss function to minimize

e.g. smart thermostat

goal: keep temperature at certain level
Senses: current room temperature
actions: AC / heat on / off

↑ simplest type of intelligent agent; reactive, decision entirely based on current decision. NOT very smart

Usually, we refer intelligent agent to agents that rely on a model of the environment to make a prediction, what effect the agent's action would have and make decisions accordingly.

SEARCH PROBLEMS

Given a problem, and a concrete definition of how a solution to this ~~problem~~ problem is specified, search for best / correct solution.

components of a search problem:

- configuration: determine relevant variables of environment

  state of the problem: set of variables. each possible assignment of values is a configuration

  (represented as <u>vertices</u> for different possible configuration

- how actions changes current configuration:

  need to know how actions affect values of variables, helps us determine given the current configuration, what are possible future configurations depending what actions are taken.

  (represented as <u>edges</u> from one ~~configuration~~ configuration to another)

- goal state: desired outcome for the problem.

  determine whether reach goal state by looking at values of variables of a state.

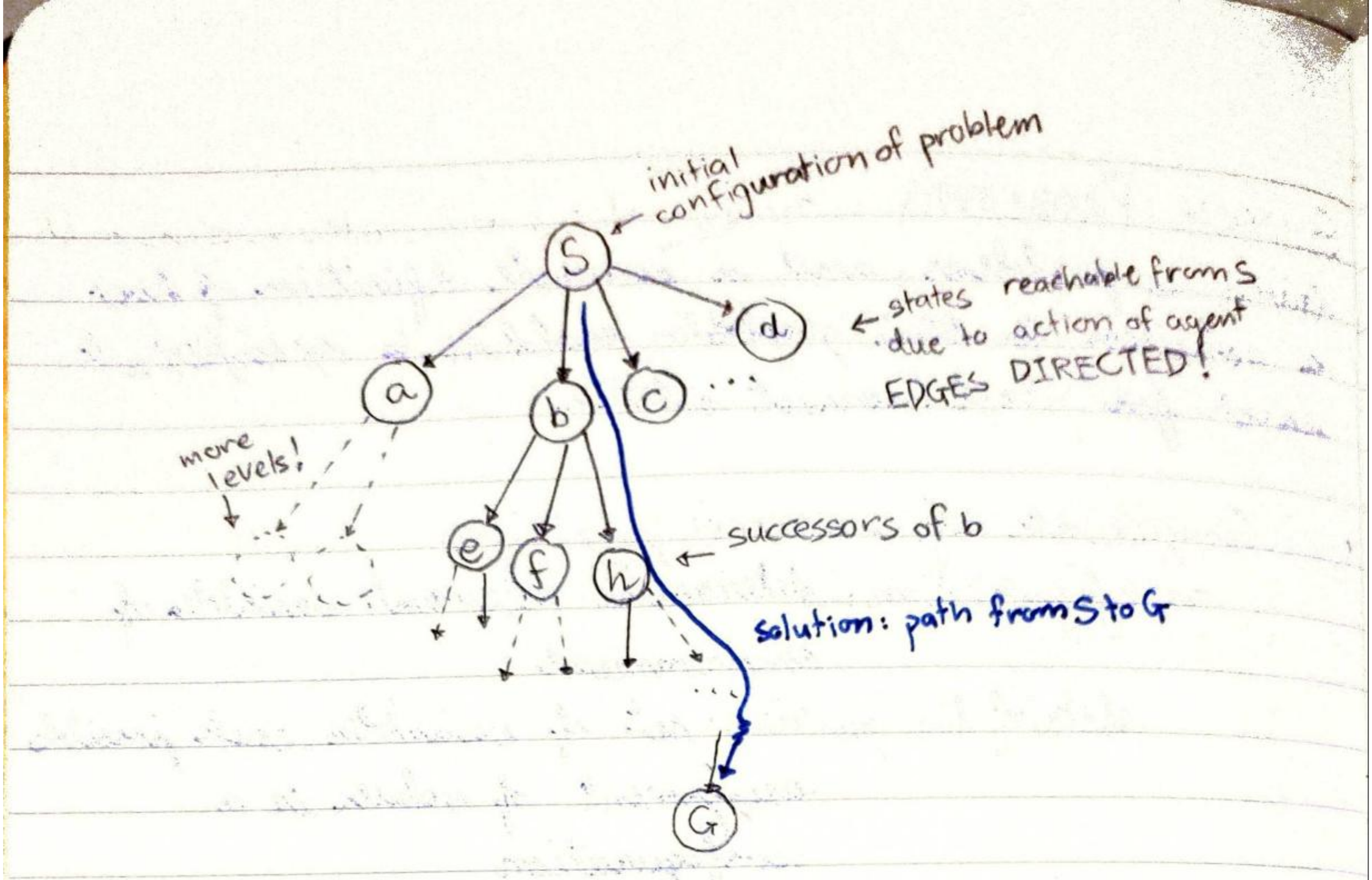
  can have multiple goal states

  (represented as subset of configuration vertices)

Search problem form TREEs.

Solution: sequence of actions from initial state → goal state

⇔ traversal in tree.



SEARCH TREE

- Root node: initial configuration
- Node: states / configurations
- Level: contains states reachable from nodes in previous level by possible actions agents can take
- Path: each path corresponds to a plan.
  sequence of actions that takes the agent from the initial state to other states in the tree.
  if final node is (a) goal node then plan is a solution for the problem.

- For most problems, search tree is too large to explore exhaustively

e.g. Problem: Finding a path between two specific locations

Definition of states: Possible locations of agent.
i.e. splitting map into a grid

Sucessor State: Locations that can be reached from current one.

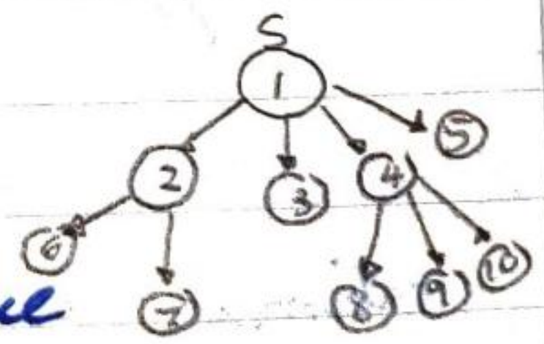
Initial State: Starting point for our path finding problem
Goal State: Destination location
* do not mix up search tree graph and map represented as graph!

SOLVING SEARCH PROBLEMS

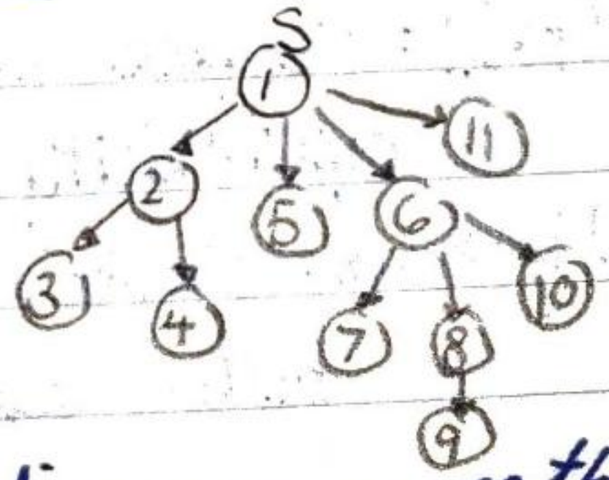
Initialize starting node (root of search tree)
Create an initially empty list of expanded nodes
Add starting node to the list

while list not empty:
    get next node* from the list
    if this node is a goal node
        success!: return path from start node to this goal node
    else
        add successor nodes for this node to the list in some order**
list is empty and didn't find a goal: failure! (return empty path)

- BREADTH FIRST SEARCH (BFS)
  ** = to queue (i.e. at the end)
  * = node at the front of the queue
  - nodes explored in order of levels



- DEPTH FIRST SEARCH (DFS)
  ** = to stack (i.e. to the top)
  * = node at the top of stack
  - usually done with recursion, discovers a path fully first



GENERALIZING SEARCH

Usually different actions have different cost.

e.g. Road A 1km speed limit 50Km/hr congested
Road B 1km speed limit 80Km/hr → "less cost"

↳ assign different weights to edges in search tree.
- optimal solution is desired

- UNIFORM COST SEARCH (UCS)

** = to priority queue (in increasing cost from s to n)
* = node in front of priority queue (current least cost)
↳ when expanded, add neighbours with cost from s to neighbour.
- update neighbour in queue if cost for neighbour is better than in queue

✡ lowest cost path. If uniform cost for edges, equivalent to BFS.
- similar to Dijkstra's, but stops when a goal found.
✡ guarantees optimal path to a goal given all weights are nonnegative.

