CS 486 Personal Notes

Marcus Chan

Taught by Pascal Poupart & Sriram Ganapathi Subramanian

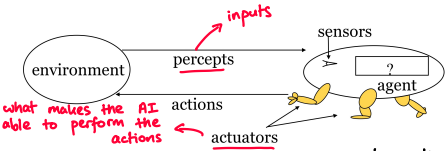
UW CS '25

Chapter 1: Introduction

REINFORCEMENT LEARNING PROBLEM

Our goal is for the AI to learn to choose actions that maximize rewards.

AGENTS & ENVIRONMENTS



The "agent function" maps percepts to actions; ie $f: P \to A$.

The "agent program" runs on the physical architecture to produce f.

RATIONAL AGENTS

A "rational agent" chooses whichever action that maximizes the expected value of its performance measure given the percept sequence to date.

Note that rationality is not omniscience, but rather learning & autonomy.

PEAS

"PEAS" helps us specify the task environment:

1. Performance measure;
   eg safety, destination, etc
2. Environment;
   eg streets, traffic, etc
3. Actuators; &
   eg steering, brakes, etc.
4. Sensors.
   eg GPS, engine sensors, etc.

PROPERTIES OF TASK ENVIRONMENTS

Task environments can be:

1. fully vs partially observable;
   - fully observable: agent knows state of the world from the stimuli
   - partially observable: agent does not directly observe the world's state
2. deterministic vs stochastic;
   - deterministic: next state is observable at any time
   - stochastic: next state is unpredictable
3. episodic vs sequential;
   - episodic: agent's current action will not affect a future action
   - sequential: agent's current action will affect a future action
4. static vs dynamic;
   - static: model is trained once
   - dynamic: model is trained continuously
5. discrete vs continuous; &
6. single agent vs multiagent.

* the former option is "easier" than the latter.

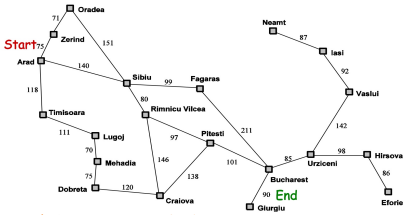
Chapter 2: Uninformed Search Techniques

SIMPLE PROBLEM SOLVING AGENT

```
function SIMPLE-PROBLEM-SOLVING-AGENT(percept) returns an action
  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
```

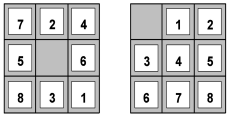
This can only tackle problems that are
1. fully observable;
2. deterministic;
3. sequential;
4. static;
5. discrete; &
6. single agent.

EXAMPLE: TRAVELLING IN ROMANIA



- initial state: In Arad
- actions: drive between cities
- goal test: In Bucharest
- path cost: distance between cities

EXAMPLE: 8-TILE PUZZLE



- states: locations of 8 tiles & blank
- initial state: any state
- actions: up, down, left, right
- goal test: does state match desired configuration
- path cost: # of steps

SEARCHING

We can visualize a state space search in terms of trees or graphs;
1. nodes correspond to states; &
2. edges correspond to taking actions.

These "search trees" are formed using "search nodes", which have
1. the state associated with it;
2. parent node & operator applied to the parent to reach the "current" node;
3. cost of the path so far; &
4. depth of the node.

EXPANDING NODES

"Expanding a node" refers to applying all legal operators to the state contained in the node & generating nodes for all corresponding successor states.

ie

GENERIC SEARCH ALGORITHM

Algorithm:
1. Initialize search with initial problem state.
2. Then repeat:
   - if no candidate nodes can be expanded, return failure
   - otherwise, choose a leaf node for expansion according to our search strategy.
   - if the node contains a goal state, return the solution.
   - otherwise, expand the node by applying the legal operators to the state associated within the node, & add the resulting nodes to the tree.

EVALUATING SEARCH ALGORITHMS

We can use the following properties when evaluating search algorithms:

1. "completeness" — is the algorithm guaranteed to find a solution (if it exists?)
2. "optimality" — does the algorithm find the optimal solution (ie lowest path cost)?
3. time & space complexity.

We consider the following variables:

1. "branching factor" (b) — the # of children each node has
2. depth of shallowest goal node (d); &
3. max length of any path in the state space (m).

BREADTH-FIRST SEARCH

Refer to CS341 notes for details; we expand all nodes on a given level before any node on the next level is expanded.

Evaluating the algorithm:

1. Completeness: yes if $b < \infty$
2. Optimality: yes if all costs are same
3. Time: $1 + b + \cdots + b^d \in O(b^d)$
4. Space: $O(b^d)$.

* all uninformed search methods have exponential time complexity.

UNIFORM COST SEARCH

Idea: we expand the node with the lowest path cost.

We can implement this using a priority queue.

Let C^* = cost of optimal solution & ε = min action cost. Then

1. Completeness: yes if $\varepsilon > 0$
2. Optimality: yes
3. Time: $O(b^{\lceil C^*/\varepsilon \rceil})$
4. Space: $O(b^{\lceil C^*/\varepsilon \rceil})$

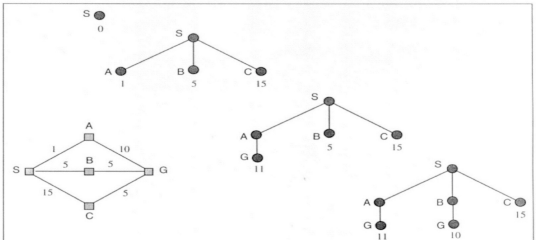


Figure 3.13 A route-finding problem. (a) The state space, showing the cost for each operator. (b) Progression of the search. Each node is labelled with $g(n)$. At the next step, the goal node with $g = 10$ will be selected.

DEPTH-FIRST SEARCH

Refer to CS341 notes for details; we expand the deepest node in the current fringe of the search tree first.

Evaluation:

1. Complete: no
   - may get stuck going down a long path
2. Optimal: no
   - might return a solution which is deeper (ie more costly) than another solution
3. Time: $O(b^m)$
   - note we might have $m > d$
4. Space: $O(bm)$

DEPTH-LIMITED SEARCH

Idea: Treat all nodes at depth ℓ as if they have no successors.
- try to choose ℓ based on the problem

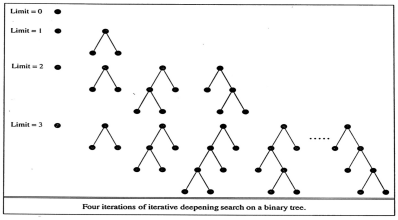
This avoids the problem of unbounded trees.

Evaluation:

1. Time: $O(b^\ell)$
2. Space: $O(b^\ell)$
3. Complete: no
4. Optimal: no

ITERATIVE-DEEPENING

Idea: repeatedly perform depth-limited search, but increase the limit each time.



Four iterations of iterative deepening search on a binary tree.

Evaluation:

1. Complete: yes
2. Optimal: yes
3. Time: $O(b^d)$
4. Space: $O(bd)$

Time:

(limit = 1) 1

(limit = 2) $1 + b$

(limit = 3) $1 + b + b^2$

$\vdots$

(+) (limit = d) $1 + b + b^2 + \cdots + b^d$

$d + (d-1)\,b + (d-2)\,b^2 + \cdots + b^d \in O(b^d)$

Chapter 3: Informed Search Techniques

MOTIVATION

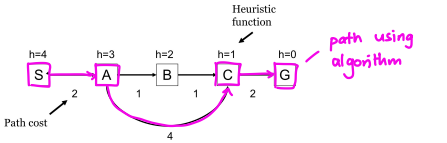
1. In search problems, we often have additional knowledge about the problem.
   - eg with the "travelling around Romania", we know dist. bw cities
     - ↳ so we can find the overhead in going the wrong direction
2. Our knowledge is often about the "merit" of nodes.
3. Notions of merit:
   1. how expensive it is to get from a state to a goal;
   2. how easy it is to get from a state to a goal.

HEURISTIC FUNCTIONS · h(n)

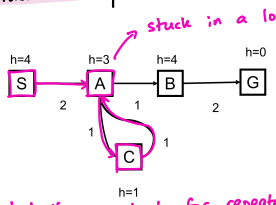
1. We need to develop domain specific "heuristic functions" h(n), which guess the cost of reaching the goal from node n.
2. In general, if $h(n_1) < h(n_2)$, we guess reaching the goal is cheaper from n_1 than from n_2.
3. We also need
   1. $h(n) = 0$ if n is a goal node
   2. $h(n) > 0$ if n is not a goal node.

GREEDY BEST-FIRST SEARCH

1. Idea: Use h(n) to rank the nodes in the fringe & expand the node with the lowest h-value.
   (ie "greedily" trying to find the least-cost solution).
2. Example:



3. Note greedy best-first is not optimal.
   - eg in the above example,
     - path found has cost = 6.
     - but cheaper path is S→A→B→C→G with cost = 6.
4. It is also not complete, as it can be stuck in loops.
   - eg



   - but if we check for repeated states then we are okay
5. This algorithm uses exponential space & worst-case time.

A* SEARCH

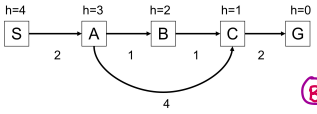
Idea: Define $f(n) = g(n) + h(n)$, where

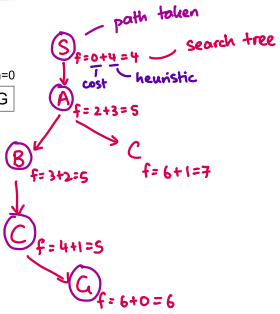
① $g(n)$ = cost of path to node n

② $h(n)$ = heuristic estimate of cost of reaching goal from node n.

We then iteratively expand the node in the fringe with the lowest f value.

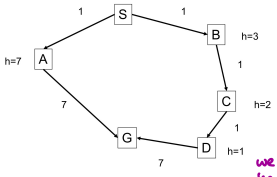
Example:

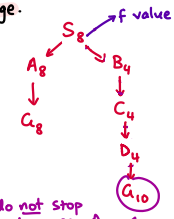




A* should terminate only when the goal state is popped from the queue / fringe.

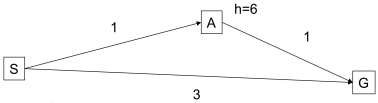
eg





we do not stop here because A_8 is still in the queue.

Note A* is not optimal.



- A* finds S→G with cost 3
- but there is a cheaper path S→A→G with cost 2.
- A* performed poorly because our heuristic was not good.

ADMISSIBLE [HEURISTICS]

Let $h^*(n)$ be the true minimal cost to the goal from node n.

Then, we say the heuristic $h(n)$ is "admissible" if

$$h(n) \leq h^*(n) \quad \forall n.$$

In particular, admissible heuristics never overestimate the cost to the goal.

h(n) IS ADMISSIBLE ⇒ A* IS OPTIMAL

If $h(n)$ is admissible, then A* with tree-search is optimal.

Proof. Let G be an optimal goal state, & $f(G) = f^* = g(G)$.

Let G_2 be a suboptimal goal state, ie $f(G_2) = g(G_2) > f^*$.

(since $h(G_1) = h(G_2) = 0$)

Assume, for a contⁿ, that A* selects G_2 from the queue; ie A* terminates with a suboptimal solution.

Let n = node currently a leaf node on an optimal path to G.

G • • G_2

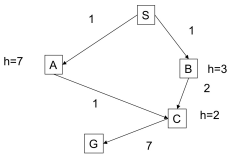
Since h is admissible, $f^* \geq f(n)$.

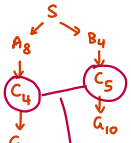
If n is not chosen for expansion over G_2, then $f(n) \geq f(G_2)$.

Thus $f^* \geq f(G_2)$. Since $h(G_2) = 0$, thus $f^* \geq g(G_2)$, a contⁿ.

REVISITING STATES IN A*

Motivation:





* the left path is cheaper than the right!

If we allow states to be expanded again, we might get a better solution!

CONSISTENT [HEURISTICS]

We say $h(n)$ is "consistent" if

$$h(n) \leq \text{cost}(n, n') + h(n') \quad \forall n, n'.$$

Note that A* graph-search with a consistent heuristic is optimal.

PROPERTIES OF A*

Note that A* is

① Complete if $h(n)$ is consistent;
- f always increases along any path

② Has exponential time complexity in the worst-case; &
- but a good heuristic helps a lot
- $O(bm)$ if heuristic is perfect

③ Has exponential space complexity.

ITERATIVE DEEPENING A* (IDA*)

1. Idea: Like iterative deepening search, but change f-cost rather than depth in each iteration.
2. This reduces the space complexity.

SIMPLIFIED MEMORY-BOUNDED A* (SMA*)

1. Idea: Proceeds like A* but when it runs out of memory it drops the worst leaf node (ie one with highest f-value).
2. If all leaf nodes have the same f-value, then drop the oldest & expand the newest.
3. This is
   ① optimal; &
   ② complete if depth of shallowest goal node < memory size.

OBTAINING HEURISTICS

1. One approach to get heuristics is to think of an easier problem & let h(n) be the cost of reaching the goal in the easier problem.
2. We can also
   ① precompute solution costs of subproblems & store them in a pattern database; or
   ② learn from experience with the problem class.

