CS 221

(1) knowledge-based systems - elicit specific domain knowledge from experts in form of rules
if [premise] then [conclusion]

modeling - inference - learning paradigm
↳ modeling: take real world problem and reduce them into neat formal mathematical objects
↳ inference: answer questions w.r.t. the model
↳ learning: modify model. initialize skeleton of a model, and decide parameters through data

reflex-based model performs fixed sequence of computation on a given input. (ex. linear classifier, deep neural networks)
↳ limited by its simplicity (fixed feedforward)

State-based model models state of a world and transitions between states triggered by actions
↳ search problems → operating in environment w/o uncertainty
↳ Markov Decision processes → tasks w/ element of chance where distribution of randomness is known
↳ Adversarial games → handle tasks when there is an opponent working against you

State based models → solutions are procedural
Constraint satisfaction problems are variable based models
↳ only has hard constraints
Bayesian networks are variable based models where variables are random variables which are dependent on each other. (CS 228)

Optimization
↳ decouple the mathematical specification of what we want to compute from the algorithm for how to compute it
discrete optimization → dynamic programming
continuous optimization → gradient descent

HW #1

1a) $f(\theta) = \frac{1}{2} \sum_{i=1}^{n} w_i (\theta - x_i)^2$

minimize $f(\theta)$ w/

$\hookrightarrow f'(\theta) = \sum_{i=1}^{n} w_i(\theta - x_i) = 0$

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

$$\theta_{min} = \frac{\sum_{i=1}^{n} w_i x_i}{\sum_{i=1}^{n} w_i}$$

if $\sum_{i=1}^{n} w_i = 0$, θ min would be undefined

1b) $f(x) = \sum_{i=1}^{d} \max_{s \in \{1,-1\}} s x_i$

$g(x) = \max_{s \in \{1,-1\}} \sum_{i=1}^{d} s x_i$

given $x \in \mathbb{R}^d$, three cases:

$\forall x > 0, \quad f(x) = g(x)$

$\forall x < 0, \quad f(x) = g(x)$

$\exists x > 0 \wedge \exists x < 0 \quad f(x) > g(x)$

Since $f(x)$ can account for each dimension

~~because~~ whereas $g(x)$ cannot.

1c) ~~ERM exercise~~

~~Perceptron~~

$E(x) = p(6) \cdot b - p(1) \cdot a + p(\forall x \notin \{1,6\}) \cdot E$

$E = \frac{b}{6} - \frac{a}{6} - \frac{5E}{6}$

$E = b - a$

1d) given $L(p) = p^4 (1-p)^3$

$\log L(p) = \log p^4 (1-p)^3 = \log p^4 + \log (1-p)^3$

$= 4 \log p + 3 \log (1-p)$

$\frac{d}{dp} \log L(p) = \frac{4}{p} + \frac{3}{1-p}$

1e) $f(w) = \rho \left(\sum \sum (a_i^T w - b_j^T w)^2 + \lambda \left(\sqrt{\sum_{n=1}^{d} w_n^2} \right)^2 \right)$

$= \sum \sum (a_i w - b_j w)(a_i - b_j) + \lambda \sum_{n=1}^{d} w_n$

$= 2 \sum \sum (a_i^T w + b_j w)(a_i^T + b_j^T) + 2 \lambda \sum_{n=1}^{d} w_n$

2a) 6 boxes in total

box dim = $(n-a, n-b)$

$|\forall \text{ dim}| = |[(n-a),(n-b)]|_{a,b \in (0,n)} \approx n^2$

$(n^2)^6 = n^{12}$, $O(n^{12})$

2b) given task is discrete we can use dynamic programming

Algorithm => travel from $(1,1)$ to (n,n) with all possible routes and take minimum route.

runtime is $O(n^2)$

2c) 1 | 2 | 3 | 5

way = fib(# + 1)

ways(5) = 4 3 3 / (3)²

way(n) = way(n-1) + ways(n-2)

if n = 1: 1

if n = 2: 2

if n = 3: 3

if n = 4: 5

, fib(n)

2d) $f(w) = \sum_{i=1}^{n} \sum_{j=1}^{n} (a_i^T w - b_j^T w)^2 + \lambda \|w\|^2$

$= w^2 \sum_{i=1}^{n} \sum_{j=1}^{n} (a_i^T - b_j^T)^2 + \lambda \sum_{n=1}^{d} w^2$

retain w^2 to edit $O(d^2)$ for runtime

(2) linear predictors - reflex models
↳ to produce optimum predictor using loss minimization framework
↳ uses a function that maps input to output

multiclass classification → y is a category
ranking → y is a permutation
structured prediction → y is a object which is built from parts
supervised learning → data provided has input and output
↳ contrasts unsupervised learning where there is only inputs

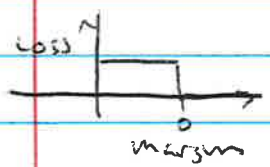
training data is a multiset of examples which form a partial specification of the desired behavior of a predictor
learning is taking the training data and producing a predictor which maps the input to the output