HEURISTIC SEARCH

- optimal is great, but some problems can be too big to do it. ✡ need faster methods
Goal: reduce computational burden without sacrificing optimality
Method: make search process explore more promising paths earlier than other paths

- A* SEARCH

makes use of heuristic function to guess which nodes in search tree are likely to be closer to goal state.

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

$f(n)$ - heuristic cost for node n
$g(n)$ - actual cost from initial node s to n
$h(n)$ - guess of cost to get to goal state from n.

* algorithm same as UCS! only difference is inclusion of $h(n)$
- don't know $g(n)$ in advance! only known when added to priority queue, and can get smaller as algorithm continues

Heuristic is evaluated at each n and must be <u>admissible</u> to guarantee optimality of UCS.
↳ $\forall n, 0 \leq h(n) \leq h^*(n)$,
where $h^*(n)$ is true cost from n to goal

* tricky! The reason we need $h(n)$ is we don't know $h^*(n)$ but $h(n)$ cannot exceed $h^*(n)$.
+ the closer to $h^*(n)$, the better.

If we overestimate $h(n)$, chances are we missed a node that could be considered in optimal path.
$f(g) = g(g) + h(g) = g(g)$ as g is goal node. UCS stops when expand g with cost $f(g)$. If exceed that, nodes are not considered.

UNIT 2- CONSTRAINT SATISFACTION PROBLEMS

CONSTRAINT SATISFACTION PROBLEM

Also a search problem, related to finding solution assigning values for a set of variables, in a way that satisfies predefined constraints.

e.g scheduling courses to classrooms and time-slots such that:
- courses taught by same instructor don't happen at the same time
- courses commonly taken together don't happen at the same time
- courses are assigned to classroom of appropriate capacity

Components of a CSP:
- set of variables
  - to assign value to
  - represented as nodes
- domain for each variable
  - available values for each variable
- set of constraints
  apply to individual variable or relate subsets of variables for validity of assignment
  e.g. D84 cannot be scheduled at noon because instructor has lunch at noon
  D84 cannot be scheduled same time as A48 as they are taught by same instructors
  - represented as edges

in this unit:
- variables are categorical
- finite domain
  ↳ becomes combinatorial optimization problem

BACKTRACKING SEARCH

variation of DFS. keeps track of assignments of values to variables.
goal: generate complete assignment without breaking constraints

Input: set of variables csp_variables
set of constraints csp_constraints
initial assignment as a set with tuples
<variable: value> (can be empty or not depending on given conditions)

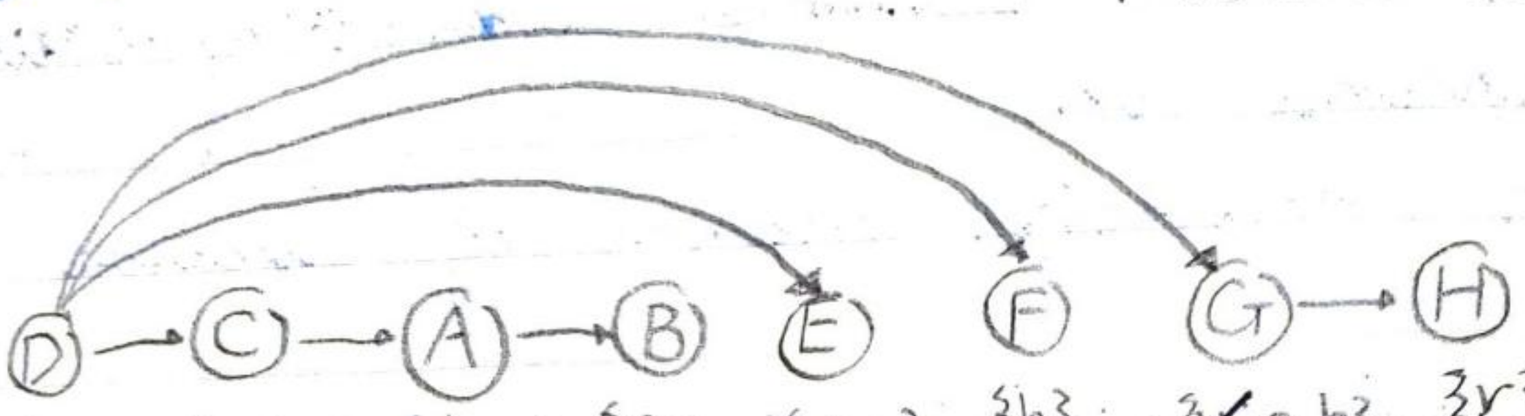
```
recursive_backtracking (assignment, csp_variables, csp_constraints)
    if is_solution (assignment) {
        return success
    } else {
        var := select_unassigned (&csp_variables)   // choose variable to assign
        foreach value in var.domain {               // try each possible value
            add <var:value> to assignment
            if constraint consistent (assignment, csp_constraints)
            { result = recursive_backtracking (assignment, csp_var, csp_con)

                if result == success { return success }
            }
        remove <var: value> from assignment
    }
    return failure
}
```

* After backward pass, all remaining values for variables have leaves at least one valid choice for child nodes.

3. SEARCH! Start at root and assign a value for each node.
   * will not backtrack because value from parent is consistent with constraints on their children.
   - if parent's domain size = 0 then no solution

e.g.



2. D remains same as any choice will leave G at least one choice

3. remove b from D because of F

4. D remains same for E.

5. A unchanged

6. C unchanged

7. remove y from D because C

1. remove r from G because child H will have invalid assignment if G=r

8. starting from D assign D=r, C=y, A=b, B=g, E=g, F=b G=g, H=r without needing to backtrack.

CUTSET CONDITIONING

Remove active constraints from original CSP to transform it to tree-structured
- by assigning a subset of variables some values before beginning search.

- there's no guarantee initial assignment will yield feasible solution!

Complexity in which we fix value for k variables of n variables that has domain size of d to obtain a tree-structured C.S.P:

$O(d^k (n-k) d^2)$

ITERATIVE METHODS AND APPROXIMATE SOLUTIONS

- backtracking too costly / want a sufficiently good solution in much less time.

LOCAL SEARCH : Simple and general

Randomly generate a full assignment of values to variables in the CSP

iterations = 0

while iterations < MAX_ITERATIONS
    randomly pick a variable that has conflicts
    randomly select a new value for selected var
    if new assignment is a solution: return
    else if new assignment breaks fewer constraints
        keep assignment
    else
        keep old assignment instead
    iterations ++

local because only one variable changes each time and is close to previous assignment

- searches in neighbourhood of possible solutions for something better

- hill climbing : improve solution over iterations
- fast because runs iterations with very little time because of how simple it is

* problem: can get stuck at local maximum where it cannot be improved further (like maximal vs maximum)

IMPROVEMENTS TO LOCAL SEARCH

1. depending on the problem, be smart with picking variable and value
2. run local search k times. keep best one
3. Deterministic Annealing : variation of local search to ~~pick~~ keep worse assignment by some probability that decreases over runs. This is to allow become unstuck and potentially reaching a better local max.
4. Beam search / Taboo search, keep track of multiple guesses found and attempt to maximize chance of finding a good solution.
5. Genetic Algorithms / Ant colony optimization

UNIT 3 - ADVERSARIAL GAMES

DEFINITION

2-player games (for this course)
- both players try to win at some predefined task with a set of predefined rules
- players take turns making a move that will be with the goal of winning the game.
- competitive, one either wins or loses.