EXAMPLE: 8-PUZZLE GAME

1. We can relax the game in 3 ways:
   ① We can move tile from position A→B if A is next to B (ignore whether position is blank)
   ② We can move tile from position A→B if B is blank (ignore adjacency)
   ③ We can move tile from position A→B regardless.
2. ③ leads to the "misplaced tile heuristic". (h_2)
   - to solve this problem we need to move each tile into its final position.
   - # of moves = # of misplaced tiles
   - admissible
3. ① leads to the "manhattan distance heuristic". (h_1)
   - to solve this we need to slide each tile into its final position
   - admissible
4. Note h_1 "dominates" h_2; ie $h_2(n) \leq h_1(n)$ $\forall n$.

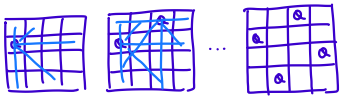
Chapter 4: Constraint Satisfaction

INTRODUCTION

1. These are useful for problems where the state structure is important.
2. In many problems, the same state can be reached independent of the order in which the moves are chosen.
3. So, we can try to solve problems efficiently by being smart about the action order.

4-QUEENS CONSTRAINT PROPAGATION

Idea: Remove conflicting squares from consideration when we put a queen down.



CONSTRAINT SATISFACTION PROBLEM (CSP)

A "constraint satisfaction problem" is defined by some $\{V, D, C\}$, where

1. $V = \{V_1, \dots, V_n\}$ is a set of variables;
2. $D = \{D_1, \dots, D_n\}$ is a set of domains, where D_i is the set of possible values for each V_i; &
3. $C = \{C_1, \dots, C_m\}$ is the set of constraints.

STATE

A "state" is an assignment of values to some or all of the variables;

ie $V_i = x_i$, $V_j = x_j$, etc.

CONSISTENT [ASSIGNMENT]

We say an assignment is "consistent" if it does not violate any constraints.

SOLUTION

A "solution" is a complete, consistent assignment.

EXAMPLE: 8 QUEENS AS A CSP

8 queens as a CSP:

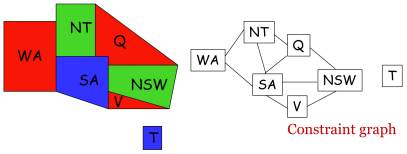
- variables: V_{ij}, $i,j = 1, \dots, 8$
- domain of each var: $\{0,1\}$
- constraints: $V_{ij} = 1 \Rightarrow V_{ik} = 0 \quad \forall k \neq j$

  $V_{ij} = 1 \Rightarrow V_{kj} = 0 \quad \forall k \neq i$

  similar constraint for diagonals

  $\sum_{i,j} V_{ij} = 8$

EXAMPLE: MAP COLORING



Constraint graph

- variables: WA, NT, ..., T (the regions)
- each var has the same domain: {red, green, blue}
- no 2 adjacent variables have the same value

  (ie $WA \neq NT$, $WA \neq SA$, etc)

PROPERTIES OF CSPs

Types of variables:

① Discrete & finite;
eg 8-queens, map coloring
* we focus on this in this course.

② Discrete with infinite domains; &
eg job scheduling

③ Continuous domains.

Types of constraints:

① "Unary constraint": relates a single variable to a value
- eg Queensland = blue

② "Binary constraint": relates two variables

③ "Higher order constraints": relates ≥3 variables.

CSPs & SEARCH

We can formulate CSPs as a search problem:

① we have N variables $V_1, \dots, V_n$;

② a valid assignment is $\{V_1 = x_1, \dots, V_k = x_k\}$, $0 \le k \le n$ where values satisfy the variable constraints.

③ states: valid assignments

④ initial state: empty assignment

⑤ successor: $\{V_1 = x_1, \dots, V_k = x_k\} \rightarrow \{V_1 = x_1, \dots, V_k = x_k, V_{k+1} = x_{k+1}\}$

⑥ goal test: complete assignment

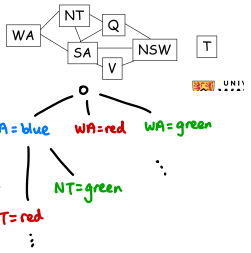
BACKTRACKING SEARCH

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

function RECURSIVE-BACKTRACKING(*assignment*, *csp*) returns a solution, or failure
  if *assignment* is complete then return *assignment*
  var ← SELECT-UNASSIGNED-VARIABLE(*Variables*[*csp*], *assignment*, *csp*)
  for each *value* in ORDER-DOMAIN-VALUES(*var*, *assignment*, *csp*) do
    if *value* is consistent with *assignment* according to Constraints[*csp*] then
      add { *var* = *value* } to *assignment*
      result ← RECURSIVE-BACKTRACKING(*assignment*, *csp*)
      if *result* ≠ *failue* then return *result*
      remove { *var* = *value* } from *assignment*
  return *failure*

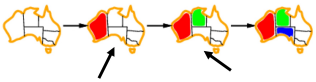
- this is DFS that choose values for one variable at a time
- we "backtrack" when a variable has no legal values to assign

EXAMPLE: MAP COLORING



MOST CONSTRAINED VARIABLE HEURISTIC

Idea: Choose the variable which has the fewest "legal" moves.



$D_{NT} = \{green, blue\}$ $D_{SA} = \{blue\}$
$D_{SA} = \{green, blue\}$ $D_{Q} = \{blue, red\}$
$D_{others} = \{red, green, blue\}$ $D_{others} = \{red, green, blue\}$

In a tie, choose the variable with the most constraints on the remaining variables.
(ie "most constraining variable").

LEAST CONSTRAINING VALUE HEURISTIC

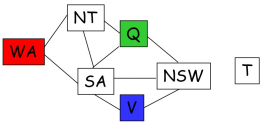
Idea: Given a variable, choose the "least constraining value", ie the one that rules out the fewest values in the remaining variables.

FORWARD CHECKING

Idea_1: We keep track of remaining legal values for unassigned variables, & terminate search when any variable has no legal values.

This_2 helps us detect failure early.

EXAMPLE: MAP COLORING



	WA	NT	Q	NSW	V	SA	T
	RGB	RGB	RGB	RGB	RGB	RGB	RGB
WA = R	R	GB	RGB	RGB	RGB	GB	RGB
Q = G	R	B	G	RB	RGB	B	RGB
V = B	R	B	G	~~RB~~	B	~~B~~	RGB

this is the empty set;
⇒ the current assignment does not lead to a solution.

Chapter 5: Uncertainty

1. Refer to STAT 231/330 for more details.
2. We use "~" to denote the complement of an event (ie ~A).

BAYES RULE

1. For 2 events A, B, note

$$P(B|A) = \frac{P(A|B)\,P(B)}{P(A)}$$

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

$\therefore P(B|A) = \frac{P(B)P(A|B)}{P(A)}$. ∎

2. In particular, it allows us to compute a belief about hypothesis B given evidence A.
3. More general forms:

$$P(A|B) = \frac{P(B|A)\,P(A)}{P(B|A)P(A) + P(B|\sim A)P(\sim A)}$$

$$P(A|B \wedge X) = \frac{P(B|A \wedge X)\,P(A|X)}{P(B|X)}$$

$$P(A = v_i|B) = \frac{P(B|A=v_i)\,P(A=v_i)}{\sum_{k=1}^{n} P(B|A=v_k)\,P(A=v_k)}$$

PROBABILISTIC INFERENCE

1. Idea: Given a prior distribution P(X) over variables X of interest & given new evidence E=e for some variable E, revise our degrees of belief: ie the "posterior" P(X|E=e).

ISSUES

1. Specifying the full joint distribution for $X_1, \ldots, X_n$ requires an exponential number of possible "worlds".
2. So, inference is also slow since we need to sum over these exponential number of worlds.

$$P(X_1, \ldots, X_{i-1}, X_{i+1}, \ldots, X_n | X_i) = \frac{P(X_1, \ldots, X_n)}{\sum_{x_1} \cdots \sum_{x_{i-1}} \sum_{x_{i+1}} \cdots \sum_{x_n} P(X_1, \ldots, X_n)}$$

CONDITIONAL INDEPENDENCE

1. Two variables X, Y are "conditionally independent" given Z if

$$P(X=x \mid Z=z) = P(X=x \mid Y=y, Z=z)$$
$$\Leftrightarrow P(X=x, Y=y \mid Z=z) = P(X=x|Z=z)\,P(Y=y|Z=z)$$
$$\Leftrightarrow \forall x \in \text{dom}(X),\ y \in \text{dom}(Y),\ z \in \text{dom}(Z)$$

2. If we know the value of Z, nothing we learn about Y will influence our beliefs about X.

VALUE OF INDEPENDENCE

1. If $X_1, \ldots, X_n$ are mutually independent, then we can specify the full joint distribution using only n parameters (ie linear) instead of $2^n - 1$ (ie exponential).
2. Although most domains do not exhibit complete mutual independence, they do instead exhibit a fair amount of conditional independence.

NOTATION: P(X)

1. We define "P(X)" as the marginal distribution over X.
   - P(X=x) is a number, P(X) is a distribution.

NOTATION: P(X|Y)

1. We define "P(X|Y)" as the family of conditional distributions over X; one for each $y \in \text{dom}(Y)$.

EXPLOITING CONDITIONAL INDEPENDENCE : CHAIN RULE

Consider a story:

- If Pascal woke up too early E, Pascal probably needs coffee C; if Pascal needs coffee, he's likely grumpy G. If he is grumpy then it's possible that the lecture won't go smoothly L. If the lecture does not go smoothly then the students will likely be sad S.

E - Pascal woke up too early G - Pascal is grumpy S - Students are sad
C - Pascal needs coffee L- The lecture did not go smoothly

S is independent of E, C, G given L
L is independent of E, C given G & so on.

$\Rightarrow P(S \mid L, G, C, E) = P(S \mid L)$
$P(L \mid G, C, E) = P(L \mid G)$
$P(G \mid C, E) = P(G \mid C)$

Then

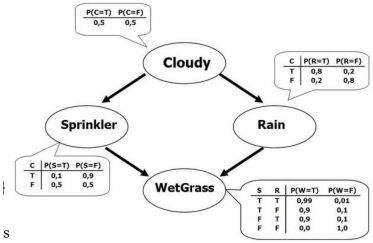
$$P(S, L, G, C, E) = P(S \mid L, G, C, E)\, P(L \mid G, C, E)\, P(G \mid C, E) \cdot P(C \mid E)\, P(E)$$
$$= P(S \mid L)\, P(L \mid G)\, P(G \mid C)\, P(C \mid E)\, P(E).$$

In this, we can specify the full joint distribution by specifying the five local conditional distributions.

Chapter 6: Bayesian Networks

BAYESIAN / BELIEF / PROBABILISTIC NETWORKS (BN)

"Bayesian networks" are graphical representations of the direct dependencies over a set of variables, alongside a set of conditional probability tables (CPT) quantifying the strength of the influences.



In particular, it has
1. A DAG with nodes = variables X_i, &
2. A set of CPTs $P(X_i \mid Parents(X_i))$ for each X_i.

Key notions:
1. parents / children of a node;
2. ancestors / descendants of a node;
3. family: set of nodes consisting of X_i & its parents.

SEMANTICS OF A BAYES NET

Idea: Every X_i is conditionally independent of all its non-descendants given its parents; ie

$$P(X_i \mid S \cup Par(X_i)) = Pr(X_i \mid Par(X_i)) \quad \forall S \subseteq NonDescendants(X_i)$$

Also, the joint distribution is recoverable using the parameters (CPTs) in the BN:

$$P(x_1, \dots, x_n) = P(x_n \mid x_{n-1}, \dots, x_1)\, P(x_{n-1} \mid x_{n-2}, \dots, x_1) \cdots P(x_1)$$
$$= P(x_n \mid Par(x_n))\, P(x_{n-1} \mid Par(x_{n-1})) \cdots P(x_1),$$

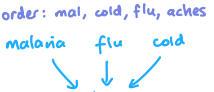
CONSTRUCTING A BN

Idea:
1. Take any ordering of the variables, and then for X_n to X_1:
   - let $Par(X_n)$ be any subset $S \subseteq \{X_n, \dots, X_1\}$ such that X_n is independent of $\{X_1, \dots, X_{n-1}\} - S$ given S.
   - Continue this for $X_{n-1}, \dots, X_1$.
2. In the end, we get a DAG, which is also a BN by construction.

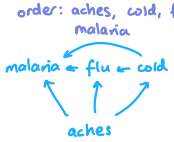
Note the order in which we consider the variables changes the resultant BN!

eg

order: mal, cold, flu, aches



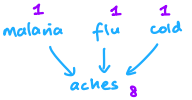
order: aches, cold, flu, malaria



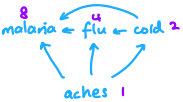
COMPACTNESS

In a BN, if each rv is directly influenced by at most k others, then each CPT will have at most 2^k entries.

So, the entire network of n variables is specified by $n \cdot 2^k$ parameters.



$1+1+1+8 = 11$



$1+2+4+8 = 15$

d-SEPARATION

First, we say a set of variables E "d-separates" X & Y if it "blocks" every undirected path in the BN between X & Y.

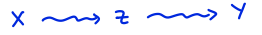
TESTING INDEPENDENCE

Then, X & Y are conditionally independent given evidence E if E d-separates X & Y.