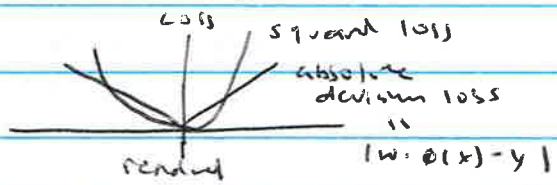
Consider predictors based on feature extractors
↳ features are the properties of the input that may be useful for predicting the output

utilize vector based representation of input → feature vector in high-dimensional vector space
A weight vector specifies the contribution of each feature vector to the prediction
↳ parameter vector of weights
Score is the weighted combination of features
↳ $w \cdot \phi(x) = \sum_{j=1}^{d} w_j \phi(x)_j$ where ϕ = feature extractor
↳ inner product of weight and feature extractor output

binary linear classifier $\Rightarrow f_w(x) = \text{sign}(w \cdot \phi(x)) = \begin{cases} +1 & \text{if } w\cdot\phi(x) > 0 \\ -1 & \text{if } w\cdot\phi(x) < 0 \\ ? & \text{if } w\cdot\phi(x) = 0 \end{cases}$
in general, binary classifier defines a hyperplane (in arbitrary # of dimensions) with weight vector
↳ points which are orthogonal $\{z \in \mathbb{R}^d : w \cdot z = 0\}$
loss minimization can be framed as an optimization problem
↳ a loss function quantifies how unhappy you would be if you used w to make a prediction on x when the correct output is y
Score on (x, y) is $w \cdot \phi(x) \Rightarrow$ how confident we are in predicting
Margin on (x, y) is $(w \cdot \phi(x))y \Rightarrow$ how correct we are

Zero-one loss - $\text{Loss}_{0\text{-}1}(x, y, w) = 1[f_w(x) \neq y]$
$= 1[(w \cdot \phi(x))y \leq 0]$

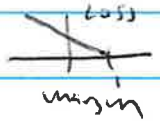




Linear regression: $f_w(x) = w \cdot \phi(x)$
↳ residual is $(w \cdot \phi(x)) - y \Rightarrow$ the amount $f_w(x)$ overshot y
↳ squared loss = $\underbrace{(f_w(x) - y)^2}_{\text{residual}}$

need to minimize $\text{TrainLoss}(w) = \frac{1}{|D_{\text{train}}|} \sum_{(x,y) \in D_{\text{train}}} \text{Loss}(x, y, w)$
gradient $\nabla_w \text{TrainLoss}(w) \Rightarrow$ direction that increases loss the most
↳ iterative optimization $\Rightarrow$ gradient descent
↳ $w \leftarrow w - \eta \nabla_w \text{TrainLoss}$ where η = step size
use stochastic gradient descent to optimize loss not objective

Hinge loss = $\max\{1 - (w \cdot \phi(x))y, 0\}$



(3) Score does framework

for feature extraction, a useful organization principle is feature template
↳ = group of features all computed in a similar way

feature vector representation
↳ array for dense feature vectors
↳ map for sparse feature vectors (most common)

hypothesis class: is the set of possible predictors w/ a fixed $\phi(x)$ and varying w, $\mathcal{F} = \{f_w : w \in \mathbb{R}^d\}$

feature extraction ⇒ domain knowledge ⇒ hypothesis class
learning = training data ⇒ obtain particular predictor

score is linear w/ $\phi(x) \cdot w$ not x itself → this can provide nonlinear decision boundary

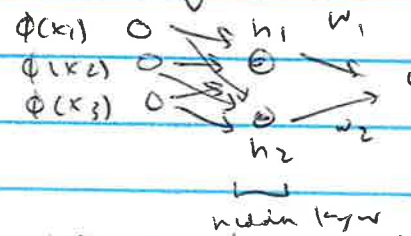
linearity allow optimization to be convex

joint learning: learn both hidden features and combination weights
↳ define $\phi(x) = [1, h_1, h_2]$
↳ $h_1 = [v_1 \cdot \phi(x) \geq 0]$
$h_2 = [v_2 \cdot \phi(x) > 0]$ ⇒ w/ fixed v, score $= \text{sign}(w_1 h_1 + w_2 h_2)$

problem: gradient of h_1 wrt $v_1 = 0$

solution: logistic function ⇒ maps $(-\infty, \infty)$ to $[0, 1]$
$\sigma(z) = (1 + e^{-z})^{-1}$
$\sigma'(z) = \sigma(z)(1 - \sigma(z))$

neural networks



can use ReLU $= \max(z, 0)$
↳ gradient doesn't disappear as z grows
↳ only involves basic operations
↳ when is computationally efficient

mapping to h is automatically learned

Computation graphs: in — mid [fmid1] — out [fmid2]

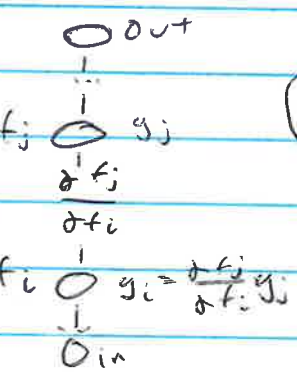
$\frac{\partial mid}{\partial in}$ $\frac{\partial out}{\partial mid}$

$\frac{\partial out}{\partial in} = \frac{\partial out}{\partial mid} \cdot \frac{\partial mid}{\partial in}$

back from
Compute the flow of info and output

forward values: f_i is the value for subexpression node at i
backward values: $g_i = \frac{\partial out}{\partial f_i}$ is how f_i influences output

forward prop computes forward values, backward prop computes backward values



neural network opinion is non convex ✗

nearest neighbor ⇒ retrieve most similar
similarity ⇒ similar examples tend to have similar output
nonparametric ⇒ hypothesis class adapts to the number of examples