STATE - possible configuration for the game
INITIAL CONFIGURATION - root node of search tree
LEVELS IN SEARCH TREE - possible configurations reached from making a move on a configuration one level above
UTILITY FUNCTION - a function that evaluates how good the configuration is for each player.
+ve ⇒ good for player1 / bad for player2
-ve ⇒ bad for player1 / good for player2

e.g. tic-tac-toe
+1 for p1 wins
-1 for p2 wins
0 for tie

MINIMAX

For player 1's turn, they will choose a move that maximizes the value of the utility function
On player 2's turn, they will choose a move that minimizes the value of the utility function

If expanding a MAX node
when child node returns a utility value
> current alpha
update alpha := child utility
if alpha ≥ beta, return alpha

If expanding a MIN node
when child node returns a utility value
< current beta
update beta := child utility
if beta ≤ alpha, return beta

✡ MAX:
alpha = current best for self
beta = current best for parent

✡ MIN:
alpha = current best for parent
beta = current best for self

GAMES WITH CHANCE

Instead of normal utility, replace with expected utility:

i.e. weighted average of utilities for all the possible outcomes

e.g. for one move, there can be 3 outcomes

a. utility = +10, p = 0.1 ⇒ 10% chance +10

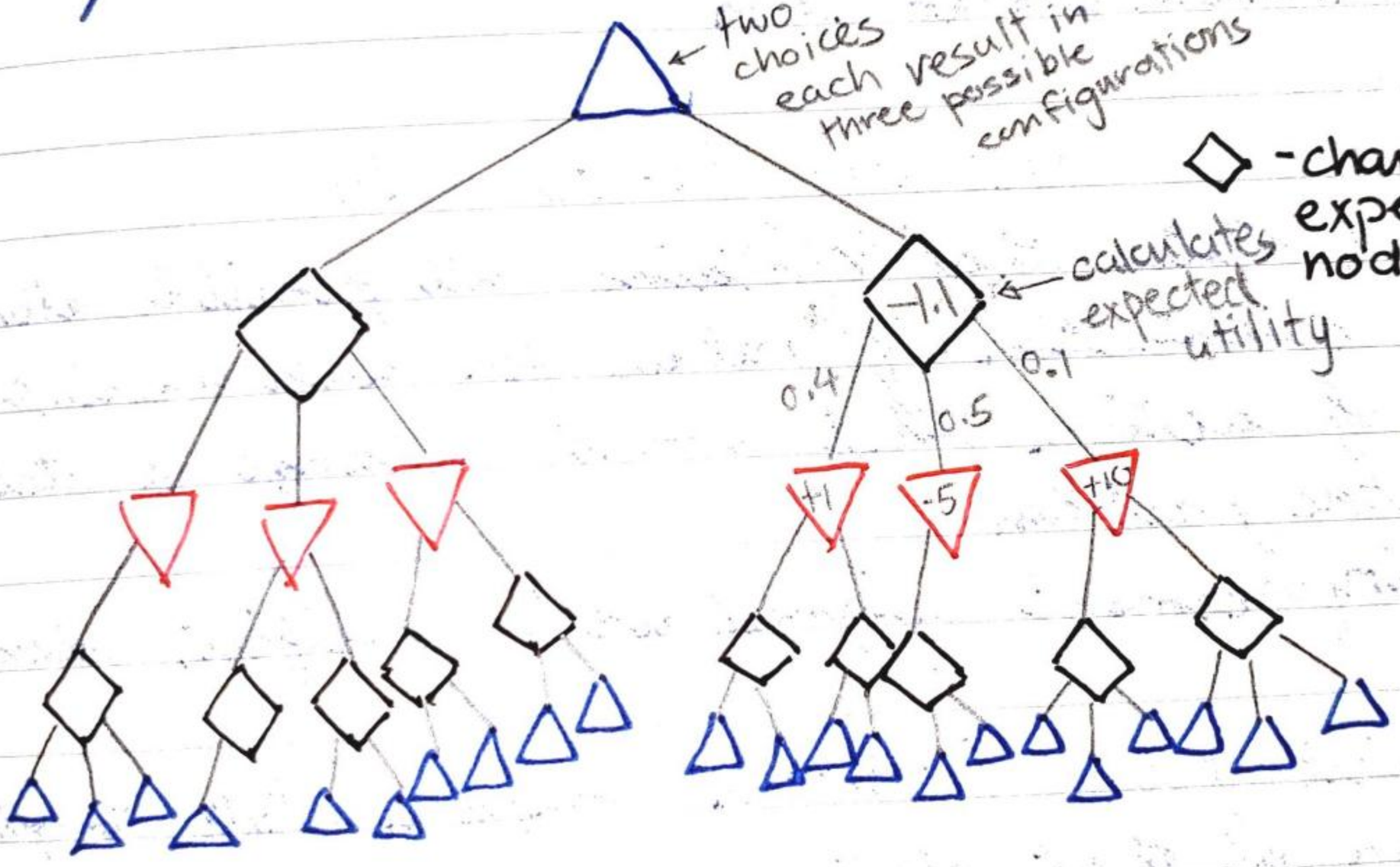
b. utility = -5, p = 0.5 ⇒ 50% chance -5

c. utility = +1, p = 0.4 ⇒ 40% chance +1

∴ expected utility = (10 × 0.1) + (-5 × 0.5) + (1 × 0.4)

= -1.1

This process becomes EXPECTI-MINIMAX



UNIT 4 REINFORCEMENT LEARNING

DEFINITION OF PROBLEM

- no precise definition of goal
- no objective function or heuristic function
- agent can observe variables related the the state of the environment
- problem evolves in discrete time steps

i.e. peform discretely:

1. choose action
2. observe outcome
3. receive reward (helpful action) or penalty (unhelpful action)
4. choose another action

SET UP - MARKOV DECISION PROCESS (MDP)

S = set of states that contains all possible states (all combination of values for each variable)

A = set of actions that would change a state in the problem

r = reinforcement signal with specified domain

GOAL: find policy $\Pi : S \rightarrow a$ maps a state to optimal action at that state.

* assume state transitions non-deterministic: for every state s and action a, the probability that results to state s` is T(s,a,s`)
* assume environment stationary as dynamic environments result in largeh sets of state variables to the point computationally impossible

ultimate goal for agent: maximize expected sum of rewards over time based on actions chosen by agent at each time step.

$$E\left[\sum_{t=0}^{\infty} \gamma^t r_t\right] \quad \text{where } \gamma \in (0,1)$$

γ: discount factor so that future rewards are worth less

ALGORITHM

For all s in S ← all states
  set V(s) := 0 // expected reward of starting at state s and following optimal policy
  for all a in A ← all actions
    set Q(s,a) := 0
While policy is not good enough // Q(s,a) has not converge or success rate of task after some training not good enough
  for all s in S
    for all a in A
      $Q(s,a) := R(s,a) + \gamma \sum_{s'} T(s,a,s') V(s)$
  $V(s) := \max_a Q(s,a)$
  $\pi(s) := \text{argmax}_a Q(s,a)$

immediate reward; all possible next states; transition probability; best value of Q(s,a) s constant; best action for state s.

VALUE ITERATION PROCESS

Above process has V(s) approximate optimal value for state s $V^*(s)$, and policy corresponds to optimal action that should be taken from each state to maximize rewards.

converges over time.

complexity for each iteration $O(|A||S|^2)$

Problem: T(s,a,s') unknown or too large to store!

Q-LEARNING

No transition probabilities required

Let algorithm explore possible outcomes of different actions taken from each state s, and arrive at good policy over time.

Estimates $Q^*(s, a)$

```
for all s in S
    for all a in A
        Q(s,a) := 0
    Π(s) := random a in A
determine initial state s for agent
alpha := small value.   //learning rate, problem dependent
                        ↑ too small takes long to converge policy. too large Does not converge properly
                        ~ reasonable [0.01, 0.1]
for j in 1 to k         // k rounds of training
    p-random := 1 - (j/k)   // start close to 1.0, decrease toward
                            ↑ initially allow exploration
later refine good policy    // 0 as j approaches k
    for i in 1 to M     // each round has M training actions
        c := rand [0, 1]
        if c <= p-random
            a := random valid action at state s
        else
            a := current known optimal action at s (Pi(s))
        Q(s,a) t+= alpha [r + γ(max_a' Q(s', a')) - Q(s, a)]]
        Π(s) = arg max_a Q(s, a)   //Update policy at end of each round.
```

2 key processes:

- Random exploration of state space: allowing agent to choose random actions given their current state and update Q table based on result of action (not determining possible future states

anymore)
- Relies on LOTS of training trials (large number to allow agent properly explore state space, as same state and same action does not necessary lead to same outcome)

- Q-learning allows us to come up with a good policy without worrying about state transition probabilities.

Problem: - many problems have large state space where large amount of training makes it impractical. (can't even store Q(s,a))

Problem: problem has policy good for it specifically
Does not generalize well.
e.g. robot moving around room with some furniture and objects in it
policy would be trained specific to room configuration

FEATURE BASED Q-LEARNING
Allows us deal with problems with larger state space and build a policy to solve similar problems

Goal: extract useful features from state variables (problem dependent)
e.g. maze with 1 mouse, cats and cheeses
- some form of distance to a cat
- some form of distance to cheese

- some form of information regarding configuration of the maze around the mouse

* feature not specific to particular configuration so that policy could still work on a similar configuration that it is not trained on.