BLOCKING IN d-SEPARATION

Let P be an undirected path from X→Y. Then the evidence set E "blocks" path P if

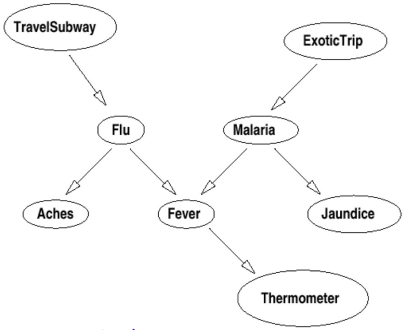
① one arc on P goes into Z & one goes out of Z, & Z∈E;



② both arcs on P leave Z & Z∈E; or

③ both arcs on P enter Z & neither Z nor any of its descendants are in E.

EXAMPLE



① subway & thermometer
- dependent
- but independent given the flu
  ↳ since flu blocks the only path (rule 1)

② aches & fever
- dependent
- but independent given the flu
  ↳ since flu blocks the only path (rule 2)

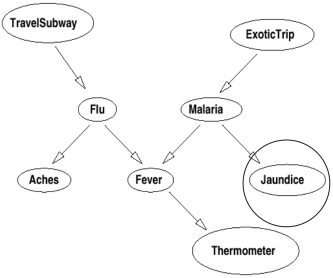
③ flu & malaria
- independent
- dependant given fever or thermometer
  ↳ rule 3

④ subway & exotic trip
- independent
- dependent given thermometer
  ↳ rule 3

SIMPLE FORWARD INFERENCE (CHAIN)

Idea: To compute the marginal distribution, we can use simple forward "propagation" of probabilities.

eg



$$P(J) = \sum_{M,ET} P(J,M,ET) \quad \text{(marginalization)}$$

$$= \sum_{M,ET} P(J|M,ET)\,P(M|ET)\,P(ET)$$

(chain rule)

$$= \sum_{M,ET} P(J|M)\,P(M|ET)\,P(ET)$$

(conditional independence)

$$= \underbrace{\sum_{M} P(J|M) \sum_{ET} P(M|ET)\,P(ET)}$$

all these terms can now be found in the CPTs.

We can do something similar if we have "upstream" evidence.

eg $P(J|ET) = \sum_{M} P(J,M|ET)$

$$= \sum_{M} P(J|M,ET)\,P(M|ET)$$

$$= \underbrace{\sum_{M} P(J|M)\,P(M|ET)}$$

terms found in CPTs

SIMPLE BACKWARD INFERENCE

Idea: For "downstream" evidence, we must reason backwards, which we can use Bayes' rule:

eg (using same BN as above)

$$P(ET|J) = \alpha P(J|ET)\,P(ET), \quad \alpha = \frac{1}{P(J)}$$

$$= \alpha \sum_{M} P(J,M|ET)\,P(ET)$$

$$= \alpha \sum_{M} P(J|M,ET)\,P(M|ET)\,P(ET)$$

$$= \alpha \sum_{M} P(J|M)\,P(M|ET)\,P(ET).$$

We can then calculate $P(J) = \sum_{ET} P(J|ET)\,P(ET)$.

VARIABLE ELIMINATION

The "variable elimination" algorithm is a general inference tool for BNs.

FACTORS

A "factor" is a function $f(X_1, \ldots, X_k)$.

We can represent factors as a table of numbers, one for each instantiation of the variables $X_1, \ldots, X_k$.

We denote $f(X, Y)$ to be a factor over the variables $X \cup Y$, where X & Y are sets of variables.

Note each CPT in a Bayes net is a factor of its family.

eg $P(C \mid A, B)$ → factor of A, B, C.

PRODUCT OF FACTORS: fg

Let $f(X, Y)$, $g(Y, Z)$ be factors with variables Y in common.

Then the "product" of f & g, $h = fg$, is defined to be

$$h(X, Y, Z) = f(X, Y) \times g(Y, Z)$$

eg

f(A,B)		g(B,C)		h(A,B,C)			
ab	0.9	bc	0.7	abc	0.63	ab~c	0.27
a~b	0.1	b~c	0.3	a~bc	0.02	a~b~c	0.08
~ab	0.4	~bc	0.2	~abc	0.28	~ab~c	0.12
~a~b	0.6	~b~c	0.8	~a~bc	0.12	~a~b~c	0.48

SUM VARIABLE OUT OF A FACTOR: $\Sigma_X f$

Let $f(X, Y)$ be a factor, where X is a variable & Y is a variable set.

Then, we can "sum out" variable X from f to produce a new factor $h = \sum_X f$, where

$$h(Y) = \sum_{x \in Dom(X)} f(x, Y).$$

eg

f(A,B)		h(B)	
ab	0.9	b	1.3
a~b	0.1	~b	0.7
~ab	0.4		
~a~b	0.6		

RESTRICTING FACTORS: $f_{X=x}$

Let $f(X, Y)$ be a factor with variable X & variable set Y.

Then, we "restrict" factor f to $X = x$, ie $h = f_{X=x}$, by doing

$$h(Y) = f(x, Y).$$

eg

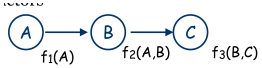
f(A,B)		h(B) = $f_{A=a}$	
ab	0.9	b	0.9
a~b	0.1	~b	0.1
~ab	0.4		
~a~b	0.6		

NO EVIDENCE CASE

Idea: Computing prior probability of the query variable X can be seen as applying these operations on factors.

EXAMPLE 1

eg



$$P(C) = \sum_{A,B} P(C \mid B)\, P(B \mid A)\, P(A)$$

$$= \sum_B P(C \mid B) \sum_A P(B \mid A)\, P(A)$$

$$= \sum_B f_3(B, C) \sum_A f_2(A, B)\, f_1(A)$$

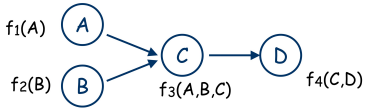
$$= \sum_B f_3(B, C)\, f_4(B)$$

$$= f_5(C).$$

Numerical example:

$f_1(A)$		$f_2(A,B)$		$f_3(B,C)$		$f_4(B)$		$f_5(C)$	
a	0.9	ab	0.9	bc	0.7	b	0.85	c	0.625
~a	0.1	a~b	0.1	b~c	0.3	~b	0.15	~c	0.375
		~ab	0.4	~bc	0.2				
		~a~b	0.6	~b~c	0.8				

EXAMPLE 2



eg $P(D) = \sum_{A,B,C} P(D|C)\,P(C|B,A)\,P(B)\,P(A)$

$= \sum_{C} P(D|C) \sum_{B} P(B) \sum_{A} P(C|B,A)\,P(A)$

$= \sum_{C} f_4(C,D) \sum_{B} f_2(B) \sum_{A} f_3(A,B,C)\,f_1(A)$

$= \sum_{C} f_4(C,D) \sum_{B} f_2(B)\,f_5(B,C)$

$= \sum_{C} f_4(C,D)\,f_6(C)$

$= f_7(D)$

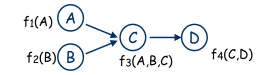
* define f_5, f_6, f_7 according to the brackets

ALGORITHM (NO EVIDENCE)

Input: query var Q, remaining vars Z, & F = set of factors corresponding to CPTs for {Q} ∪ Z.

1. Choose an elimination ordering $Z_1, \ldots, Z_n$ of variables in **Z**.
2. For each Z_j -- in the order given -- eliminate $Z_j \in$ **Z** as follows:
   (a) Compute new factor $g_j = \Sigma_{Zj}\, f_1 \times f_2 \times \ldots \times f_k$, where the f_i are the factors in F that include Z_j
   (b) Remove the factors f_i (that mention Z_j) from F and add new factor g_j to F
3. The remaining factors refer only to the query variable Q. Take their product and normalize to produce P(Q)

eg



query: P(D)

elim. order: A, B, C

Steps:

① add $f_5(B,C) = \sum_{A} f_3(A,B,C)\,f_1(A)$;

remove $f_1(A)$, $f_3(A,B,C)$

② add $f_6(C) = \sum_{B} f_2(B)\,f_5(B,C)$

- we don't need to sumout f_3 as we removed it in step ①

remove $f_2(B)$, $f_5(B,C)$

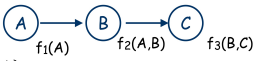
③ add $f_7(C) = \sum_{C} f_4(C,D)\,f_6(C)$

remove $f_4(C,D)$, $f_6(C)$

④ The remaining factor $f_7(D)$ is our (possibly unnormalized) probability P(D)

EVIDENCE CASE

eg



$P(A|C=c) = \alpha P(A)\,P(C=c|A)$ (Bayes' thm)

$= \alpha P(A) \sum_{B} P(C=c|B)\,P(B|A)$

$= \alpha f_1(A) \sum_{B} f_3(B,c)\,f_2(A,B)$

$= \alpha f_1(A) \sum_{B} f_4(B)\,f_2(A,B)$

$= \alpha f_1(A)\,f_5(A)$

$= f_6(A)$

ALGORITHM (WITH EVIDENCE)

Input: Given query var Q, evidence vars E (observed to be e), remaining vars Z & set of factors involving CPTs for {Q} ∪ Z, F:

1. Replace each factor $f \in F$ that mentions a variable(s) in E with its restriction $f_{E=e}$ (somewhat abusing notation)
2. Choose an elimination ordering $Z_1, \ldots, Z_n$ of variables in **Z**.
3. For each Z_j -- in the order given -- eliminate $Z_j \in$ **Z** as follows:
   (a) Compute new factor $g_j = \Sigma_{Zj}\, f_1 \times f_2 \times \ldots \times f_k$, where the f_i are the factors in F that include Z_j
   (b) Remove the factors f_i (that mention Z_j) from F and add new factor g_j to F
4. The remaining factors refer only to the query variable Q. Take their product and normalize to produce P(Q)

eg same example

Query: P(A | D=d)

① replace $f_4(C,D)$ with $f_5(C) = f_4(C,d)$

② proceed similar to before

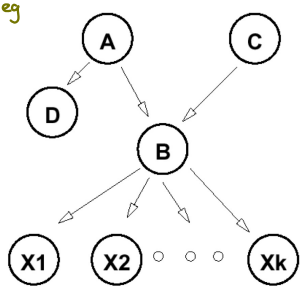
ANALYSIS

1. After eliminating Z_j, the factors remaining in set F refer only to $X_{j+1}, \ldots, Z_n$ & Q.
2. Also, no factor mentions any evidence variable E after the initial restriction.
3. Note
   ① The number of iterations is linear in the # of variables; &
   ② The complexity is exponential in the # of variables.

POLYTREES

1. Polytrees are basically "trees" (ie no undirected cycles) that can have multiple start nodes.
2. Idea: In these, the inference is linear wrt the size of the network.
3. To do this, we eliminate only "singly-connected nodes".

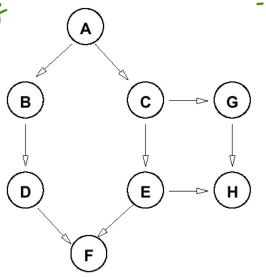
eg



- eliminate $D, A, C, X_1, \ldots, X_k$
- if we eliminate B before these, we get factors that include all of $A, C, X_1, \ldots, X_k$!

LEAST NEIGHBORS HEURISTIC

eg



- A, F, H, C, B, C, E is good
  ↳ we eliminate the nodes with 2 neighbors first
  ↳ leaving the nodes with 3 neighbors at the end.
- if we started with B, the ordering would be bad since the size of the factors is larger.

Idea: When choosing an ordering, prioritize nodes with the least number of neighbors.

RELEVANCE

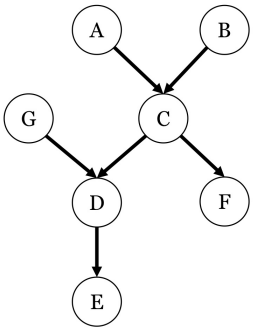
1. Motivation: Certain variables have no impact on the query.

   eg $A \rightarrow B \rightarrow C$

   - To calculate P(A), we only need to look at A's CPT!
   - However, if we do var elimination, we get trivial factors (ie whose value is just 1)

2. Thus, when considering variables, we can restrict our attention to only the "relevant" ones;
   ie given query Q & evidence E:
   ① Q is relevant;
   ② If Z is relevant, Parents(Z) are relevant; &
   ③ If $E' \in E$ is a descendent of a relevant node, then E' is relevant.

eg



① Q = P(F)
relevant: F, C, B, A

② Q = P(F|E)
relevant: F, C, B, A, E, D, G

③ Q = P(F|E,C)
relevant: whole graph, but really none except C, F since C cuts off all influence of others.

Chapter 7: Causal Inference

CAUSALITY

"Causality" is the study of how things influence each other & how causes lead to effects.

CAUSAL DEPENDENCE

We say "X causes Y" if changes to X induce changes in Y.

Note joint distributions captures correlations between X & Y, not causations.

- $P(Y|X) \not\Rightarrow$ X causes Y

CAUSAL BAYESIAN NETWORK

A "causal Bayesian network" is one where all edges indicate direct causal effects.

eg malaria, flu, cold → aches

CAUSAL INFERENCE

Causal networks can solve "intervention queries", ie what the effect of an action is.