(4) Score => linear predictor = $w \cdot \phi(x)$
neural network = $\sum_{j=1}^{n} w_j \sigma(v_j \cdot \phi(x))$ => less hand designed feature and harder learning
stochastic gradient descent = $w \leftarrow w - \eta \nabla_w L(x, y, w)$
rote learning => w/ strawman algorithm, minimize the objective directly
↳ overfits on training and doesn't generalize

the goal is to minimize error on unseen future examples
test set: contains examples not used for training
↳ each time the model is tested on the test set, the less good of a indicator it becomes
approximation error => how good is the hypothesis class? (larger = lower)
estimation error => how good is the learned predictor relative to the potential of the hypothesis class

given $g = \arg\min_{f \in F} Err(f)$, $Err(\hat{f}) - Err(g) + Err(g) - Err(f^*)$
given $\hat{f}$ is our learned predictor
and f^* is the target predictor

approximation error decreases as the hypothesis class increases but estimation error increases due to statistical learning theory
↳ control the # of possible values of w to control size of hypothesis class
↳ by reducing the dimensionality of w
↳ by controlling/reduce the norm (length) of $||w||$
↳ add in regularization
↳ early stopping helps since each weight update allows for higher possibility of w getting larger

hyperparameters: params of the learning algorithm (ex. features, regularization penalty)
validation set: taken out of training data which acts as a surrogate for the test set
k-fold cross-validation: divide training set in k parts. Train on k-1 parts and use other part as validation set. Iterate on new model. Take mean or median of all the validation errors.

unsupervised learning - 1) labeled data 2) help argue better problem
Data has lots of rich latent structure: want methods to discover this structure automatically
clustering: want similar points to be in the same cluster, dissimilar points to be in different cluster
↳ assigns each point into a cluster
assignment vector = $z = [z_1, \ldots z_n]$ w/ $x = [x_1, \ldots x_n]$
where $z_i \in \{1, \ldots k\}$
k-means: associates each cluster w/ a centroid s.t. each point is assigned a centroid based on proximity
alternating minimization: reduces hard problem to easier problems

w/ k-means => initialize centroids randomly
set assignments vector based on centroid
set centroids based on assignment vector
↳ average of all points in each cluster

↳ k-means may get stuck in a local minima

HW #2

1a) [pretty, good, bad, plot, not, scary]

x(1) [1, 0, 1, 0, 0, 0], -1

x(2) [0, 1, 0, 1, 0, 0], 1

x(3) [0, 1, 0, 0, 1, 0], -1

x(4) [1, 0, 0, 0, 0, 1], 1

given $Loss_{hinge}(x, y, w) = \max(0, 1 - w \cdot \phi(x) \cdot y)$

and $w = [0, 0, 0, 0, 0, 0]$

$\nabla_w Loss_{hinge} = \begin{cases} 0 & \text{when } 0 \\ -\phi(x) \cdot y & \text{when } 1 - w \cdot \phi(x) \cdot y \end{cases}$

x(1,2) $w \cdot \phi(x) \cdot y = 0$, so $w - \eta(-\phi(x) \cdot y)$

$\leftarrow [0,0,0,0,0,0] + 0.5[1,0,1,0,0,0] \cdot -1$

$\leftarrow [-0.5, 0, -0.5, 0, 0, 0]$

1b) good [1, 0, 0] 1

not good [1, 1, 0] -1

bad [0, 0, 1] -1

not bad [0, 1, 1] 1

prove $w \cdot \phi(x) y > 0$

$$\left(\begin{bmatrix} 1 & 0 & 0 \\ 1 & 1 & 0 \\ 0 & 0 & 1 \\ 0 & 1 & 1 \end{bmatrix} \cdot \begin{bmatrix} w_1 \\ w_2 \\ w_3 \end{bmatrix} \right) \cdot y = \begin{bmatrix} w_1 \\ w_1 + w_2 \\ w_3 \\ w_2 + w_3 \end{bmatrix} \cdot \begin{bmatrix} 1 \\ -1 \\ -1 \\ 1 \end{bmatrix}$$

$$= \begin{bmatrix} w_1 \\ -w_1 - w_2 \\ -w_3 \\ w_2 + w_3 \end{bmatrix} > \begin{bmatrix} 0 \\ 0 \\ 0 \\ 0 \end{bmatrix}$$

$w_1 > 0$

$w_1 + w_2 < 0$

$w_3 < 0$

$w_2 + w_3 > 0$

since $w_3 < 0$, $w_2 > 0$ since $w_2 + w_3 > 0$,

but $w_1 + w_2 < 0$ when $w_1 > 0$ so it is impossible

if you introduce either phone, "not good" or "not bad" as a feature =>

$$\begin{bmatrix} 1&0&0&0 \\ 1&1&0&1 \\ 0&0&1&0 \\ 0&1&1&0 \end{bmatrix} \begin{bmatrix} w_1 \\ w_2 \\ w_3 \\ w_4 \end{bmatrix} = \begin{bmatrix} w_1 \\ w_2+w_3+w_4 \\ w_3 \\ w_2+w_3 \end{bmatrix} p_y > 0$$

$$\left.\begin{array}{l} w_1 > 0 \\ w_1+w_2+w_4 < 0 \\ w_3 < 0 \\ w_2+w_3 > 0 \end{array}\right\} \text{allows linear classifier to get zero error}$$