After finding the features, training process is straightforward:

Q-table replaced by:

$$Q(S) = \sum_{i=1}^{k} w_i f_i(S) = w_1 f_1(S) + w_2 f_2(S) + \cdots + w_k f_k(S)$$

f_i = function of feature i $f_i(S)$ = feature i at state S

w_i = weighting for feature

Function determines whether ~~it~~ being ~~in~~ ~~st~~ at state S is good ~~of~~ or bad for agent.

Goal of process: estimate the weights so $Q(S)$ can provide information about how good or bad state is.

Policy depends on learned weights for features that apply to any state we encountered, or may encounter in the future, not states! ⇒ don't care about size of state-space.

Initialize w_i = random([-0.5, 0) U (0, 0.5]) for all i = 1 to n (n features)

For j in 1 to K // perform K rounds of training

p-random = $1 - \frac{j}{K}$ // starts close to 1.0, decreases towards 0 as j approaches K

For i in 1 to M

pick c = random([0, 1])

if $c \leq p\_random$

choose random action a from valid actions at state s

else

choose a to be current known optimal action in $Pi(S)$: // policy

given state s, for each possible action a_i in S

determine feature values after a_i action taken

$Q(S_i) = \sum_{j=1}^{n} w_j f_j(S)$ where S_i is resulting state

choose a_i which has largest $Q(S_i)$

// with action a, receives instantaneous reward r

// state change to s' through observation

// update each w_i as follows

$w_i \mathrel{+}= \alpha [r + \gamma \underline{Q(s')} - Q(s)] f_i(S)$

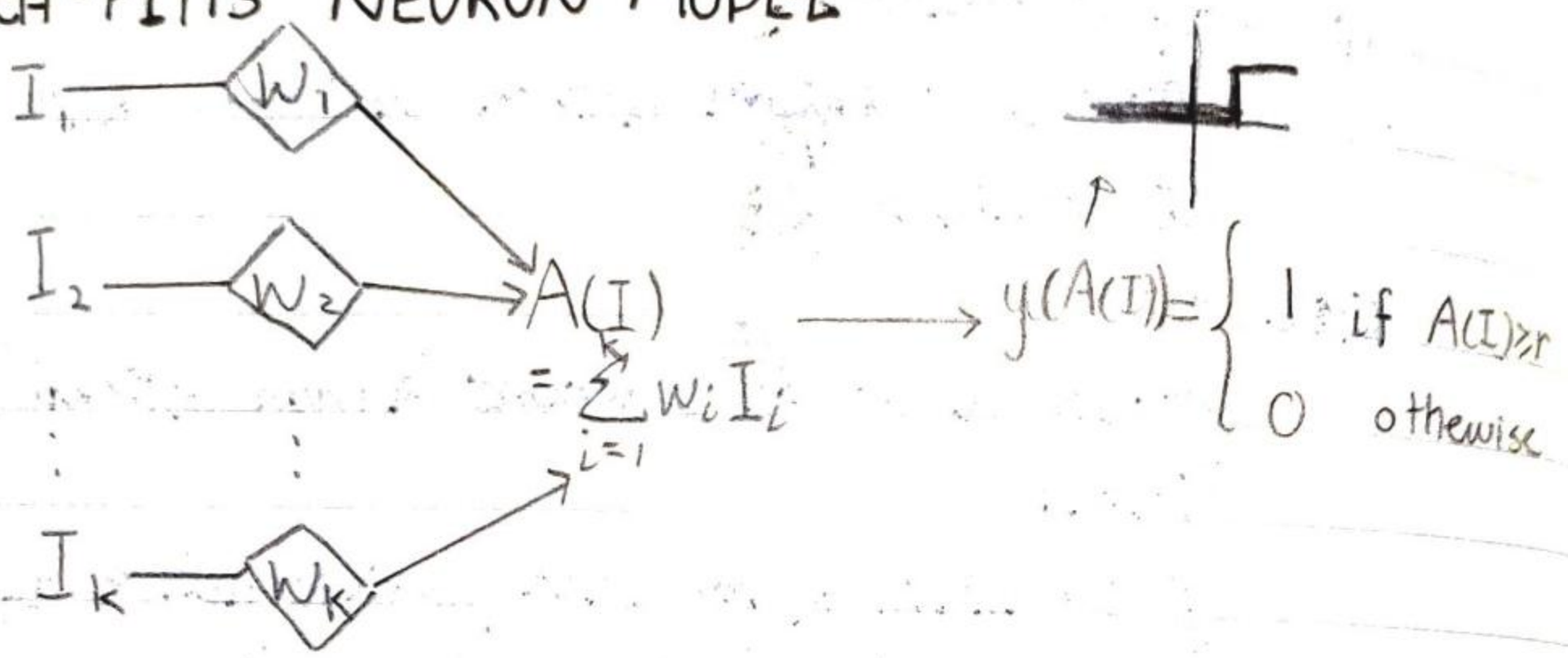
max Q(s')

Form of gradient descent: iteratively approximate weights that allow the agent to maximize its expected rewards.

UNIT 5 - NEURAL NETWORKS

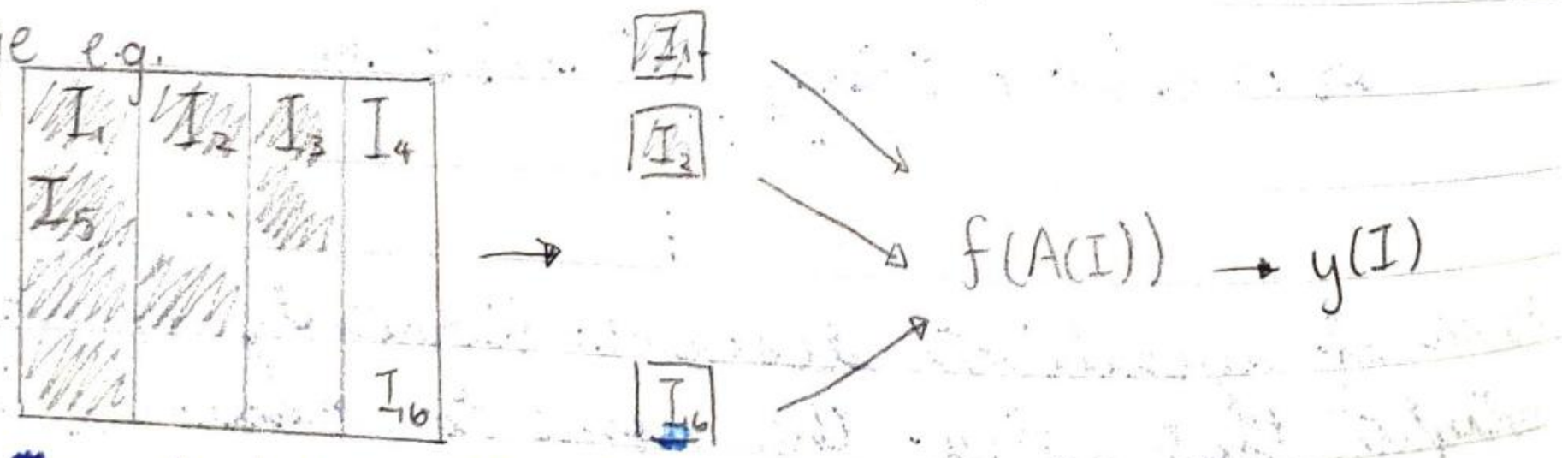
Based on idea that human brain can learn and carry out complex tasks because of the organization and connection of billions of simple processing units (neurons).