- but non-causal networks cannot.

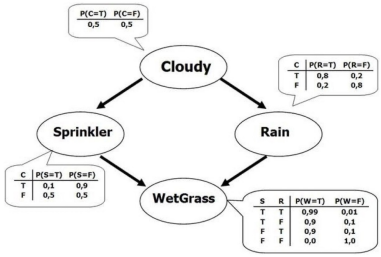
OBSERVATION VS INTERVENTION

"Observational queries" are in the form "What is the likelihood of Y given X?" ie $P(Y|X=x)$.

"Interventional queries" are in the form "How does doing X affect Y?" ie $P(Y|do(X=x))$.

- the "do" keyword specifies the query is an intervention.

EXAMPLE: CAUSAL GRAPH



observational: $P(WG|S=true)$

- factors: $P(C)$, $P(R|C)$, $P(S|C)$, $P(WG|S,R)$
- evidence: $S=true$
- eliminate: C, R

interventional: $P(WG|do(S=true))$

- we can remove the CPT from S since we "explicitly set" $S=true$.
- factors: $P(C)$, $P(R|C)$, $P(WG|S,R)$
- evidence: $S=true$
- eliminate: C, R

INFERENCE WITH THE DO OPERATOR

To do inference for $P(X|do(Y=y), Z=z)$:

1. Remove edges pointing to Y & $P(Y|Parents(Y))$
2. Perform variable elimination as usual (evidence is $Y=y, Z=z$).

COUNTER-FACTUAL ANALYSIS

"Counter-factual analysis" explores outcomes that did not occur, but could have occurred under different conditions.
- basically a "what-if?" analysis

This can help test causal relationships.
eg "would the patient have died if he was not treated"

STRUCTURAL CAUSAL MODEL / SCM

Idea: We want to separate causal relations from "noise".

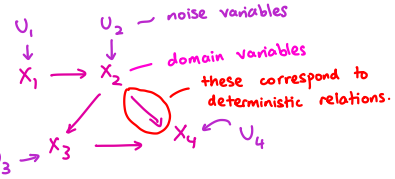
A "structural causal model" consists of

① X: endogenous / domain variables
② U: exogenous variables / noise
③ Only deterministic relations given by equations in the form

$$X_i = f(\text{parents}(X_i), U_i)$$

where U_i corresponds to the noise variable associated with X_i.

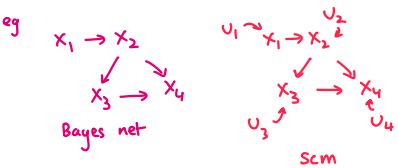
eg



$X_1 = f_1(U_1)$, $X_2 = f_2(X_1, U_2)$,
$X_3 = f_3(X_2, U_3)$, $X_4 = f_4(X_3, X_2, U_4)$

We can convert SCMs to causal Bayesian networks, but not v.v.

eg



Then

$$P(X_1) = \sum_{U_1} P(U_1)\, \underbrace{f(U_1)}_{X_1}$$

$$P(X_2 | X_1) = \sum_{U_2} P(U_2)\, \underbrace{f(X_1, U_2)}_{X_2}$$

...

SCMs are more descriptive since they separate causal relations from noise.

METHOD

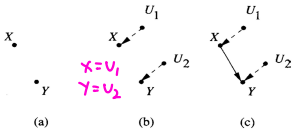
For a causal model M, to find $P(Y=y \mid e, do(X=x))$:

① Update $P(u)$ to find $P(u|e)$ (abduction);
- u = noise variables

② Replace the equations corresponding to variables in set X by the equations $X = x$ (action); &

③ Use the modified model to calculate $P(Y=y)$.

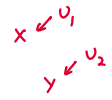
EXAMPLE



Model 1	$u_2=0$		$u_2=1$		Marginal	
	$x=1$	$x=0$	$x=1$	$x=0$	$x=1$	$x=0$
$y=1$ (death)	0	0	0.25	0.25	0.25	0.25
$y=0$ (recovery)	0.25	0.25	0	0	0.25	0.25

Model 2	$u_2=0$		$u_2=1$		Marginal	
	$x=1$	$x=0$	$x=1$	$x=0$	$x=1$	$x=0$
$y=1$ (death)	0	0.25	0.25	0	0.25	0.25
$y=0$ (recovery)	0.25	0	0	0.25	0.25	0.25

model B:



evidence: X = true, Y = true
$\Rightarrow P(U_2 = 1 \mid \text{evidence}) = 1.$
Then
$Y = U_2 = 1.$

model C:

evidence: X = true, Y = true
$\Rightarrow P(U_2 = 1 \mid \text{evidence}) = 1.$
Then
$Y = XU_2 + (1-X)(1-U_2)$
$= 0(1) + (1-0)(1-1)$
$= 0.$

Chapter 8: Reasoning Over Time

STATIC VS DYNAMIC INFERENCE

So far, we have assumed "static inference"; ie the world does not change.

However, we need to perform "dynamic inference" in the real world since the world evolves over time.

In particular, we need

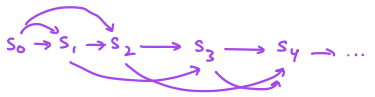
1. A set of all possible states/worlds;
2. A set of time-slices/snapshots;
3. Different probability distributions for each state at each time-slice; &
4. Dynamics encoding how distributions change over time.

STOCHASTIC PROCESS

A "stochastic process" is defined by

1. a set of states S; &
2. some stochastic dynamics $P(S_t | S_{t-1}, \dots, S_0)$.

eg



This is a Bayes net with 1 r.v. per time slice.

Problems:

1. we may have infinite variables; and so
2. we may have infinitely large conditional probability tables.

To solve this, we will assume

1. "Stationary process": dynamics do not change over time; ie the CPT is the same regardless of the time step.
2. "Markov assumption": current state depends only on a finite history of past states.

K-ORDER MARKOV PROCESS

Idea: The last k states are sufficient for inference.

eg - first-order: $P(S_t | S_{t-1}, \dots, S_0) = P(S_t | S_{t-1})$

- second-order: $P(S_t | S_{t-1}, \dots, S_0) = P(S_t | S_{t-1}, S_{t-2})$

Advantage: we can specify the entire process with finitely many time slices.

eg for 1st order: $S_{t-1} \rightarrow S_t$

- dynamics: $P(S_t | S_{t-1})$
- prior: $P(S_0)$

HIDDEN MARKOV MODELS

Motivation: In general,

1. States are not directly observable,
2. Uncertain dynamics increase state uncertainty; but
3. Observations made from sensors reduce state uncertainty.

A "Hidden Markov model" encapsulates this and includes

1. a set of states S;
2. a set of observations O;
3. a transition model $P(S_t | S_{t-1}, \dots, S_0)$;
4. an observation model $P(O_t | S_{t-1}, \dots, S_0)$; &
5. a prior $P(S_0)$.

eg 1st order HMM:

- $P(S_t | S_{t-1})$: State transition with uncertainty
- $P(O_t | S_t)$: uncertainty in measurements from sensors

INFERENCE IN TEMPORAL MODELS

I_1 We have 4 common tasks:

① "Monitoring": $P(s_t \mid o_t, \dots, o_1)$

② "Prediction": $P(s_{t+k} \mid o_t, \dots, o_1)$

③ "Hindsight": $P(s_k \mid o_t, \dots, o_1)$, $k<t$

④ "Most likely explanation": $\underset{s_0, \dots, s_t}{\text{argmin}}\, P(s_t, \dots, s_0 \mid o_t, \dots, o_1)$

I_2 We can solve ①-③ using variable elimination & ④ with a variant.

MONITORING

I_1 Idea: We want to compute

$$P(s_t \mid o_t, \dots, o_1).$$

ie the distribution of the current state given observations.

I_2 We can solve this using the "forward algorithm", which corresponds to variable elimination:

1. Factors: $P(s_0)$, $P(s_i \mid s_{i-1})$, $P(o_i \mid s_i)$, $1 \le i \le t$
2. Restrict $o_1, \dots, o_t$ to observations made
3. Sumout $s_0, \dots, s_{t-1}$: ie

PREDICTION

I_1 Goal: we want to compute

$$P(s_{t+k} \mid o_t, \dots, o_1);$$

ie the distribution over future state given observations.

I_2 We can also use the forward algorithm:

1. Factors: $P(s_0)$, $P(s_i \mid s_{i-1})$, $P(o_i \mid s_i)$, $1 \le i \le t+k$
2. Restrict $o_1, \dots, o_t$ to observations made
3. Sumout $s_0, \dots, s_{t+k-1}, o_{t+1}, \dots, o_{t+k}$

HINDSIGHT

I_1 Goal: we want to compute

$$P(s_k \mid o_t, \dots, o_1)$$

I_2 We can use "forward-backward algorithm" to solve this:

1. Factors: $P(s_0)$, $P(s_i \mid s_{i-1})$, $P(o_i \mid s_i)$, $1 \le i \le t+k$
2. Restrict $o_1, \dots, o_t$
3. Sumout $s_0, \dots, s_{k-1}, s_{k+1}, \dots, s_t$

MOST LIKELY EXPLANATION

I_1 Goal: We want to compute

$$\underset{s_0, \dots, s_t}{\text{argmax}}\, P(s_0, \dots, s_t \mid o_t, \dots, o_1).$$

I_2 We can use the "Viterbi algorithm" to solve this:

1. Factors: $P(s_0)$, $P(s_i \mid s_{i-1})$, $P(o_i \mid s_i)$, $1 \le i \le t$
2. Restrict $o_1, \dots, o_t$
3. "Maxout" $s_0, \dots, s_t$

COMPLEXITY OF TEMPORAL INFERENCE

I_1 HMMs are Bayes nets with a polytree structure.

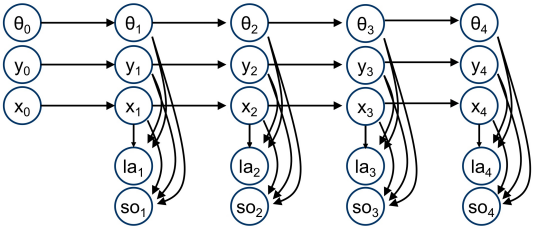
I_2 Thus, variable elimination is

① Linear wrt # of time slices; &

② Linear wrt the largest CPT.

DYNAMIC BAYESIAN NETWORKS

Idea: Encode states & observations with several random variables, and exploit conditional independence to save time & space.



This allows us to write the transition and observation models very compactly.

NON-STATIONARY PROCESS

If the process is not stationary, we can add new state components until dynamics are stationary.

NON-MARKOVIAN PROCESS

If the process is not Markovian, we can add new state components until dynamics are Markovian.

However, note this may significantly increase computational complexity.

- so we should find the smallest state description that is Markovian & stationary.

Chapter 9: Decision Tree Learning

INDUCTIVE LEARNING

θ_1 Idea: Given a training set of examples of the form $(x, f(x))$, return a "hypothesis" function h that approximates f.

θ_2 Types:
① Classification; &
② Regression.

HYPOTHESIS SPACE

The "hypothesis space" is the set of all hypotheses h that the learner may consider.

REALIZABLE

θ_1 We say a learning problem is "realizable" if the hypothesis space contains the true function.

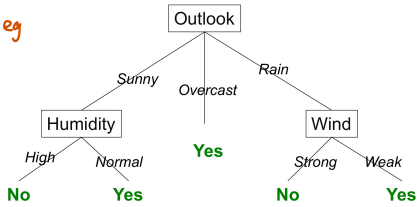
θ_2 We can use a large hypothesis space, but there is a tradeoff between the expressiveness of a hypothesis class & the complexity of finding a simple, consistent hypothesis within the space.

DECISION TREES

θ_1 A decision tree contains
① Nodes, labelled with attributes;
② Edges, labelled with attribute values;
&
③ Leaves, labelled with classes.

θ_2 Idea: Classify an instance by starting at the root, testing the attribute specified by the root, then moving down the branch corresponding to the value of the attribute; we continue this until we reach a leaf, then which we return the class.

eg



θ_3 We can express any boolean function as a decision tree.
- but some functions require exponentially large trees

INDUCING A DECISION TREE

Idea: We find a small tree consistent with the training examples by recursively choosing the most significant attribute as the root of the subtree.

Algorithm:

function DTL(*examples*, *attributes*, *default*) returns a decision tree
  if *examples* is empty then return *default*
  else if all *examples* have the same classification then return the classification
  else if *attributes* is empty then return MODE(*examples*)
  else
    $best \leftarrow$ CHOOSE-ATTRIBUTE(*attributes*, *examples*)
    $tree \leftarrow$ a new decision tree with root test *best*
    for each value v_i of *best* do
      $examples_i \leftarrow$ {elements of *examples* with $best = v_i$}
      $subtree \leftarrow$ DTL($examples_i$, $attributes - best$, MODE(*examples*))
      add a branch to *tree* with label v_i and subtree *subtree*
    return *tree*

CHOOSING AN ATTRIBUTE

In particular, at each iteration, we want to choose an attribute that is most useful for classifying examples.

Ideally, a good attribute is one that splits the examples into either "all positive" or "all negative".