2a) given $f(x) = \sigma(w \cdot \phi(x))$ where $\sigma(z) = (1+e^{-z})^{-1}$

$\text{Loss} = \frac{1}{2}(f(x) - y)^2$

2b) $\nabla_w \text{Loss} = (\sigma(w \cdot \phi(x)) - y) \cdot \frac{\partial}{\partial w}$

$\frac{\partial}{\partial w} \sigma(w \cdot \phi(x)) = -(1+e^{-w \cdot \phi(x)})^{-1} \cdot -e^{-w \cdot \phi(x)} \cdot \phi(x)$

$= -(1+e^{-w\phi(x)})^{-1} e^{-w\phi(x)} \phi(x)$

$\nabla_w \text{Loss} = [(1+e^{w\phi(x)})^{-1} - y][-e^{-w\phi(x)} \phi(x) (1+e^{-w\phi(x)})^{-1}]$

2c) $\nabla_w^{y=1} \text{Loss} = [(1+e^{w\phi(x)})^{-1} - 1][-e^{-w\phi(x)} \phi(x) (1+e^{-w\phi(x)})^{-1}]$

set $w = \infty$,

$\nabla \text{Loss} = [(1+0)^{-1} - 1][-0] = 0$ (= min)

2d) $\nabla_{w=0}^{y=1} \text{Loss} = [(1+1)^{-1} - 1][-1 \cdot \phi(x)(1+1)]$

$= (\frac{1}{2} - 1)(-\phi(2)) = \phi(x)$ (= max)

2e) $\text{Loss}_D = \frac{1}{2}(\sigma(w \cdot \phi(x)) - y) = 0$

$\sigma(w \cdot \phi(x)) - y = 0$

$y = \sigma(w \cdot \phi(x))$

$\text{Loss}_{D'} = \frac{1}{2}(w \cdot \phi(x) - y') = 0$

$w \cdot \phi(x) = y'$

$y' = \sigma^{-1}(y)$

$\sigma^{-1}(z) \Rightarrow \quad z = (1+e^{-y})^{-1}$

$\ln(z^{-1} - 1) = -y$

$\sigma^{-1}(z) = \frac{1}{\ln z^{-1} - 1}$

5 Search problems define the possibilities } state based
search algorithms explore these possibilities

↳ problem will return an entire action sequence
↳ agent reasoning about consequences of its choice of action sequence

begin by building a search tree
↳ start w/ s_{start}: starting state
↳ each edge of tree is actions w/ a corresponding cost
↳ each root-to-leaf path is an action sequence
↳ the path w/ lowest cost is the goal

backtracking search - simplest approach of trying all paths
↳ recursively called on correct thing
↳ depth first search
↳ eg. if a search tree has a max depth = D
and b available actions per state (branching factor = b)
↳ memory complexity = $O(D)$
↳ time complexity = $O(b^D)$

depth first search → to decrease time complexity, assume all action cost 0
↳ find the terminal state

Breadth first search: all action cost same, thus explores nodes in order of increasing depth
↳ maintain frontier and pops off start and pushes its successors

add in iterative deepening => modify DFS to explore up to certain depth
↳ then has cut off depth

dynamic programming => avoid exponential runtime of tree search
↳ minimize time complexity by doing computation at once

```
def backtrack(s, path):
    if end(s): update min cost path
    for each a ∈ Actions(s)
        extend path w/ successor(s,a) and cost(s,a)
        recursively call backtrack
    return min. cost
```

[State s] - cost(s,a) - [State s'] - FutureCost(s') - [end state]

$$FutureCost(s) = \begin{cases} 0 & \\ \min_{a \in Actions(s)} [Cost(s,a) + FC(Succ(s,a))] & \end{cases}$$

↳ thus store all futureCost as min. cost path to some end state
↳ converts tree into directed acyclic graph w/ all nodes since all costs are cached

state: a summary of all past actions sufficient to choose future actions optimally

this dynamic programming is backtracking search w/ memoization
↳ only works for acyclic graphs

for cyclic graphs => use uniform cost search → enumerates states in order of increasing past cost

↳ use 3 sets => explored: states we have found optimal path to,
frontier: states seen but not optimized
unexplored: state unseen

UCS: add Start to Frontier (has priority on lowest cost)
loop: check if frontier is empty
remove s w/ smallest priority from frontier
if IsEnd(s): return
add s to explored
for each a ∈ Actions(s):
get $Succ(s) \to s'$
if s' is explored, continue
update frontier w/ s' and priority $p + Cost(s,a)$

(6) Search problem: start state, possible actions, given cost, successor, end state
↳ find minimum cost path from start state to end state

forward problem: search ⇒ (state, action) ⇒ (successor, cost, end state)
inverse problem: (learning) ⇒ (state, action) ⇒ cost
modeling choice ⇒ cost(s,a) = w[a] (weight of action)

where total cost = $\sum_a w[a]$

structured perceptron: for each action: $w[a] \leftarrow 0$
for each iteration: $t = 1 \ldots T$
for each example (x, y) in Dtrain:
compute min cost y' given w
for each action, $w[a] \leftarrow w[a] - 1(y)$
for each action, $w[a] \leftarrow w[a] + 1(y')$