MCCULLOCH-PITTS NEURON MODEL



1. Computes weighted sum of input values → activation A(I)
2. Processes activation through activation unit
3. Determines value of output by applying activation function to activation

Usage e.g.



pixel "i shaded $\Rightarrow I_i = 1$, else $I_i = 0$

goal: $y(I) = 1$ when image has pattern of "P"

idea: $w_i = 1$ if we want $I_i = 1$

$w_i = -1$ if we want $I_i = 0$

So neuron can recognize this specific pattern perfectly.

larger image ⇒ provide larger input

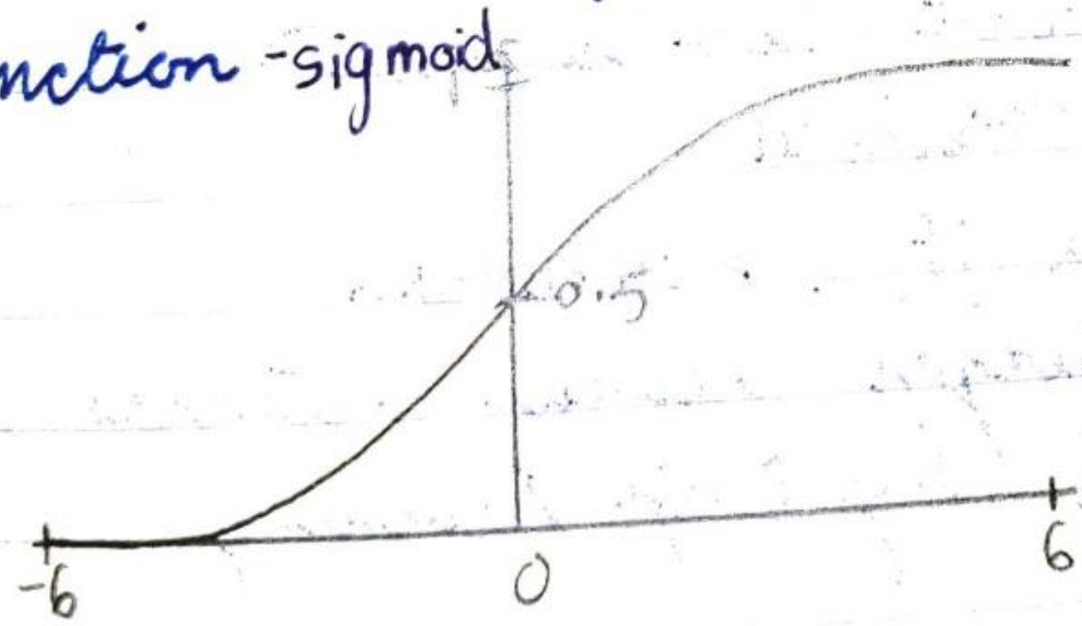
Problem:
- In realistic examples, pixels that are 'on' or 'off' will not look exactly the same
- The neuron has to learn which pixels should be more often 'on' or 'off' and which don't matter
We want to be able to adjust weights by training on samples in order to learn perform well in any given pattern

ACTIVATION FUNCTIONS

Threshold activation function

not often used in practice - to implement learning process we need a differentiable function

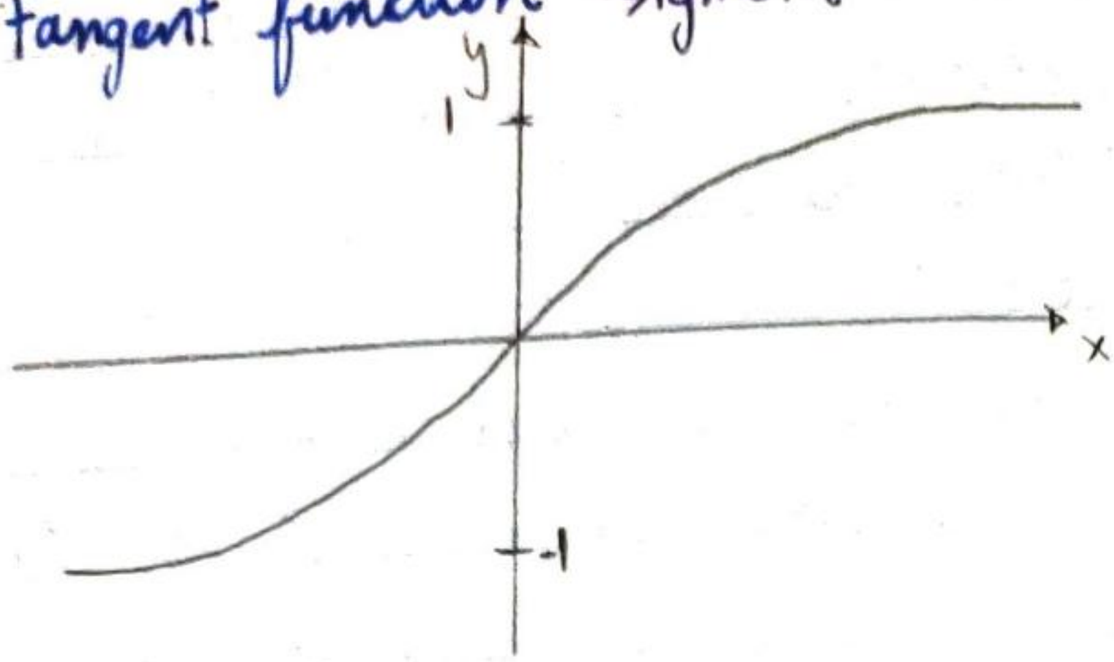
Logistic function - sigmoid



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

behaves like soft threshold:
much smaller than threshold: output close to 0
much greater than threshold: output close to 1
close to threshold : output increases from 0 to 1

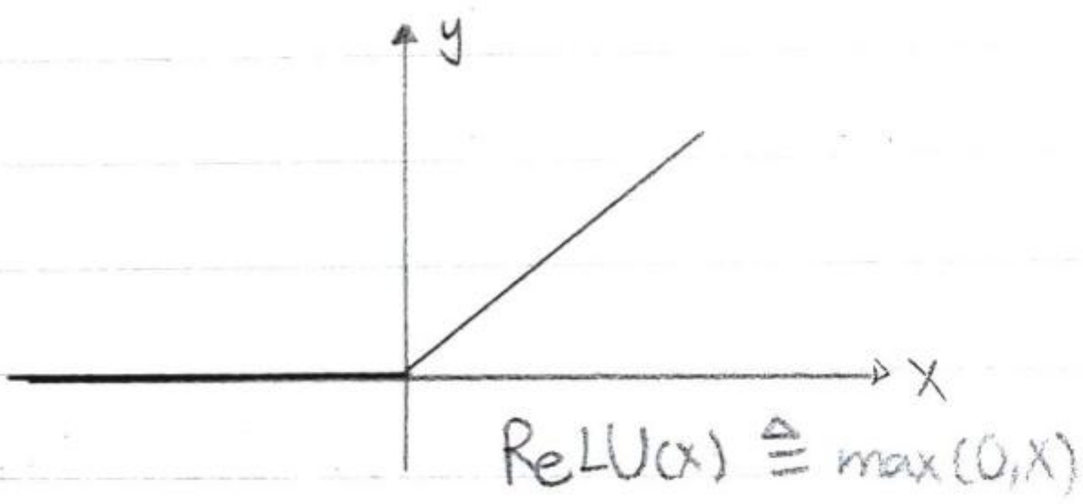
hyperbolic tangent function - sigmoid

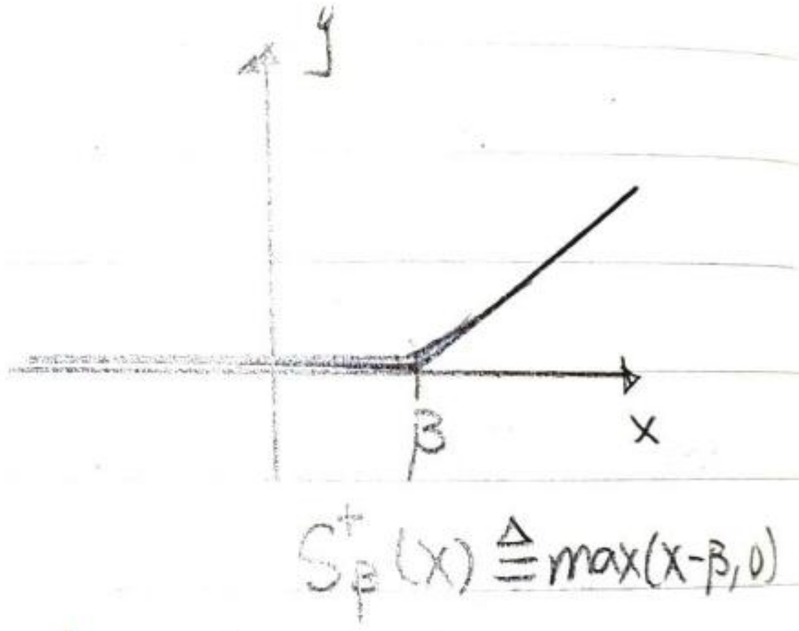


$$f(x) = \tanh(x) = \frac{e^x - e^{-x}}{e^x + e^{-x}}$$

similar shape as the logistic function but values range from -1 to 1 which has advantages for some tasks.