- "patrons?" is better here

For a training set with p positive examples & n negative examples, the "entropy" is

$$I\left(\frac{p}{p+n}, \frac{n}{p+n}\right) = -\frac{p}{p+n}\log_2\left(\frac{p}{p+n}\right) - \frac{n}{p+n}\log_2\left(\frac{n}{p+n}\right).$$

Then, if an attribute A divides the training set E into subsets $E_1, \dots, E_v$ according to their values for A, where A has v distinct values, then the "remainder" of A is

$$\text{remainder}(A) = \sum_{i=1}^{v} \frac{p_i+n_i}{p+n} I\left(\frac{p_i}{p_i+n_i}, \frac{n_i}{p_i+n_i}\right)$$

Finally, the "information gain" (IG) of A is

$$IG(A) = I\left(\frac{p}{p+n}, \frac{n}{p+n}\right) - \text{remainder}(A)$$

We choose the attribute with the largest IG.

PERFORMANCE OF A LEARNING ALGORITHM

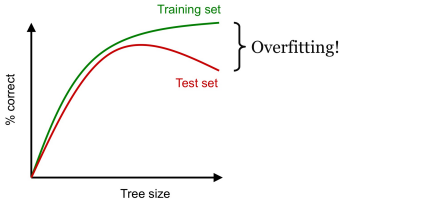
We can verify the performance of a learning algorithm by using a test set, which are examples the algorithm did not see during training.

OVERFITTING

We say a hypothesis $h \in H$ "overfits" the training data if there exists some alternative hypothesis $h' \in H$ such that

① h has smaller error than h' over the training examples; but

② h' has smaller error than h over the entire distribution of instances.



Overfitting can occur if

① the data is noisy; or

② the training set is too small to give a representative sample of the target function.

To avoid overfitting, we can

① prune statistically irrelevant nodes; or

② stop growing tree when the test set performance decreases, using "cross-validation".

CHOOSING TREE SIZE

λ_1 However, since we are now choosing the tree size based on the test set, it becomes part of the training set when optimizing the tree size.

λ_2 So, we cannot trust the test set to be representative of future accuracy.

λ_3 Solution: we split the data into

① training set: compute the decision tree;

② validation set: optimize hyperparameters

eg tree size

③ test set: measure performance

λ_4 Choosing tree size based on the validation set:

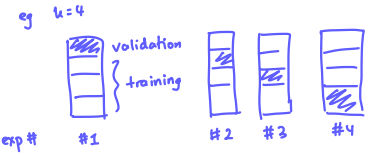
Let TS be the Tree Size
For $TS = 1$ to max value
  $decisionTree_{TS} \leftarrow train(TS, trainingData)$
  $accuracy_{TS} \leftarrow eval(decisionTree_{TS}, validationData)$
$TS^* \leftarrow argmax_{TS}\ accuracy_{TS}$
$decisionTree_{TS^*} \leftarrow train(TS^*, trainingData \cup validationData)$
$accuracy \leftarrow eval(decisionTree_{TS}, testData)$
Return $k^*, accuracy$

$eval(decisionTree, dataset)$
  $correct \leftarrow 0$
  For each $(x, y) \in dataset$
    if $y = decisionTree(x)$ then $correct \leftarrow correct + 1$
  $accuracy \leftarrow \frac{correct}{|dataset|}$
  return $accuracy$

CROSS-VALIDATION

λ_1 Idea: Repeatedly split the training data into two parts; one for training and one for validation, and then report the average validation accuracy.

eg $k=4$



↳ then take the average of the validation accuracy.

λ_2 This ensures the validation accuracy is representative of future accuracy.

λ_3 In "k-fold cross validation", we split the training data into k equal size subsets, and run k experiments, each time validating on one subset & training on the remaining subsets.
Then, we report the average validation accuracy of the k experiments.

λ_4 Selecting tree size via cross-validation:

Let TS be the Tree Size
Let k be the number of $trainData$ splits
For $TS = 1$ to max value
  For $i = 1$ to k do (where i indexes $trainData$ splits)
    $decisionTree_{TS} \leftarrow train(TS, trainData_{1..i-1,i+1..k})$
    $accuracy_{TS,i} \leftarrow eval(decisionTree_{TS}, trainData_i)$
  $accuracy_{TS} \leftarrow average(\{accuracy_{TS,i}\}_{\forall i})$
$TS^* \leftarrow argmax_{TS}\ accuracy_{TS}$
$decisionTree_{TS^*} \leftarrow train(TS^*, trainData_{1..i-1,i+1..k})$
$accuracy \leftarrow eval(decisionTree_{TS^*}, testData)$
Return $TS^*, accuracy$

Chapter 10: Statistical Learning

Idea: We have uncertain knowledge about the world, & learning reduces this uncertainty.

In particular, we have our

① hypotheses H: our probabilistic theories of the world; &

② data D: our evidence about the world.

BAYESIAN LEARNING

"Bayesian learning" consists of

① the prior $P(H)$;

② the likelihood $P(d|H)$; &

③ our evidence $d = \{d_1, \ldots, d_N\}$,

and we want to compute

$$P(H|d) = kP(d|H)P(H)$$

via Bayes' theorem.

To predict an unknown quantity X, we can use

$$P(X|d) = \sum_i P(X|d, h_i)P(h_i|d)$$
$$= \sum_i P(X|h_i)P(h_i|d)$$

EXAMPLE: CANDY

- Favorite candy sold in two flavors:
  - Lime (hugh)
  - Cherry (yum)
- Same wrapper for both flavors
- Sold in bags with different ratios:
  - 100% cherry → h_1
  - 75% cherry + 25% lime → h_2
  - 50% cherry + 50% lime → h_3
  - 25% cherry + 75% lime → h_4
  - 100% lime → h_5

} our hypotheses

Assume prior is

$$P(H) = \langle 0.1, 0.2, 0.4, 0.2, 0.1 \rangle$$

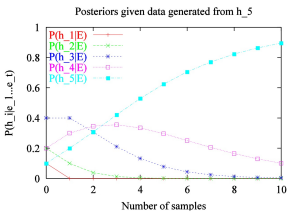
If we assume candies are "iid":

$$P(d|h) = \prod_j P(d_j|h)$$

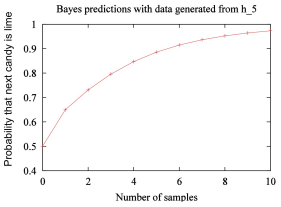
Suppose first 10 candies all taste lime:

$$\Rightarrow P(d|h_5) = 1^{10} = 1.$$
$$\Rightarrow P(d|h_3) = 0.5^{10} \approx 0.0001$$
$$\Rightarrow P(d|h_1) = 0^{10} = 0$$

Posterior:



Prediction:



BAYESIAN LEARNING PROPERTIES

Properties:

① Optimal: given prior, no other prediction is correct more often than the Bayesian one

② No overfitting: all hypotheses weighted & considered.

But when the hypothesis space is large, Bayesian learning may be intractable.

MAXIMUM A POSTERIORI / MAP

Idea: Make our prediction based on the most probable hypothesis h_{MAP}: ie

$$h_{MAP} = \underset{h_i}{\text{argmax}}\ P(h_i|d)$$

This "approximates" Bayesian learning.

eg candy example

1 lime: $h_{MAP} = h_3$, $P(\text{lime}|h_{MAP}) = 0.5$

2 limes: $h_{MAP} = h_4$, $P(\text{lime}|h_{MAP}) = 0.75$

3 limes: $h_{MAP} = h_5$, $P(\text{lime}|h_{MAP}) = 1$

etc.

However, the prediction from MAP is less accurate than the Bayesian prediction since it relies on only one hypothesis h_{MAP}.

It also has "controlled overfitting" (prior can be used to penalize complex hypotheses).

Also, finding h_{MAP} may be an intractable optimization problem!

$$\begin{aligned} h_{MAP} &= \underset{h}{\text{argmax}}\ P(h|d) \\ &= \underset{h}{\text{argmax}}\ P(h)\,P(d|h) \\ &= \underset{h}{\text{argmax}}\ P(h)\prod_i P(d_i|h) \\ &= \underset{h}{\text{argmax}}\ \left(\log P(h) + \sum_i \log P(d_i|h)\right) \end{aligned}$$

MAXIMUM LIKELIHOOD / ML

Idea: Simplify MAP by assuming the priors are uniform (ie $P(h_i) = P(h_j)\ \forall i,j$), and let

$$h_{ML} = \underset{h}{\text{argmax}}\ P(d|h).$$

and make our prediction based on h_{ML} only:

$$P(X|d) \cong P(X|h_{ML}).$$

Properties:

① Less accurate than Bayesian & MAP; but ML, MAP & Bayesian predictions converge as data increases.

② Subject to overfitting.

Finding h_{ML} is easier than finding h_{MAP}:

$$h_{ML} = \underset{h}{\text{argmax}} \sum_i \log P(d_i|h)$$

STATISTICAL LEARNING

Note,

① if the data is known, ie all attributes are known, then learning is easy.

② if the data is unknown, then learning is harder.

EXAMPLE 1: CANDY 1

- hypothesis h_θ: $P(\text{cherry}) = \theta$, $P(\text{lime}) = 1-\theta$.
- data d: c cherries, ℓ limes

ML hypothesis: θ is relative freq of observed data

↳ $\theta = \frac{c}{c+\ell}$, $P(\text{cherry}) = \frac{c}{c+\ell}$, $P(\text{lime}) = \frac{\ell}{c+\ell}$

Then

↳ $P(d|h_\theta) = \theta^c(1-\theta)^\ell$

$\Rightarrow \log P(d|h_\theta) = c\log\theta + \ell\log(1-\theta)$

$\Rightarrow \frac{d \log P(d|h_\theta)}{d\theta} = \frac{c}{\theta} - \frac{\ell}{1-\theta}$.

Set this to 0 to find optimal θ: $\Rightarrow \theta = \frac{c}{c+\ell}$

EXAMPLE 2: CANDY 2

BN: flavor → wrapper, P(F=cherry) = θ

F	P(W=red | F)
c	θ_1
l	θ_2

Hypothesis: $h_{\theta,\theta_1,\theta_2}$

Data: - c cherries; g_c green wrappers, r_c red wrappers
- l limes; g_l green wrappers, r_l red wrappers.

Then
$L(\theta,\theta_1,\theta_2) = P(d \mid h_{\theta,\theta_1,\theta_2}) = \theta^c(1-\theta)^l \theta_1^{r_c}(1-\theta_1)^{g_c} \theta_2^{r_l}(1-\theta_2)^{g_l}$.

Getting $\ell(\theta,\theta_1,\theta_2)$, and setting $\frac{\partial \ell}{\partial(\theta,\theta_1,\theta_2)} = 0$, we get

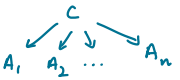
$\theta = \frac{c}{c+l}$, $\theta_1 = \frac{r_c}{r_c+g_c}$, $\theta_2 = \frac{r_l}{r_l+g_l}$.

LAPLACE SMOOTHING

1. Idea: If there is no sample for a certain outcome, we may get overfitting.
   eg no cherries eaten so far
   ↳ P(cherry) = $\theta = \frac{c}{c+l} = 0$
   ↳ this is dangerous since it rules out outcomes.
2. To solve this, we employ "Laplace (add-one) smoothing", where we add one to all counts.
   eg P(cherry) = $\theta = \frac{c+1}{c+l+2}$ (>0).

NAIVE BAYES MODEL

1. Idea: we want to predict a class C based on attributes A_i.



2. Parameters:
   ① $\theta = P(C = \text{true})$
   ② $\theta_{i1} = P(A_i = \text{true} \mid C = \text{true})$
   ③ $\theta_{i2} = P(A_i = \text{true} \mid C = \text{false})$
3. Assumption: A_i's are independent given C.
4. Note Naive Bayes models usually don't perform as well as decision tree models since the latter does not assume conditional independence of the attributes.
5. Parameter learning:
   ① Parameters: $\theta_{V, pa(V)=v}$
   - $\theta_{V, pa(V)=v} = P(V \mid pa(V) = v)$
   - we can get this from the CPTs
   ② Data d:
   $d_i = \langle V_1 = v_{1,i}, \ldots, V_n = v_{n,i} \rangle$
   ③ Max likelihood:
   $$\hat{\theta}_{V, pa(V)=v} = \frac{\#(V, pa(V)=v)}{\#(pa(V)=v)}$$
   - $pa(V)$ = parents of V

Chapter 11: Neural Networks

ARTIFICIAL NEURAL NETWORKS

Idea: Mimic the brain to do computation; in particular:
1. Nodes correspond to neurons; &
2. Links correspond to synapses (links).

UNIT

For each unit i, it has
1. Weights, w — refers to the strength of the link from unit i to unit j:

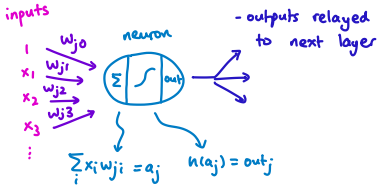
$$a_j = \sum_i w_{ji} x_i + w_{j0} = w_j \bar{x}$$

2. Activation function, h - corresponds to the numerical signal produced:

$$y_j = h(a_j).$$

- h should be non-linear

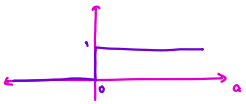
Picture:



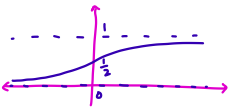
Note the unit should be "active" (ie output near 1) when fed with the "right" inputs, and "inactive" (output near 0) when fed with the "wrong" inputs.