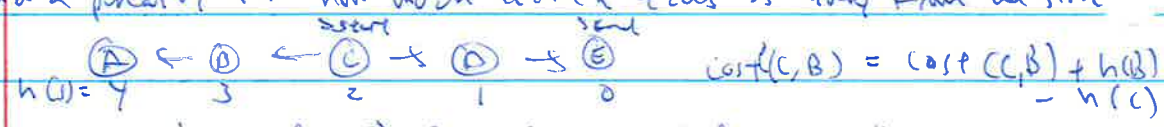
↳ thus decrease cost of true y and increase cost of predicted y

the perceptron algorithm performs SGD on a modified hinge loss w/ $\eta = 1$, $\text{Loss}(x, y, w) = \max\{-(w \cdot \phi(x))y, 0\}$ where margin of 1 is changed w/ zero.

the structured perceptron loss = $\max_{y'} \{\sum_{a \in y} w[a] - \sum_{a \in y'} w[a]\}$

↳ $w \leftarrow w - \phi(y) + \phi(y')$

A* biases the explored states UCS towards the end state
↳ UCS explores states based on $\text{PastCost}(s)$
↳ Try to utilize future cost w/ heuristic (estimation)
↳ $\text{cost}'(s,a) = \text{cost}(s,a) + h(\text{Succ}(s,a)) - h(s)$
add a penalty for how much action takes us away from end state



cannot have any heuristic ⇒ eg. negative edge cost
consistency: if $\text{cost}'(s,a) = \text{cost}(s,a) + h(\text{Succ}(s,a)) - h(s) \geq 0$
and $h(\text{End}) = 0$
correctness: if h is consistent, A* will return minimum cost path
efficiency of A*: $\text{PastCost}(s) \leq \text{PastCost}(\text{end}) - h(s)$ ⇒ larger $h(s)$ ⇒ better
A* will explore all s

admissible heuristic underestimate $\text{FutureCost}(s)$
Relaxation: compute $\text{FutureCost}(s)$ on easier problem w/ modeling
↳ closed form solution: relax the constraints of problem into easier problem to obtain numerics
↳ actual future cost of relaxed states (w/ dynamic programming or UCS)
run until all states are explored.
↳ define some relaxed problem (where edges are removed) and call UCS on that. This will be the future cost of the relaxed problem
define heuristic based on future cost of relaxed problem
↳ independence: relax original problem into independent subproblems
relaxed search problems need to have $\text{cost}_{\text{rel}}(s,a) \leq \text{cost}(s)$
$h(s) = \text{FutureCost}_{\text{rel}}(s) \Rightarrow h(s)$ is consistent
if $h_1(s), h_2(s)$ are consistent, use $h(s) = \max\{h_1(s), h_2(s)\}$
If relaxed problem has no solution, original doesn't either

HW#3 |

1a) If input = "anteater"

greedy output = "an", "tea", "ter" ← not fluent, thus higher cost

optimal output = "anteater" ← fluent thus lower cost

2a) input = "h drk t"

possible fills => h → {he}

drk → {drunk, drink, drank}

t → {ate, tea}

greedy output = "he drunk ate" ← not fluent (high cost)

optimal output = "he drank tea" ← fluent (low cost)

3a) input = "hdrkt"

same possible fill as 2a

greedy: "he drunk ate" ← not fluent (high cost)

optimal: "he drank tea" ← fluent (low cost)

(7) deterministic successor assumption is optimistic
↳ randomness: taking action might lead to any one of many possible states

total reward: utility

Markov decision process: represented as a graph where nodes include both states and chance, and edges are actions and random outcome of action
↳ states, start state, possible actions from s, probabilities of next state given action, reward, is end, and discount factor

↳ set of states and actions from each state
↳ transition distribution specifies for each state and action a distribution over possible successor states, w/ each transition associated w/ an reward.

compared to search problems, MDP has a transition probability over next state, and now we are maximizing reward instead of minimizing cost

transition probabilities $T(s, a, s')$ specify the probability of ending up in state s' if action a is taken in state s

policy π is a mapping from each state $s \in$ states to an action
↳ specifies an action for every single state, not just the states along optimal path

Markov property: regardless we end up in a state, we faced w/ the same problem and therefore should take the same optimal action
↳ satisfied though transitions and rewards

following a policy yields a random path
↳ the utility of a policy is the discounted sum of the rewards on the path (this is a random variable)

↳ the value of a policy is the expected utility ← what we want to maximize

discounting ⇒ reward is applied discounting factor exponentially
↳ this has a fairly small horizon

value of the policy is the expected utility received by following policy π from state s

q-value of policy is the expected utility if take an action from s then follow policy π

$V_\pi(s)$ (s) → $\pi(s)$ → $(s, \pi(s))$ (Q_π) → $T(s, \pi(s), s')$ → (s') V_π

↳ computes value of policy

policy evaluation → goal $V_\pi(s)$ = expected utility by following policy π

$$= \begin{cases} 0 & \text{if end} \\ Q_\pi & \text{o/w} \end{cases}, \quad Q_\pi = \sum T(\text{reward} + \gamma V_\pi(s'))$$ (?)

↳ Q_π considers all possible transitions to successor state s' and receive the application of immediate reward plus discounted future reward

↳ to derive recurrence, apply law of total expectation and Markov property.