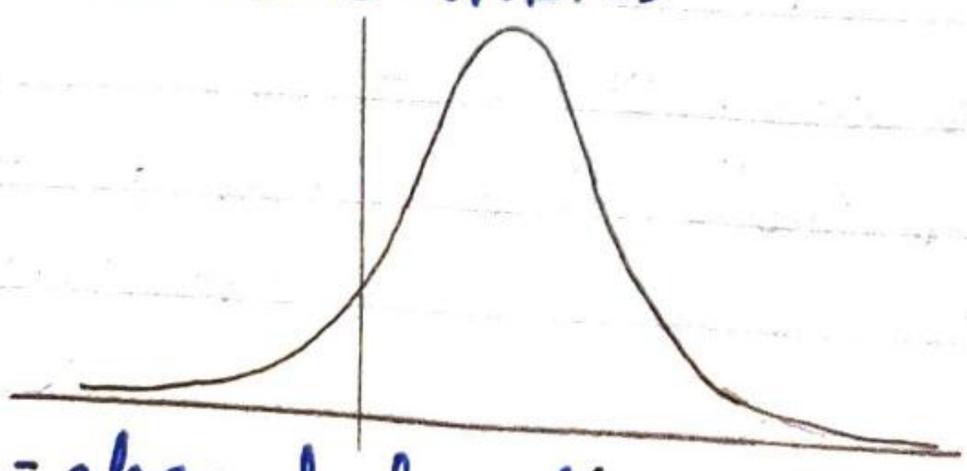
rectified linear unit (ReLU)





soft threshold unit. Two segments.
0 input < threshold
linear increase with x if x > threshold
often used in larger neural networks such as those employed by Deep Learning

radial basis functions (RBFs)



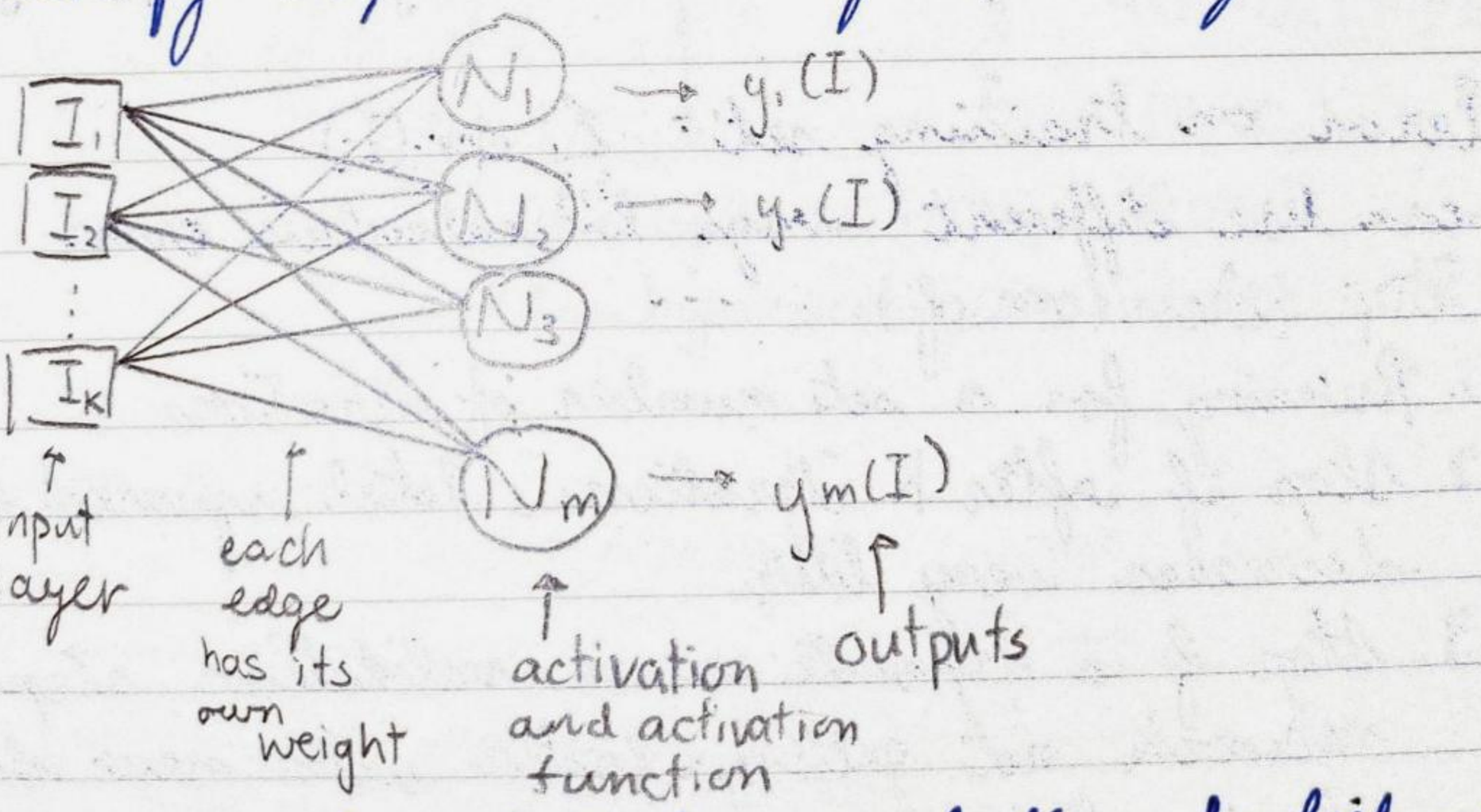
Gaussian-shaped functions used for problems in which spatial layout or pattern of an input importa

SINGLE LAYER FEED FORWARD NEURAL NETWORK

set of artificial neurons all connected to same set of inputs
have their own output

e.g. recognizing hand-written characters.
each neuron responsible for recognizing one character.
output should be large when input is the corresponding character
should be small when different character in input

By comparing output of all neurons, picking neuron with largest output for given input
⇒ identify input with high accuracy.



Each neuron independent from eachother of which will be trained to look for a different pattern
training process type of RL.
Require: annotated training set so we can train with samples where we know outputs
PROCESS: adjusting weights based on error.

Training Loop

Until error on training set* is small

For each input I_i on training set**

Feed-forward pass: show the input I_i to the network, and compute outputs for all neurons (on a multi-layer network, this is done by layer from those closest to the ~~tray~~ input and proceeding toward the output layer)

At output layer: compute error*** between neuron's output and expected output for that neuron

Adjust network weights to reduce error*** on this training sample.

* Common one? squared error $err_j(I_i) = (T_{ji} - O_{ji})^2$

(I_i: input, T_{ji}: target, O_{ji}: output (actual))

Error on training set = $\sum_{i,j} err_j(I_i)$

can use different ways to calculate error

Stop when (one of):

1. Running for a set number of iterations
2. Stop if after K iterations, total squared error decreases very little
3. Stop if a separate cross-validation step shows network not getting better with new data during training

** how to loop?

1. loop on every input in training set and update each time - converge slow
2. batch updates - group inputs into subsets of k and accumulate squared error before adjusting weights common

3. stochastic batch update - each time subset of k inputs chosen randomly - network will be ~~result~~ resilient to accidental patterns in training data; over-fitting

*** same as *

***BACK PROPAGATION

update weight using:

$$w_{ab} = w_{ab} + \alpha \frac{\partial Err_b}{\partial w_{ab}}$$

(α: learning rate)

where input = I_a

output = O_b

$$\frac{\partial Err_b}{\partial w_{ab}} = \frac{\partial A(I)}{\partial w_{ab}} \cdot \frac{\partial O_b}{\partial A(I)} \cdot \frac{\partial Err_b}{\partial O_b}$$

$\frac{\partial A(I)}{\partial w_{ab}} = I_a$ from $A(I) = \sum_{i=1}^{R} w_i I_i$