COMMON ACTIVATION FUNCTIONS

"Threshold" function:



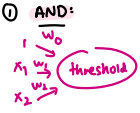
"Sigmoid" function:



LOGIC GATES

Idea: We want to design ANNs to represent boolean functions.

1. AND:



$a = w_0(1) + w_1(x_1) + w_2 x_2.$

$h(a) = \begin{cases} 0, & a \le 0 \\ 1, & \text{otherwise.} \end{cases}$

We can use $w_0 = -1.5$, $w_1 = w_2 = 1$.

2. OR:

↳ we can use $w_0 = -0.5$, $w_1 = w_2 = 1$.

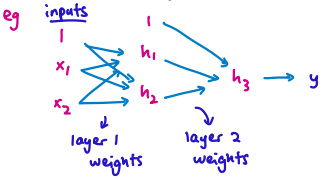
3. NOT:

↳ we can use $w_0 = 0$, $w_1 = -1$.

NETWORK STRUCTURES

Types:
1. Feed-forward network: consists of a directed acyclic graph

eg



2. Recurrent network: consists of a directed cyclic graph.
   - can memorize information

PERCEPTRON

A "perceptron" is a single layer feed-forward network.

Note a perceptron is a linear separator.

MULTILAYER NETWORKS

Idea: Neural networks with ≥ 1 hidden layer of sufficiently many sigmoid units can approximate any function arbitrarily closely.

(see slides for idea)

WEIGHT TRAINING

1. Our parameters are the weights in the layers $\langle W^{(1)}, W^{(2)}, \dots \rangle$.
2. Idea: we want to minimize the errors.
3. To do this, we can use backpropagation.

LEAST SQUARED ERROR

1. Our loss/error function is

$$E(w) = \frac{1}{2}\sum_n E_n(w)^2 = \frac{1}{2}\sum_i \| f(x_i, w) - y_i \|_2^2$$

We want to minimize this.

2. To do this, we can use sequential gradient descent:

$$w_{ji} \leftarrow w_{ji} - \eta \cdot \frac{\partial E_n}{\partial w_{ji}}$$

3. To compute the gradient efficiently, we can use backpropagation, or in reality, automatic differentiation.

BACKPROPOGATION ALGORITHM

1. First phase: forward phase - compute output z_j for each unit j.

eg

- $z_j = h(a_j)$, $a_j = \sum_i w_{ji} z_i$

2. Second phase: "backword phase" — compute δ_j at each unit j.

- For each w_{ji}: $\frac{\partial E_n}{\partial w_{ji}} = \frac{\partial E_n}{\partial a_j} \cdot \frac{\partial a_j}{\partial w_{ji}} = \delta_j z_i$,
  $\delta_j = \frac{\partial E_n}{\partial a_j}$.
- Then

$$\delta_j = \begin{cases} h'(a_j)(z_j - y_i) & \text{(base case: } j \text{ is output unit)} \\ h'(a_j)\sum_k w_{kj}\delta_k & \text{(recursive case: } j \text{ is hidden)} \end{cases}$$

- Since $a_j = \sum_i w_{ji} z_i$, thus $\frac{\partial a_j}{\partial w_{ji}} = z_i$.

EXAMPLE

<see annotated slides>

Chapter 12: Deep Neural Networks

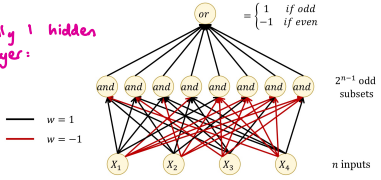
DEEP NEURAL NETWORKS

- A "deep neural network" is a NN with many hidden layers.
- Advantage: high expressivity.

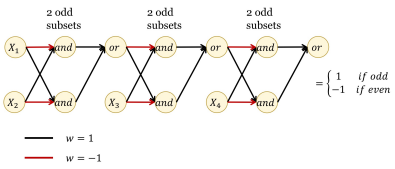
EXPRESSIVENESS

- Idea: Although NNs with 1 layer of sigmoid/hyperbolic units can approximate arbitrarily closely NNs with several layers, the number of units may decrease exponentially as the number of layers increases.
- Example: parity function

only 1 hidden layer:



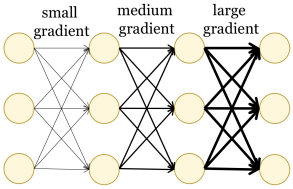
2n-2 hidden layers:



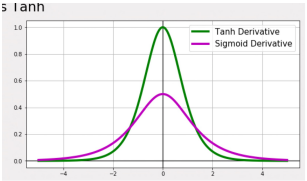
- with more hidden layers, we need less hidden nodes.

VANISHING GRADIENTS

- Idea: Deep NNs using sigmoid/hyperbolic units often suffer from vanishing gradients.



- This is because the derivatives of the Sigmoid & tanh functions are ≤ 1.



eg $y = \sigma(w_3 \sigma(w_2 \sigma(w_1 x)))$

$x \xrightarrow{w_1} h_1 \xrightarrow{w_2} h_2 \xrightarrow{w_3} h_3$

Then

$$\frac{\partial y}{\partial w_3} = \sigma'(a_3)\,\sigma(a_2)$$

$$\frac{\partial y}{\partial w_2} = \sigma'(a_3)\,w_3\,\sigma'(a_2)\,\sigma(a_1)$$

$$\frac{\partial y}{\partial w_1} = \sigma'(a_3)\,w_3\,\sigma'(a_2)\,w_2\,\sigma'(a_1)\,x$$

↓ as products of factors ≤ 1 gets "longer", the gradient vanishes

- Solution: we use the "rectified linear unit" activation function:

$$h(a) = \max(0, a).$$

  - gradient is 0 or 1
  - sparse computation

- "Soft" version / "softplus":

$$h(a) = \log(1 + e^a)$$

  - note this does not prevent gradient vanishing (gradient < 1)

OVERFITTING

Idea: As the number of parameters is often larger than the amount of data, it increases the risk of overfitting.

DROPOUT

This helps solve overfitting.

Idea: Randomly "drop" some units from the network when training.

Training: at each iteration of gradient descent:

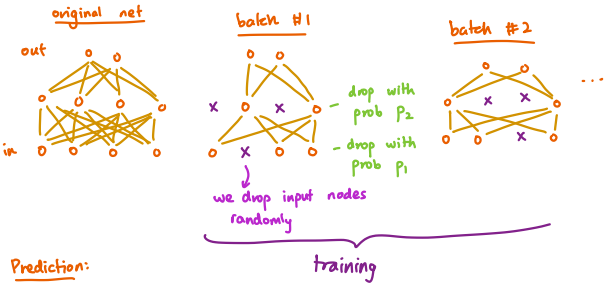
① each input unit is dropped with probability p_1; &

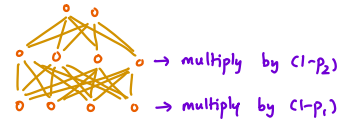
② each hidden unit is dropped with probability p_2.

Prediction:

① Multiply each input unit by $1-p_1$; &

② Multiply each hidden unit by $1-p_2$.





Algorithm:

Training: let $\odot$ denote elementwise multiplication

- Repeat
  - For each training example $(\boldsymbol{x}_n, y_n)$ do
    - Sample $z_n^{(l)}$ from $Bernoulli(1-p_l)^{k_l}$ for $1 \le l \le L$
    - Neural network with dropout applied:

$$f_n(\boldsymbol{x}_n, \boldsymbol{z}_n; \boldsymbol{W}) = h_l\left(\boldsymbol{W}^{(L)}\left[\dots h_2\left(\boldsymbol{W}^{(2)}\left[h_1\left(\boldsymbol{W}^{(1)}\left[\tilde{\boldsymbol{x}}_n \odot \boldsymbol{z}_n^{(1)}\right]\right) \odot \boldsymbol{z}_n^{(2)}\right]\right) \dots \odot \boldsymbol{z}_n^{(L)}\right]\right)$$

    - Loss: $Err(y_n, f_n(\boldsymbol{x}_n, \boldsymbol{z}_n; \boldsymbol{W}))$
    - Update: $w_{kj} \leftarrow w_{kj} - \eta \frac{\partial Err}{\partial w_{kj}}$
  - End for
- Until convergence

Prediction: $f(\boldsymbol{x}_n; \boldsymbol{W}) = h_l\left(\boldsymbol{W}^{(L)}\left[\dots h_2\left(\boldsymbol{W}^{(2)}\left[h_1\left(\boldsymbol{W}^{(1)}\left[\tilde{\boldsymbol{x}}_n(1-p_1)\right](1-p_2)\right]\right) \dots (1-p_L)\right]\right)$

Intuitively, each dropout iteration trains a different sub-network, and we merge these during training.

Chapter 13: Decision Networks

MOTIVATION

- Sometimes, we need to make decisions under uncertainty.

PREFERENCE ORDERING: $\succsim$

- A "preference ordering" $\succsim$ is a ranking of all possible states of affairs/worlds S.
  - these could be outcomes of actions, states in a search problem, etc
- In particular, we use the notation
  1. $s \succsim t \Rightarrow s$ is at least as good as t
  2. $s \succ t \Rightarrow s$ is strictly preferred to t
  3. $s \sim t \Rightarrow$ agent is indifferent between s & t

  where s & t are states.
- If an agent's actions are deterministic, then we know what states will occur.
- Otherwise, we can represent this using lotteries:

$$L = [p_1, s_1; \ldots; p_n, s_n]$$

where state s_i occurs with probability p_i.

AXIOMS

- Given 3 states A, B & C:
  1. either $A \succ B$, $A \prec B$ or $A \sim B$ (orderability);
  2. $A \succ B$, $B \succ C \Rightarrow A \succ C$ (transitivity);
  3. $A \succ B \succ C \Rightarrow \exists p$ s.t. $[p, A; 1-p, C] \sim B$ (continuity);
  4. $A \sim B \Rightarrow [p, A; 1-p, C] \sim [p, B; 1-p, C]$ (substitutability);
  5. $A \succ B \Rightarrow (p \geq q \Leftrightarrow [p, A; 1-p, B] \succsim [q, A; 1-q, B])$ (monotonicity)
  6. $[p, A; 1-p, [q, B; 1-q, C]] \sim [p, A; (1-p)q, B; (1-p)(1-q), C]$ (decomposibility)

UTILITY FUNCTION

- A "utility function" $U: S \to \mathbb{R}$ associates a "utility" with each outcome.
- In particular, $U(s)$ measures our degree of preference for s.
- Note U induces a "preference ordering" $\succsim_U$ over S by $s \preceq_U t \Leftrightarrow U(s) \leq U(t)$.

EXPECTED UTILITY: EU(d)

- Idea: Under uncertainty, each decision d induces a distribution P_d over possible outcomes, where $P_d(s)$ is the probability of outcome s under decision d.
- The "expected utility" of decision d is

$$EU(d) = \sum_{s \in S} P_d(s)\, U(s)$$

PRINCIPLE OF MAXIMUM EXPECTED UTILITY (MEU)

- MEU states the optimal decision under conditions of uncertainty is the one with the highest expected utility.

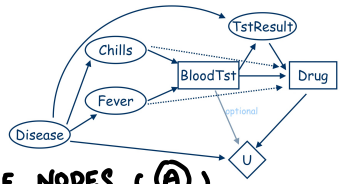
DECISION NETWORKS / INFLUENCE DIAGRAMS

"Decision networks" provide a way of representing sequential decision problems.

Idea:

① Represent the variables like in a BN;
② Add decision / controllable variables; &
③ Add utility variables that describe how good different states are.

eg



CHANCE NODES (Ⓐ)

"Chance nodes" are random variables.

- denoted by circles

Like a BN, they contain CPTs with probabilistic inference on their parents.

DECISION NODES (A)

"Decision nodes" are variables set by the decision maker.

- denoted by squares

In particular, the parents reflect information available at the time the decision is to be made.

eg

- the values of chills & fever need to be observed before the decision to take the test must be made

VALUE NODES (Ⓐ)

"Value nodes" specify utility of a state.

- denoted by a diamond

In particular, the utility depends only on the states of the parents of the value node.

ASSUMPTIONS

We assume

① decision variables are totally ordered: &
- ie decisions are made in sequence $D_1, \dots, D_n$

② "no-forgetting" property: any information that is available when decision D_i is made is available when D_j is made, $i<j$.
- thus all parents of D_i are parents of D_j
- we use dashed lines to indicate this

eg

POLICIES: δ

A policy δ is a set of mappings δ_i, one for each decision node D_i, where

$$\delta_i : \text{Dom}(\text{Par}(D_i)) \to \text{Dom}(D_i).$$

In particular, δ_i associates a decision with each parent assignment for D_i.

eg

A policy for BT could be

$\delta_{BT}(c, f) = bt$
$\delta_{BT}(c, \sim f) = \sim bt$
$\delta_{BT}(\sim c, f) = bt$
$\delta_{BT}(\sim c, \sim f) = \sim bt$

VALUE OF A POLICY: EU(δ)

The "value" of policy δ is the expected utility given that decisions are executed according to δ.

Essentially,

$$EU(\delta) = \sum_X P(X, \delta(X))\, U(X, \delta(X))$$

where $\delta(X)$ denotes the assignment to decision variables dictated by δ given the assignment X.