↳ if $u_t = r_t + \gamma r_{t+1} + \dots \gamma^n r_{t+n}$

↳ $V_\pi(s) = E(u_1 \mid s_0 = s) = \sum P(s_1 = s' \mid s_0 = s, a_1 = \pi(s)) \cdot E(u_1 \mid s_1 = s', s_0 = s, a_1 = \pi(s))$

↳ $E(u \mid s_1 = s', s_0 = s, a_1 = \pi(s)) = \text{Reward}(s, \pi(s), s') + \gamma V_\pi(s')$

↳ iterative algorithm: start w/ arbitrary policy then apply recurrence to converge to the value

↳ initialize in loop: $V_\pi^{(t)}(s) \leftarrow \sum_{s'} T(s, \pi, s')[\text{Reward}(s, \pi(s), s') + \gamma V_\pi^{(t-1)}(s')]$

until values converge given $\max_{s \in \text{states}} |V_\pi^{(t)}(s) - V_\pi^{(t-1)}(s)| < \epsilon$

Value iteration → finding best policy by trying to set directly to maximum expected utility

↳ optimal value is the maximum value attained by any policy

optimal policy ⇒ $V_{opt}(s) = \begin{cases} 0 & \text{end} \\ \max_{a \in Actions(s)} Q_{opt}(s,a) & \text{o/w} \end{cases}$

$Q_{opt} = \sum T(s,a,s')[Reward(s,a,s') + \gamma V_{opt}(s')]$

V_{opt} taking the best action = largest $Q_{opt}(s,a)$

$\pi_{opt}(s) = \arg\max \max_{a \in Actions(s)} Q_{opt}(s,a)$

optimal action is to take the action a w/ largest Q_{opt}

↳ initialize and loop: $V_{opt}(s) \leftarrow \max_{a \in Actions(s)} \underbrace{\sum_{s'} T(s,a,s')(Reward + \gamma V_{opt})}_{Q_{opt}^{(t-1)}(s,a)}$

when t is iteration

↳ convergence: if either $\gamma < 1$ or MDP is acyclic ⇒ value iteration will converge

policy evaluation ⇒ $(MDP, \pi) \to V_\pi$

value iteration ⇒ MDP → (V_{opt}, π_{opt})

(8) MDP ⇒ graph w/ state and chance-state nodes w/ actions as state → chance-state and transitions as chance-state → state. Actions are controlled and transitions are probabilistic

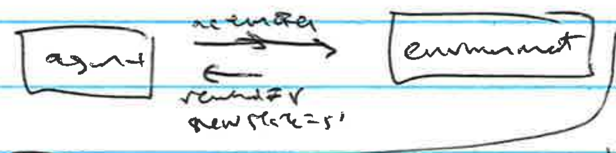
given a policy, it yields a sequence of actions w/ associated rewards ⇒ episodes (a path in MDP)

each episode has a utility ⇒ discounted sum of rewards and value ⇒ expected utility

$V_\pi(s) = \begin{cases} 0 & \text{if end} \\ Q_\pi(s, \pi(s)) & \text{o/w} \end{cases}$

$Q_\pi(s,a) = \sum_{s'} T(s,a,s')(Reward(s,a,s') + \gamma V_\pi(s'))]$

w/ MDP w/o known transitions and reward function ⇒ reinforcement learning



Model-based Monte Carlo

$\hat{T}(s,a,s') = \frac{\#\text{ times }(s,a,s')}{\#\text{ times }(s,a)}$

$\widehat{Reward}(s,a,s') = r$ in (s,a,r,s')

model-based monte carlo tries to estimate the model (transitions and rewards) using monte carlo simulation

Monte Carlo is a standard way to estimate expectation of r.v. by taking an average over samples of r.v.

Samples are not independent but come from a Markov chain so it can be shown that these estimates converge to expectation by ergodic theorem

our policy maps $s \to \pi(s)$, which allows for ?? ?? to a ?? to ?? ??, thus need exploration

distinguishes supervised learning from reinforcement learning since agent needs to only ?? ?? data

if π is nondeterministic, can explore all state/action infinitely often then estimates of transitions and reward will converge

$\hat{Q}_{opt}(s,a) = \sum_{s'} \hat{T}(s,a,s')\left[\hat{Reward}(s,a,s') + \eta \hat{V}_{opt}(s')\right]$

model free => estimates $\hat{Q}_{opt}$ directly

Q_π is expected utility starting at s, taking action a and following policy π

$u_t = r_t + \eta r_{t+1} + \ldots \eta^n r_{t+n}$

$\hat{Q}_\pi$ = average of u_t where $s_{t-1} = s$, $a_t = a$

↳ (s,a) will occur in simulation

model-free → on-policy since it depends on π policy and model-based → off-policy since the model estimates data from any policy

$\hat{Q}_\pi(s,a) \leftarrow (1-\eta)\hat{Q}_\pi(s,a) + \eta u$, u = data

η = interpolation term $\propto 1/\sqrt{\#\text{ updates to } (s,a)}$

SARSA → $u \leftarrow r + \eta \underbrace{Q_\pi(s',a')}_{\text{replaced w/ estimate}}$

↳ combines data as generated

bootstrapping => SARSA uses estimated $\hat{Q}_\pi(s,a)$ instead of raw data

estimate Q_{opt} model-free => Q-learning → off-policy

↳ $Q_{opt}(s,a) = \sum_{s'} T(s,a,s')\left[Reward(s,a,s') + \eta V_{opt}(s')\right]$