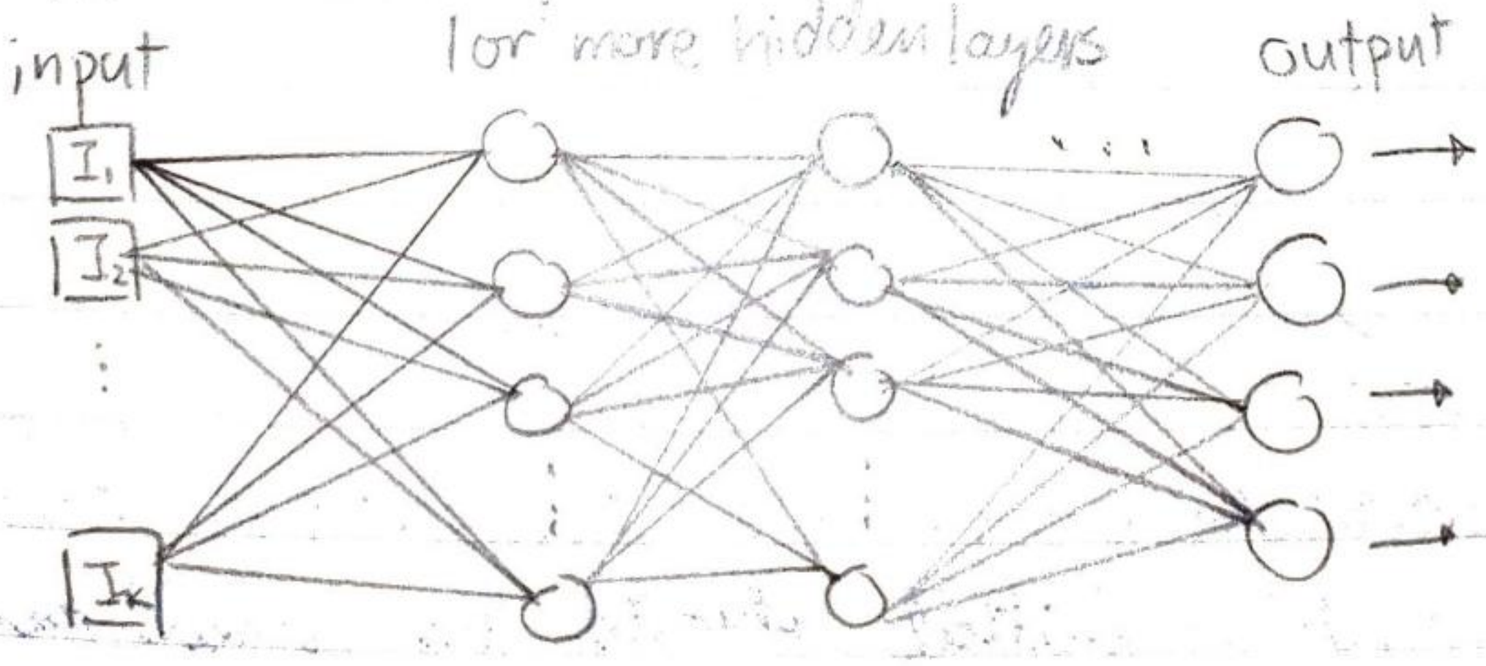
$\frac{\partial Err_b}{\partial O_b} = -2(T_b - O_b)$ from $(T_b - O_b)^2$ (can be dropped to be absorbed by α)

$\frac{\partial O_b}{\partial A(I)}$:

from logistic: $f(A(I)) \cdot (1 - f(A(I)))$

from hyperbolic tangent: $1 - \tanh^2(A(I))$

MULTI-LAYER NETWORKS



hidden - only see black-box with only a set of outputs and d' doesn't have access to network structure

each layer fully-connected to next one:
outputs of neurons in layer 1 act as input for neurons in layer 2 etc.

↳ each succeeding layer has access to more complex, more informative and likely more useful features

UPDATING WEIGHTS IN A HIDDEN LAYER

Weight updates from hidden layer to output layer identical to the single weight updates

Updates from input to output

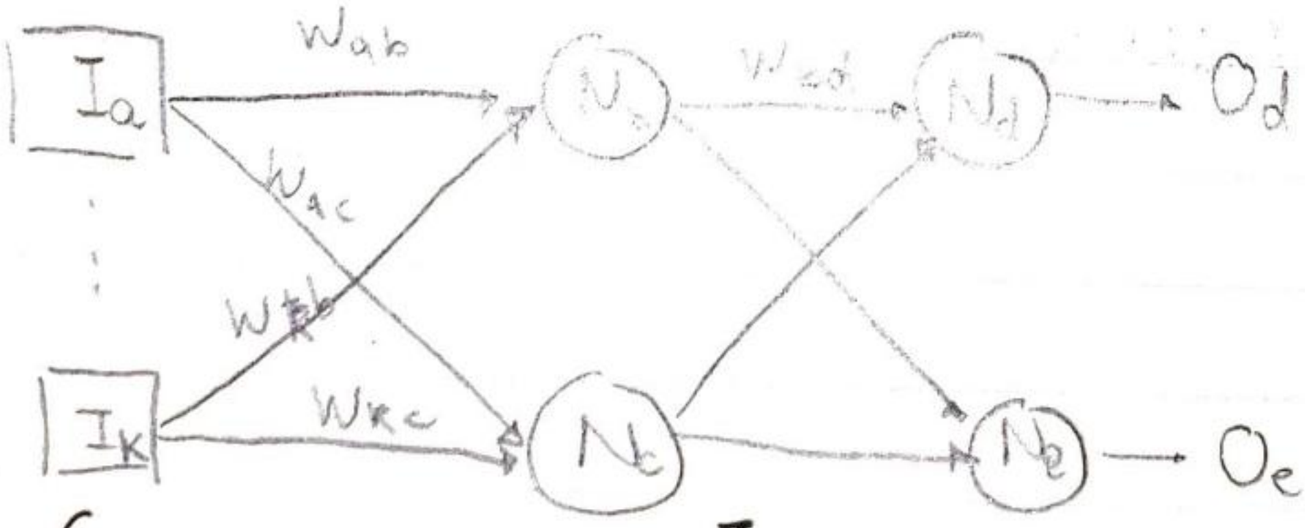
$$\frac{\partial Err_b}{\partial w_{ab}} = \frac{\partial A(I)}{\partial w_{ab}} \cdot \frac{\partial O_b}{\partial A(I)} \cdot \frac{\partial Err_b}{\partial O_b}$$

$\frac{\partial A(I)}{\partial w_{ab}}$: same

$\frac{\partial O_b}{\partial A(I)}$

$\frac{\partial Err_b}{\partial O_b}$: ?

$$\frac{\partial Err_b}{\partial O_b} = \sum_{j \in \text{neurons connected to B}} w_{bj} \frac{\partial f(x)}{\partial A_j(I)} (T_j - O_j)$$



TECHNICAL CONSIDERATIONS IN TRAINING A NEURAL NET

initializing weights: initial should be small random numbers (+ve, -ve), this will cause different results across training rounds, but small weights blow up: aka exploding gradient, typical in networks trained with vanilla stochastic Gradient Descent. When a weight grows too big, reset to

small random value / make learning rate smaller.
vanishing gradient: for sigmoid functions, small gradient towards +ve and -ve ends ⇒ neurons become saturated as weight updates close to zero.
target values might be better with 0.2, 0.8 -0.8, +0.8 to avoid ~~outputs~~ weights give values logistic where gradient tanh is flat.

MOVING TO DEEP LEARNING

For larger networks, standard SGD and squared error become insufficient.
Other aspects to improve:
- better methods to keep track of gradient and to update network weights
- different activation functions (e.g. ReLU)
- different error (loss) functions
- much larger input datasets
- more computing power
- framework to set up and train network

UNIT 6 - DECISION MAKING UNDER UNCERTAINTY - BAYES NETS

Some AI problems involve some amount of uncertainty

- Noise - affects values of state variables the agent uses to plan and take actions
- Imperfect Model - some variables not measurable, or not included as state variables
- Limitations in sensors
  or in ability of agent to observe certain state vars
- Incorrect assumptions about the world - assume static environment when environment might be changing

PROBABILISTIC INFERENCE

= deduce based on info

1. Given all available information about the environment, deduce the most likely state for the variables we are interested in
2. Decision making uses inferred states to take suitable actions

RANDOM VARIABLE

variable whose value depend on outcome of a random event

- have a domain, represented by probability distributions
  - PDF (probability distribution function) for continuous
  - PMF (probability mass function) for discrete
    → tells us likelihood of the random variable having any value of in the domain

RV X_1 has value v_1

JOINT DISTRIBUTION $p(X_1 = v_1, X_2 = v_2, \cdots, X_k = v_k)$

Joint probability distribution gives likelihood that the set of variables in our problem having a specific combination of values.

discrete variables ⇒ table with k dimensions, each entry stands for a possible combination of the values for the variables

e.g. RV T represents temperature = {'hot', 'cold'}
W represents weather = {'sunny', 'rainy'}

Joint probability table looks like:

p(T, W)	W = 'sunny'	W = 'rainy'
T = 'hot'	0.4	0.1
T = 'cold'	0.2	0.3

✡ each probability $p(X_1, X_2, \cdots, X_k) \geq 0$ and

$\sum_{x_1, x_2, \cdots, x_k} p(X_1, X_2, \cdots, X_k) = 1$ i.e Σprobabilities in the table = 1

✡ size of table grows exponentially!
k variables, each has domain size $d \Rightarrow d^k$ entries

EVENTS

possible outcomes (specific values for the variables)
usually interested in probability of event happening
e.g. P(T = hot, W = sunny), P(T = cold), P(W = rainy)
with a joint probability table compute probabilities of events by adding up probabilities of entries we are interested in (that match)

Marginalization - ignore the affect of a certain variable
e.g. To find P(T = hot), add P(T = hot, W = sunny) and P(T = hot, W = rainy)
which means we marginalize over temperature.