OPTIMAL POLICIES

δ^* is "optimal" if

$$EU(\delta^*) \geq EU(\delta)$$

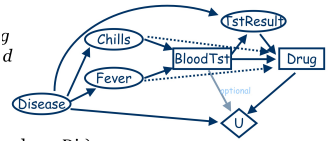
for all policies δ.

To compute the best policy:

1. Start with the last decision;
2. For each assignment to parents & for each decision value, compute the expected value of choosing that value of D;
3. Set the policy choice for each value of parents to be the value of D that has max value;
4. Repeat these steps for each decision in "reverse" order.

To compute the expected values, we can use variable elimination.

eg



- eg suppose we have asst <c, f, bt, pos> to Par(drug)
- we want $EU(Drug = md \mid c, f, bt, pos)$
- in variable elimination, we can treat C, F, BT, TR, Dr as evidence
- then eliminate remaining variables — in this case, only Disease is left
- we are left with the factor
  $EU(md \mid c, f, bt, pos) = \sum P(Dis \mid c, f, bt, pos, md)\, U(Dis, bt, md)$

Finally, we find which D maximises $EU(D \mid evidence)$, which will be in the optimal policy.

OPTIMAL POLICIES FOR BNs

In BNs, utility nodes are just factors that can be dealt with using variable elimination.

Thus, for this case, we can just use variable elimination.

OPTIMIZING POLICIES: NOTES

Idea: If a decision node D has no decisions that follow it, we can find its policy by instantiating each of its parents and computing the expected utility of each decision for each parent instantiation.

- no-forgetting ⇒ all other decisions are already instantiated.

When a decision D is optimized, we can treat it as a random variable.

- just treat the policy as a new CPT
- given parent instantiation x, D gets $\delta(x)$ with probability 1

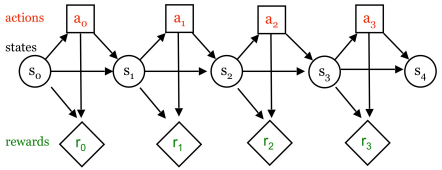
At each iteration of the decision optimization process, we can optimize D_i by using simple variable elimination calculations.

Chapter 14: Markov Decision Processes

SEQUENTIAL DECISION MAKING

"Sequential decision making" combines static decision making (eg in decision networks) & sequential inference (eg HMMs, dynamic BNs)

MARKOV DECISION PROCESSES



1. Idea: These are indefinite / infinite / large finite decision networks.
2. Formal definition: a Markov decision process has
   1. states $s \in S$;
   2. actions $a \in A$;
   3. rewards $r \in R$;
   4. transition model $P(s_t \mid s_{t-1}, a_{t-1})$
   5. reward model $R(s_t \mid a_t)$;
   6. discount factor $0 \leq \gamma \leq 1$; &
   7. horizon (# of time steps) h.
3. Our goal is to find the optimal policy; ie an optimal way to act at every state to maximize the utility / reward.

CURRENT ASSUMPTIONS

Assumptions:

1. Process is stochastic;
2. Process is sequential;
3. States are fully observable;
4. Model is complete; &
   - no learning is required
5. States & actions are discrete.
   - note that we can cycle between states.

TRANSITION MODEL: $P(s_t \mid s_{t-1}, a_{t-1})$

Assumptions:

1. Markov: $P(s_t \mid s_{t-1}, a_{t-1}, s_{t-2}, a_{t-2}, \dots) = P(s_t \mid s_{t-1}, a_{t-1})$
2. Stationary: $P(s_t \mid s_{t-1}, a_{t-1})$ is same given (s_t, a_{t-1}, s_{t-1}) $\forall t$.

REWARD MODEL

1. Reward function: $R(s_t, a_t) = r_t$
2. Assumption: the reward function is stationary; ie $R(s_t, a_t)$ is the same for a given (s, a).
3. However, the terminal reward does not have to be stationary.
   eg $+1/-1$ for winning/losing
4. Goal: maximize sum of expected rewards $\sum_t R(s_t, a_t)$.

DISCOUNTED REWARDS

1. Idea: If h is infinite, then $\sum_t R(s_t, a_t) = \infty$, which is not ideal.
2. Solution: use "discounted rewards"

$$\sum_t \gamma^t R(s_t, a_t)$$

3. where $0 \leq \gamma \leq 1$ is the discount factor.
   Intuition: we prefer utility sooner than later.

POLICY

1. The "policy" is the choice of action at each time step.
2. Formally, this maps states to actions

$$\pi(s_t) = a_t.$$

3. Goal: Find the optimal policy

$$\pi^* = \underset{\pi}{\operatorname{argmax}} \sum_{t=0}^{h} \gamma^t E_\pi[r_t]$$

POLICY OPTIMIZATION

To evaluate a policy, we can compute the value of following π:

$$V^{\pi}(s_0) = \sum_{t=0}^{h} \gamma^t \sum_{s_t} P(s_t \mid s_0, \pi) R(s_t, \pi(s_t))$$

The optimal policy is such that

$$V^{\pi^*}(s_0) \geq V^{\pi}(s_0) \quad \forall \pi.$$

Algorithms:

1. Value iteration
2. Policy iteration

Computation can be done

1. "offline": before the process starts
2. "online": as the process evolves.

VALUE ITERATION

Idea: we find the max values iteratively at the t^{th} time step:

$$V_0(s) = \max_a R(s,a) \quad \forall s$$

$$V_t(s) = \max_a R(s,a) + \gamma \sum_{s'} P(s' \mid s, a) V_{t-1}(s') \quad \forall s$$

In particular,

$$a_t = \operatorname*{argmax}_a R(s,a) + \gamma \sum_{s'} P(s' \mid s, a) V_{t-1}(s') \quad \forall s$$

Algorithm:

valueIteration(MDP)

$V_0^*(s) \leftarrow \max_a R(s,a) \ \forall s$

For $n = 1$ to h do

$V_n^*(s) \leftarrow \max_a R(s,a) + \gamma \sum_{s'} \Pr(s' \mid s, a) V_{n-1}^*(s') \ \forall s$

Return V^*

We can represent value iteration in a matrix form:

$R^a \in \mathbb{R}^{|S|}$

$V_n^* \in \mathbb{R}^{|S|}$

$T^a \in \mathbb{R}^{|S| \times |S|}$

HORIZON EFFECT

If h is finite, the policy is non-stationary, and there is no guarantee to converge.

- best action different at each time step

If h is infinite, the policy is stationary, and there is a guarantee for the value iteration to converge.

- same best action at each time step

INFINITE HORIZON

To deal with a infinite horizon, we can use

1. a large enough n and execute the policy at the n^{th} iteration; or
2. continue iterating until $|V_n - V_{n-1}| < \varepsilon$.
   - ε is the "threshold".

POLICY ITERATION

Idea: We alternate between 2 steps:

1. Policy evaluation; given π,

$$V^{\pi}(s) = R(s, \pi(s)) + \gamma \sum_{s'} P(s' \mid s, \pi(s)) V^{\pi}(s') \quad \forall s$$

2. Policy improvement:

$$\pi(s) \leftarrow \operatorname*{argmax}_a R(s,a) + \gamma \sum_{s'} P(s' \mid s, a) V^{\pi}(s') \quad \forall s$$

Algorithm:

policyIteration(MDP)

Initialize π_0 to any policy

$n \leftarrow 0$

Repeat

Eval: $V_n = R^{\pi_n} + \gamma T^{\pi_n} V_n$

Improve: $\pi_{n+1} \leftarrow argmax_a \ R^a + \gamma T^a V_n$

$n \leftarrow n + 1$

Until $\pi_{n+1} = \pi_n$

Return π_n

COMPLEXITY

Value iteration:

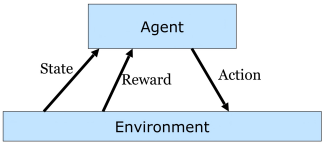
1. Each iteration: $O(|S|^2|A|)$
2. Many iterations; linear convergence

Policy iteration:

1. Each iteration: $O(|S|^3 + |S|^2|A|)$
2. Few iterations; linear-quadratic convergence

Chapter 15: Reinforcement Learning

PROBLEM



1. We want to learn to choose actions that maximize rewards.
2. We have states, actions & rewards, but do not know the transition or reward models.
3. Goal: We want to find

$$\pi^* = \arg\max_{\pi} \sum_{t=0}^{h} \gamma^t E_\pi[r_t]$$

4. Idea: We want to learn the model.

COMPONENTS

RL agents may include

1. the model $P(s'|s,a)$, $R(s,a)$;
2. the policy $\pi(s)$; &
3. the value function $V(s)$.

CATEGORIES

Value based
- No policy (implicit)
- Value function

Policy based
- Policy
- No value function

Actor critic
- Policy
- Value function

Model based
- Transition and reward model

Model free
- No transition and no reward model (implicit)

Online RL
- Learn by interacting with environment

Offline RL
- No environment
- Learn only from saved data

MODEL FREE EVALUATION

1. Idea: Given a policy π, estimate $V^\pi(s)$ without any transition or reward model.
2. Strategies:
   1. Monte-Carlo evaluation:

$$V_\pi(s) = E_\pi[\sum_t \gamma^t r_t | s, \pi] \approx \frac{1}{n(s)} \sum_{k=1}^{n(s)} \sum_t [\gamma^t r_t^{(k)} | s, \pi]$$

→ several sample approximation

   2. Temporal difference (TD) evaluation:

$$V_\pi(s) = E[r|s,\pi(s)] + \gamma \sum_{s'} P(s'|s,\pi(s)) V^\pi(s') \approx r + \gamma V^\pi(s')$$

one sample approximation

MONTE CARLO EVALUATION

1. Let

$$G_k = \sum_t \gamma^t r_t^{(k)}$$

2. Then

$$V_n^\pi(s) \approx V_{n-1}^\pi(s) + \frac{1}{n(s)}(G_{n(s)} - V_{n-1}^\pi(s))$$

3. Incremental update:

$$V_n^\pi(s) \leftarrow V_n^{\pi-1}(s) + \alpha_n(G_n - V_{n-1}^\pi(s)), \quad \alpha_n = \frac{1}{n(s)}$$

TEMPORAL DIFFERENCE EVALUATION

1. Incremental update:

$$V_n^\pi(s) \leftarrow V_{n-1}^\pi(s) + \alpha_n\left(r + \gamma V_{n-1}^\pi(s') - V_{n-1}^\pi(s)\right)$$

2. If α_n is decreased appropriately with the # of times a state is visited, then $V_n^\pi(s)$ converges to the correct value.

3. Sufficient conditions for $\alpha_n(s)$:

$$\sum_n a_n = \infty$$
$$\sum_n a_n < \infty$$

4. We often choose $\alpha_n(s) = \frac{1}{n(s)}$.

5. Algorithm:

TDevaluation(π, V^π)
Repeat
  Execute $\pi(s)$
  Observe s' and r
  Update counts: $n(s) \leftarrow n(s) + 1$
  Learning rate: $\alpha \leftarrow \frac{1}{n(s)}$
  Update value: $V^\pi(s) \leftarrow V^\pi(s) + \alpha(r + \gamma V^\pi(s') - V^\pi(s))$
  $s \leftarrow s'$
Until convergence of V^π
Return V^π

COMPARISON

Monte Carlo	TD
- unbiased estimate	- biased estimate
- high variance	- low variance
- needs many trajectories	- needs less trajectories

MODEL-FREE CONTROL

1. Idea: Instead of evaluating the state value function $V^\pi(s)$, evaluate the "state-action value function" $Q^\pi(s,a)$

$$Q^\pi(s,a) = E(r|s,a) + \gamma \sum_{s'} P(s'|s,a) V^\pi(s')$$

- value of executing a followed by π

2. Then, we use the policy

$$\pi'(s) = \underset{a}{\operatorname{argmax}}\ Q^\pi(s,a).$$

BELLMAN'S EQUATION

1. Optimal state-action value function:

$$Q^*(s,a) = E[r|s,a] + \gamma \sum_{s'} P(s'|s,a) \max_{a'} Q^*(s',a')$$

where $V^*(s) = \max_a Q^*(s,a)$,

$$\pi^*(s) = \underset{a}{\operatorname{argmax}}\ Q^*(s,a).$$

Q-LEARNING

Qlearning(s, Q^*)
Repeat
  Select and execute a — how do we choose the action?
  Observe s' and r
  Update counts: $n(s,a) \leftarrow n(s,a) + 1$
  Learning rate: $\alpha \leftarrow \frac{1}{n(s,a)}$
  Update Q-value:
  $Q^*(s,a) \leftarrow Q^*(s,a) + \alpha\left(r + \gamma \max_{a'} Q^*(s',a') - Q^*(s,a)\right)$ ↳ $\forall s, a$
  $s \leftarrow s'$
Until convergence of Q^*
Return Q^*

EXPLORATION VS EXPLOITATION

1. Idea: If the agent always chooses the action with the highest value, it is "exploiting", and the learned model is not accurate.