$V_{opt}(s') = \max_{a' \in Actions(s')} \hat{Q}_{opt}(s',a')$

$Q_{opt}(s,a) \leftarrow (1-\eta) Q'_{opt}(s,a) + \eta\left(r + \eta \hat{V}_{opt}(s')\right)$

exploration policy => explicitly explore (s,a)

implicitly explore by actually exploring (s',a') w/ simulated feedback in generalization

trade-off between exploration & exploitation

↳ value only epsilon-greedy

↳ $\pi_{act}(s) = \begin{cases} \arg\max_{a \in Actions(s)} Q_{opt}(s,a) & \text{prob } 1-\epsilon \\ \text{random}(Actions(s)) & \text{prob } \epsilon \end{cases}$

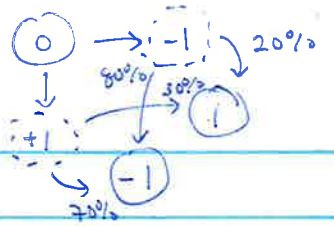
Q-learning => $\hat{Q}_{opt} \leftarrow \hat{Q}_{opt} - \eta\left(\hat{Q}_{opt}(s,a) - \eta \hat{V}_{opt}(s)\right)$

function approximation → $\hat{Q}_{opt} = w \cdot \phi(s,a)$

parameterized => $\phi(s,a)$ = features, w = weights } generalization

HW4

(a)



$$r = \begin{cases} 20 & s = -2 \\ 100 & s = 2 \\ -5 & \text{o/w} \end{cases}$$

Send = 2 or -2, $\gamma = 1$

initialize $V_{opt}(s) = \{-2:0, -1:0, 0:0, 1:0, 2:0\}$

$$V_{opt}(s) \leftarrow \max_{a \in Actions(s)} \sum_{s'} T(s,a,s')\left[Reward(s,a,s') + \gamma V_{opt}(s')\right]$$

for $i=1$

$V_{opt}(s=0) = (a_{+1}, a_{-1}) = ((0.7(-5) + 0.3(-5)), (0.8(-5) + 0.2(-5)))$

$= (a_{+1}, a_{-1}) = (-5, -5)$

$V_{opt}(s=1) = (a_{+1}, a_{-1}) = (0.7(-5) + 0.3(100)), (0.8(-5) + 0.2(100))$

$= (a_{+1}, a_{-1}) = (26.5, 16)$

$V_{opt}(s=-1) = (a_{+1}, a_{-1}) = (0.7(20) + 0.3(-5), (0.8(20) + 0.2(-5))$

$= (a_{+1}, a_{-1}) = (15, 12.5)$

for $i=2$, $V_{opt}(s) = \{-2:0, -1:15, 0:-5, 1:26.5, 2:0\}$

$V_{opt}(s=0) = (a_{+1}, a_{-1}) = ((0.7(-5-15) + 0.3(-5-26.5), (0.8(-5-15) + 0.2(-5-26.5))$

$= (14, 11)$

$V_{opt}(s=1) = (a_{+1}, a_{-1}) = ((0.7(26.5-5) + 0.3(26.5+100)), (0.8(26.5-5), 0.2(26.5+100))$

$= (53, 42.5)$

$V_{opt}(s=-1) = (a_{+1}, a_{-1}) = ((0.7(15+20) + 0.2(15-5)), (0.8(15+20) + 0.2(15-5))$

$= (27.5, 30.0)$

(b) $\pi_{opt} = \{-2:0, -1:a_{-1}, 0:a_{+1}, 1:a_{+1}, 2:0\}$

2b) since acyclic MDP has no cycles, there no probability of revisiting a node already explored. Thus with 1 pass, we can compute V_{opt} for each node.

2c) multiply all transition probabilities by γ and introduce new transitions to new end state w/ probability of $(1-\gamma)$. Set original MDP discount to 1

(9) use tree to describe the possibilities of games ⇒ game tree
two-player zero-sum ⇒ turns, state of game fully observable
↳ zero-sum is the utility of agent and opponent is zero
↳ each state has a designated player which specifies whose turn it is and only that player gets to choose the action for the state s st Players(s) = ...
↳ only when utility function are collected thus an ultra-delayed utility
∀ all utility is at end state, different players in control at different states

Policies → deterministic: actions that player take in state (s) st $\pi(s) \in Actions(s)$
↳ stochastic: probability of player taking action a in states st $\pi_p(s,a) \in (0,1)$

game evaluation { value / expected utility = 3 cases at s
$$\begin{cases} Utility(s) & IsEnd(s) \\ \sum_{a \in Actions(s)} \pi_{agent}(s,a) V_{eval}(Succ(s,a)) & Player = agent \\ \sum_{a \in Actions(s)} \pi_{opp}(s,a) V_{eval}(Succ(s,a)) & Player \text{ is } opp \end{cases}$$

expectimax ⇒ tries to find best policy ⇒ at max nodes take the max, otherwise take the average

$$V_{exptmax}(s) = \begin{cases} Utility(s) & IsEnd(s) \\ \max_{a \in Actions(s)} V_{exptmax}(Succ(s,a)) & player = agent \\ \sum_{a \in Actions(s)} \pi_{opp}(s,a) V_{exptmax}(Succ(s,a)) & player = opp \end{cases}$$

minimax ⇒ assume worst-case for assuming opponent's policy ⇒ minimize agent's utility
↳ take min at leaf nodes then choose max at root

$$V_{minimax}(s) = \begin{cases} Utility(s) & IsEnd \\ \max_{a \in Actions(s)} V_{minimax}(Succ(s,a)) & agent \\ \min_{a \in Actions(s)} V_{minimax}(Succ(s,a)) & opp \end{cases}$$