Conditional Probability

We are interested in the value of unobserved variables <u>given</u> the observed values for other variables

i.e. Conditional Probability $p(a|b) = \frac{p(a,b)}{p(b)}$

Probabilistic Inference

1. Observe values for subset of variables
2. compute conditional probabilities for the remaining variables using joint probability table
3. as more information becomes available, conditional probabilities are updated accordingly.

Product Rule

$$p(x_1, x_2, \cdots, x_k) = \prod_i p(x_i | x_1, x_2, \cdots, x_{i-1})$$
$$= p(x_1) \cdot p(x_2|x_1) \cdot p(x_3|x_2, x_1) \cdot \cdots p(x_k|x_{k-1}, \cdots x_1)$$

Conditional Independence

Take advantage of conditional independence relationships that will allow us to factor the joint probability into smaller / easier to manage conditional probability tables

e.g. S - smoke detected (T/F)
F - fire! (T/F)
A - alarm sounding (T/F)

$p(S,F,A) = p(S)\, p(F|S)\, p(A|F,S)$,

state of alarm depend on fire and smoke

However, if we observed presence of smoke, probability of alarm going off increases whether or not a fire is in the room.

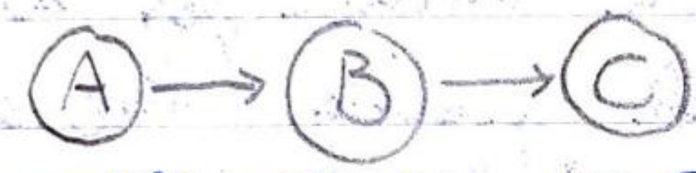
i.e. the value of A is conditionally independent of F, given S...

BAYES NETS

Graphical models used to represent dependencies between variables in a probabilistic inference problem.

GOAL: to identifying conditional and marginal independence relationships in order to simplify the inference process.

INDIRECT CAUSE



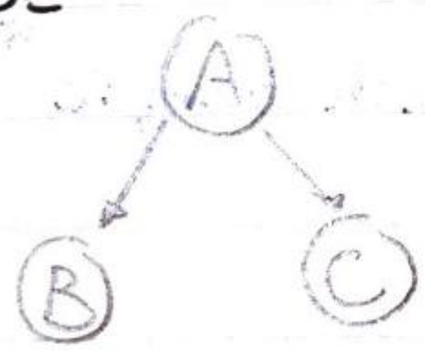
$p(A,B,C) = p(A)\,p(B|A)\,p(C|B)$

once B known, A no effect on C, but if B not known then C affects A via B

instead of $P(C|B,A)$

e.g. Fire → Smoke → alarm

COMMON CAUSE



once A known, B and C independent

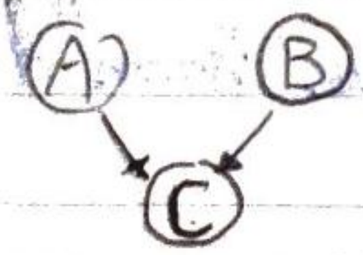
but if A not know, they affect eachother

$p(A,B,C) = p(A)\,p(B|A)\,p(C|A)$

covid

e.g. loss of taste, cough

COMMON EFFECT



once C known, A and B affect eachoth

but if C not known, they are independent

$p(A,B,C) = p(A)\,p(B)\,p(C|A,B)$

e.g. chickenpox, flu → fever

Inference In Bayesian Nets

Using the Bayes Net model we can use it to perform inference to answer:

- most likely value of variables $A, B, \ldots, X$ (for arbitrary subset)
- if some variables are observed with some values, how does that change the most likely value for the rest?
- for given observation of some variables, what is the optimal decision I can make?

Bayes Net Sampling

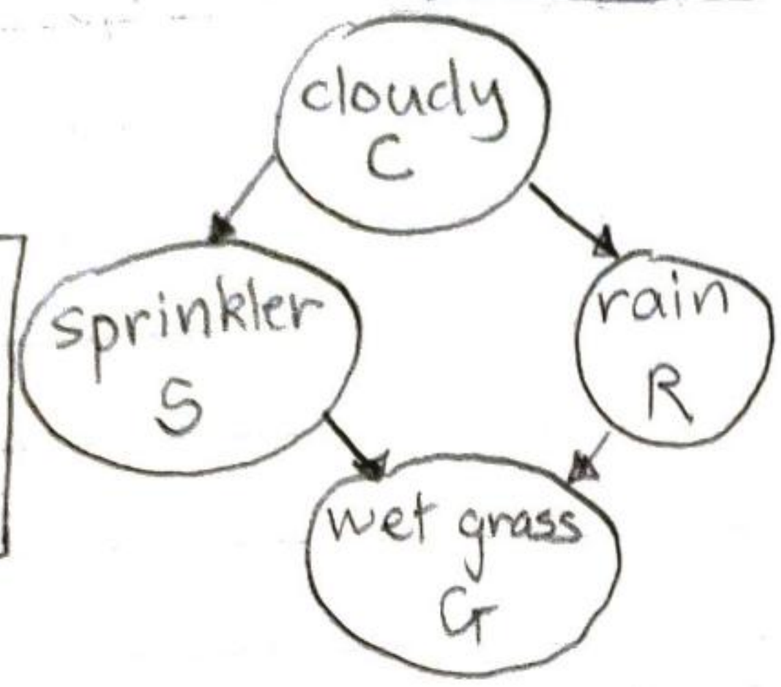
Random sample possible values for all unobserved variables in the network using the associated conditional probability tables at each node.

For N samples // large N
    randomly draw values for each of the unobserved variables in the network
    determine the weight of each sample*

For each variable
    create histogram of possible values (by counting: how many times each possible value was observed in the random sample, multiplied by the corresponding weight for each sample)
    normalize the histogram to obtain a probability distribution of values for that variable
    determine the most likely value from the histogram

e.g.

P(C)	
C=0	0.7
C=1	0.3



P(S|C)	C=0	C=1
S=0	0.8	0.5
S=1	0.2	0.5

P(R|C)	C=0	C=1
R=0	0.99	0.2
R=1	0.01	0.8

assume value for each variable $\in \{T, F\}$

P(W|S,R)	S=0, R=0	S=0, R=1	S=1, R=0	S=1, R=1
W=0	0.99	0.1	0.05	0.01
W=1	0.1	0.9	0.95	0.99

P(C) - does not depend on other variables
P(S|C) - depends whether cloudy or not
P(R|C) - depends whether cloudy or not
P(W|S,R) - depends on both rain and sprinklers

Random sampling:

1. start with variables that do not depend on anything

```
C: r = rand()
   if (r ≤ 0.7)
       C = 0
   else
       C = 1
```

e.g. r = 0.46271, C = 0

2. sample variables that depend on only one other variable

```
S: r = rand()
   if (r ≤ 0.8)   ← from P(S|C) when C=0
       S = 0
   else
       S = 1
```

e.g. r = 0.9321, S = 1
e.g. r = 0.1231, R = 0

(similar for R)

⋮

3. Sample variables that depend on step n ◆ 1 variable

W: r = rand() (look at column S=1, R=0)

if (r ≤ 0.05)

W = 0

else

W = 1

Complete Sample! C=0, R=0, S=1, W=1

* weight of sample is 1.0 because <u>all</u> variables were randomly sampled from the corresponding tables.

process repeats until there are N random samples for possible assignments to variables

With few hundred thousands ~ couple million samples, we can approximate very closely the actual probability for certain variables to have a certain value.

i.e. $\frac{\#\text{rows in samples that match Var = value}}{\#\text{ of rows}}$

RANDOM SAMPLING WITH OBSERVED VARIABLES

Importance sampling: ~~no~~ modify the process so that samples will use observed values for the variables instead of sampling.

PROBLEM: no longer fair samples ⇒ multiply by a weight to account for how likely event happen when randomly sampled.

* weight for a sample = 1.0 * P(this variable has this value)

e.g. if R=1 is observed, fix R=1 and random sample the rest:

C=0 S=0 R=1 W=1

Weight = 1.0 × P(R=1 | C=0) = 1.0 × 0.01 = 0.01

gives the weight of this row

e.g. to find P(C=0),

$$P(C=0) = \frac{\sum_{\text{rows with } C=0} w_{row}}{\sum_{\text{all rows}} w_{row}}$$

w_{row}: weight for that row