2. By taking random actions ("exploration"), the agent may learn the model, but parts of it will never be used.

3. Thus, we need a balance.

COMMON EXPLORATION METHODS

Methods:

1. ε-greedy: with prob ε, execute random action; otherwise, execute the best action $a^* = \underset{a}{\operatorname{argmax}}\ Q(s,a)$

2. Boltzmann exploration: increasing temp T increases stochasticity

$$P(a) = \frac{e^{Q(s,a)/T}}{\sum_a e^{Q(s,a)/T}}$$

EXPLORATION & Q-LEARNING

Q-learning converges to optimal Q-values if
1. every state is visited infinitely often;
2. the action selection becomes greedy as $t \to \infty$; &
3. the learning rate is decreased fast enough, but not too fast:

$$\sum_n \alpha_n \to \infty, \quad \sum_n (\alpha_n)^2 < \infty.$$

eg do ε-greedy, but decrease ε over time

Chapter 16: Deep Reinforcement Learning

LARGE STATE SPACES

1. Idea: For large state spaces, Q-learning is impractical since the update function has complexity proportional to the state space size.
2. We need to approximate
   1. the policy $\pi(s) \to a$;
   2. the Q-function $Q(s,a) \to \mathbb{R}$; &
   3. the value function $V(s) \to \mathbb{R}$.

Q-FUNCTION APPROXIMATION

Let $s = (x_1, \dots, x_n)^T$.

1. Linear:

$$Q(s,a) \approx \sum_i w_{ai} x_i$$

2. Non-linear (eg neural network):

$$Q(s,a) \approx g(x; w)$$

GRADIENT Q-LEARNING

1. Idea: We want to minimize the squared error between
   1. the Q-value estimate: $Q_w(s,a)$
   2. the target: $r + \gamma \max_{a'} Q_{\bar{w}}(s',a')$.
2. Squared error:

$$Err(w) = \frac{1}{2}[Q_w(s,a) - r - \gamma \max_{a'} Q_{\bar{w}}(s',a')]^2$$

3. Gradient:

$$\frac{\partial Err}{\partial w} = [Q_w(s,a) - r - \gamma \max_{a'} Q_{\bar{w}}(s',a')] \frac{\partial Q_w}{\partial w}$$

4. We can then use gradient descent.

Initialize weights w at random in $[-1,1]$
Observe current state s
Loop
  Select action a and execute it
  Receive immediate reward r
  Observe new state s'
  Gradient: $\frac{\partial Err}{\partial w} = [Q_w(s,a) - r - \gamma \max_{a'} Q_w(s',a')] \frac{\partial Q_w(s,a)}{\partial w}$
  Update weights: $w \leftarrow w - \alpha \frac{\partial Err}{\partial w}$
  Update state: $s \leftarrow s'$

CONVERGENCE OF APPROXIMATION Q-LEARNING

Given $\sum \alpha_t = \infty$, $\sum \alpha_t^2 < \infty$:

1. Linear approximation Q-learning converges; but
2. Non-linear approximation Q-learning may diverge.
   - adjusting w to increase $\hat{Q}$ at (s,a) may introduce errors at nearby state-action pairs.

MITIGATING DIVERGENCE

To mitigate divergence, we can use
① Experience replay; &
② Using 2 networks:
- Q-network; &
- target network.

EXPERIENCE REPLAY

Idea: store previous experiences <s,a,s',r> into a buffer & sample a mini-batch of previous experiences at each step to learn by Q-learning.

Advantages:
① break correlations between successive updates (more stable learning)
② less interactions with environment needed (better data efficiency)

TARGET NETWORK

Idea: Use a separate target network which is only updated periodically.

repeat for each (s, a, s', r) in mini-batch:

$$w \leftarrow w - \alpha_t \left[Q_w(s,a) - r - \gamma \max_{a'} Q_{\bar{w}}(s', a') \right] \frac{\partial Q_w(s,a)}{\partial w}$$

$\bar{w} \leftarrow w$

- similar to value iteration.

Advantage: mitigate divergence.

DEEP Q-NETWORK / DQN

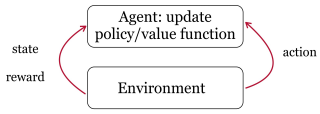
A "deep Q-network" uses gradient Q-learning with
① deep neural networks;
② experience replay; &
③ target network.

Initialize weights w and $\overline{w}$ at random in $[-1,1]$
Observe current state s
Loop
  Select action a and execute it
  Receive immediate reward r
  Observe new state s'
  Add $\langle s, a, s', r \rangle$ to experience buffer
  Sample mini-batch of experiences from buffer
  For each experience $\langle \hat{s}, \hat{a}, \hat{s'}, \hat{r} \rangle$ in mini-batch
    Gradient: $\frac{\partial Err}{\partial w} = [Q_w(\hat{s}, \hat{a}) - \hat{r} - \gamma \max_{\hat{a'}} Q_{\overline{w}}(\hat{s'}, \hat{a'})] \frac{\partial Q_w(\hat{s}, \hat{a})}{\partial w}$
    Update weights: $w \leftarrow w - \alpha \frac{\partial Err}{\partial w}$
  Update state: $s \leftarrow s'$
  Every c steps, update target: $\overline{w} \leftarrow w$

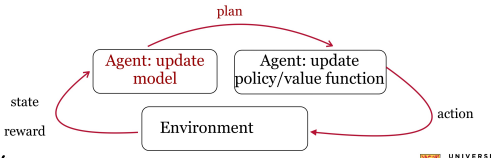
Chapter 16: Model-Based Reinforcement Learning

MODEL-FREE VS MODEL-BASED RL

In model-free online RL, there are no explicit transition or reward models.



In model-based online RL, we learn an explicit transition and/or reward model.



Benefit: Increased sample efficiency

Drawback: Increased complexity.

MODEL-BASED RL METHOD

Idea: At each step:

1. Execute action;
2. Observe resultant state & reward;
3. Update transition / reward model;
4. Update policy / value function.

Algorithm with value iteration:

ModelBasedRL(s)
  Repeat
    Select and execute a // similar to Q-learning
    Observe s' and r
    Update counts: $n(s,a) \leftarrow n(s,a)+1$,
      $n(s,a,s') \leftarrow n(s,a,s')+1$

    Update transition: $\Pr(s'|s,a) \leftarrow \frac{n(s,a,s')}{n(s,a)} \; \forall s'$

    Update reward: $R(s,a) \leftarrow \frac{r + (n(s,a)-1)R(s,a)}{n(s,a)}$

    Solve: $V^*(s) = \max_a R(s,a) + \gamma \sum_{s'} \Pr(s'|s,a) V^*(s') \; \forall s$

    $s \leftarrow s'$
  Until convergence of V^*
  Return V^*

COMPLEX MODELS

Idea: Use function approximations for the transition & reward models:

1. Linear model:

$$pdf(s'|s,a) = N(s'|w^T x, \sigma^2 I)$$

2. Non-linear model:
   - Stochastic: Gaussian process;

$$pdf(s'|s,a) = GP(s|m(\cdot), k(\cdot,\cdot))$$

   - deterministic: neural network;

$$s' = T(s,a) = NN(s,a)$$

PARTIAL PLANNING

Idea: In complex models, fully optimizing the policy/value function at each time step is intractable.

To mitigate this, we can do partial planning that involves

1. a few steps of Q-learning; &
2. learning from simulated experience.

MODEL-BASED RL WITH Q-LEARNING

ModelBasedRL(s)
  Repeat
    Select and execute a, observe s' and r
    Update transition: $w_T \leftarrow w_T - \alpha_T (T_{w_T}(s,a) - s') \nabla_{w_T} T_{w_T}(s,a)$

    Update reward: $w_R \leftarrow w_R - \alpha_R (R_{w_R}(s,a) - r) \nabla_{w_R} R(s,a)$

    Repeat a few times:
      sample $\hat{s}, \hat{a}$ arbitrarily

      $\delta \leftarrow R_{w_R}(\hat{s},\hat{a}) + \gamma \max_{\hat{a}'} Q_{w_Q}(T_{w_T}(\hat{s},\hat{a}),\hat{a}') - Q_{w_Q}(\hat{s},\hat{a})$

      Update Q: $w_Q \leftarrow w_Q - \alpha_Q \delta \nabla_{w_Q} Q_{w_Q}(\hat{s},\hat{a})$
    $s \leftarrow s'$
  Until convergence of Q
  Return Q

PARTIAL PLANNING VS REPLAY BUFFER

Idea: In model-free Q-learning with a replay buffer, we update the Q-function based on samples from the replay buffer;
in the previous algorithm, we update Q by generating samples from the model.

Replay buffer:

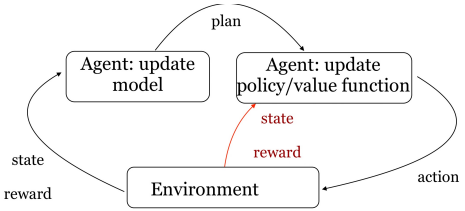
① simple;
② real samples; but
③ no generalization to other state-action pairs.

Partial planning with model:

① complex;
② simulated samples; but
③ generalization to other state-action pairs.

DYNA-Q

Idea: We learn an explicit transition & reward model & learn directly from real experience.



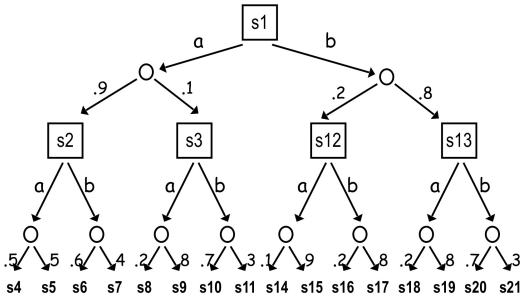
- outer loop: similar to model-based
- inner loop: similar to model-free

Algorithm:

Dyna-Q(s)
  Repeat
    Select and execute a, observe s' and r
    Update transition: $w_T \leftarrow w_T - \alpha_T(T_{w_T}(s,a) - s')\nabla_{w_T} T_{w_T}(s,a)$
    Update reward: $w_R \leftarrow w_R - \alpha_R(R_{w_R}(s,a) - r)\nabla_{w_R} R(s,a)$
    $\delta \leftarrow r + \gamma \max_{a'} Q_{w_Q}(s',a') - Q_{w_Q}(s,a)$
    Update Q: $w_Q \leftarrow w_Q - \alpha_Q \delta \nabla_{w_Q} Q_{w_Q}(s,a)$
    Repeat a few times:
      sample $\hat{s}, \hat{a}$ arbitrarily
      $\delta \leftarrow R_{w_R}(\hat{s},\hat{a}) + \gamma \max_{\hat{a}'} Q_{w_Q}(T_{w_T}(\hat{s},\hat{a}),\hat{a}') - Q_{w_Q}(\hat{s},\hat{a})$
      Update Q : $w_Q \leftarrow w_Q - \alpha_Q \delta \nabla_{w_Q} Q_{w_Q}(\hat{s},\hat{a})$
    $s \leftarrow s'$
  Return Q

PLANNING FROM CURRENT STATE: MONTE CARLO TREE SEARCH

💡1 Idea: instead of planning at arbitrary states, plan from the current state, which helps improve the next action.



💡2 To make this tractable:

① Approximate leaf values with value of default policy;

$$Q^*(s,a) \approx Q^{\pi}(s,a) \approx \frac{1}{n(s,a)} \sum_{k=1}^{n} G_k$$

② Approximate chance nodes' expectation by sampling from transition model:

$$Q^*(s,a) \approx R(s,a) + \gamma \sum_{s'} V(s')$$

③ For decision nodes, only expand the most promising actions.

$$a^* = \underset{a}{\operatorname{argmax}}\ Q(s,a) + c\sqrt{\frac{2\ln(n(s))}{n(s,a)}}$$

$$V^*(s) = \max_{a^*} Q^*(s,a^*)$$

💡3 Algorithm:

UCT(s_0)
create root $node_0$ with state $state(node_0) \leftarrow s_0$
while within computational budget do
$node_l \leftarrow TreePolicy(node_0)$
$value \leftarrow DefaultPolicy(state(node_l))$
$Backup(node_l, value)$
return $action(BestChild(node_0, 0))$

TreePolicy($node$)
while $node$ is nonterminal do
if $node$ is not fully expanded do
return $Expand(node)$
else
$node \leftarrow BestChild(node, C)$
return $node$

Expand($node$)
choose $a \in$ untried actions of $A(state(node))$
add a new child $node'$ to $node$
with $state(node') \leftarrow T(state(node), a)$ deterministic transition
return $node'$

BestChild($node$,c)

$$\text{return } \arg\max_{node' \in children(node)} V(node') + c\sqrt{\frac{(2\ln n(node))}{n(node')}}$$

DefaultPolicy($node$)
while $node$ is not terminal do
sample $a \sim \pi(a \mid state(node))$
$s' \leftarrow T(state(node), a)$
return $R(s, a)$

Single Player

Backup($node$,$value$)
while $node$ is not null do

$$V(node) \leftarrow \frac{n(node)V(node) + value}{n(node) + 1}$$

$n(node) \leftarrow n(node) + 1$
$node \leftarrow parent(node)$

Two Players (adversarial)

BackupMinMax($node$,$value$)
while $node$ is not null do

$$V(node) \leftarrow \frac{n(node)V(node) + value}{n(node) + 1}$$

$n(node) \leftarrow n(node) + 1$
$value \leftarrow -value$
$node \leftarrow parent(node)$