↳ $V(\pi_{max}, \pi_{min}) \geq V(\pi_{agent}, \pi_{min})$ for all π_{agent}
↳ $V(\pi_{max}, \pi_{min}) \leq V(\pi_{max}, \pi_{opp})$ for all π_{opp}
minimax policy is not optimal for an agent if opponent is known not to be playing the adversarial policy

expectiminimax ⇒ take min then expected then max at root

ply ⇒ each level in tree = # players · depth of tree
speed up minimax w/ → evaluation function: use domain-specific knowledge
alpha-beta pruning: general purpose

can also limit depth of search by decrementing d_{max} at last player's turn
pruning: maintain lower and upper bounds on values ⇒ if intervals don't overlap, then
alpha-beta pruning: a gets min, b gets max, and (choose optimally w/o knowing values) following an ordering that may be good on heuristic/eval function.

(10) $\pi_{agent}(s,w) = \arg\max_{a \in Actions(s)} V(Succ(s,a);w)$
$\pi_{opp}(s,w) = \arg\min_{a \in Actions} V(Succ(s,a);w)$
~~can generate data using policies based on current $V(s;w)$~~
↳ gives episode of state, action and reward
prediction = $V(s;w)$ and target = $r + \eta V(s';w)$
objective function = $\frac{1}{2}(\text{prediction} - \text{target})^2$
↳ then gradient wrt w and update w

TD learning: on each (s,a,r,s') ← temporal difference learning
$$w \leftarrow w - \eta\,(V(s;w) - (r + \eta V(s';w)))\,\nabla_w V(s;w)$$
~~where $V(s;w) = w \cdot \phi(s)$ for linear functions~~
and $\nabla_w V(s;w) = \phi(s)$

Q-learning operates on $Q_{opt}(s,a)$, off-policy and no knowledge of transitions

TD learning operates on $V_\pi(s)$, on-policy and need to know rules of game

↳ Q value → how good a given is to take in state

↳ value function → how good it is to be in a state

simultaneous game ⇒ no ordering on players moves

↳ pure strategy = single action $\in$ Actions (deterministic)

↳ mixed strategy = probability distribution, $0 \le \pi(a) \le 1$ (stochastic)

↳ game value ⇒ the value of the game if player A follows π_A and player B follows π_B: $V(\pi_A, \pi_B) = \sum_{a,b} \pi_A(a)\pi_B(b)V(a,b)$

↳ since both players want to optimize simultaneously, proved

$\max_b \min_a V(a,b) \le \min_a \max_b V(a,b)$ ⇒ other don't matter if can reach

↳ if player A chose mixed strategy, he must tell player B the full probability distribution he using

↳ for any fixed mixed strategy π_A:

$\min_{\pi_B} V(\pi_A, \pi_B)$ can be attained by pure strategy

↳ von Neumann minimax theorem

↳ $\max_{\pi_A}\min_{\pi_B} V(\pi_A,\pi_B) = \min_{\pi_B}\max_{\pi_A} V(\pi_A,\pi_B)$

↳ pure strategy ⇒ going second is better } during w/ simultaneous minimax moves

↳ mixed strategy ⇒ doesn't matter }

non-zero-sum ⇒ introduce payoff matrix ⇒ $V_p(\pi_A,\pi_B)$ for player p

↳ Nash equilibrium ⇒ $V_A(\pi_A,\pi_B) \ge V_A(\pi_A',\pi_B)$ for all π_A'

$V_B(\pi_A,\pi_B) \ge V_B(\pi_A,\pi_B')$ for all π_B'

(sum any finite # of pure strategy Nash equilibria is at least one)

HW #5) 1a) $V_{minimax}(s,d) = \begin{cases} \text{utility}(s) & \text{if isEnd}(s) \\ \text{Eval}(s) & \text{if } d = 0 \\ \max_{a \in Actions} V_{minimax}(Succ(s,a), d-1) & \text{player} = \text{agent} \\ \min_{a \in Actions} V_{minimax}(Succ(s,a), d) & \text{player} = \text{opp} \end{cases}$ ← decrement d on player = agent

3a) $V_{expmax}(s,d) = \begin{cases} \text{utility}(s) & \text{if isEnd}(s) \\ \text{Eval}(s) & \text{if } d = 0 \\ \max_{a \in Actions} V_{expmax}(Succ(s,a), d-1)) & \text{player} = \text{agent} \\ \frac{1}{|Actions(s)|} \cdot \sum_{a \in Actions(s)} V_{expmax}(Succ(s,a), d)) & \text{player} = \text{opp} \end{cases}$

(11) Constraint satisfaction problems

↳ variable ordering doesn't affect correctness
↳ variables are interdependent in a local way } variable based models

Modeling ⇒ solving to problem → assignments to variables
Inference ⇒ decision about variable ordering, etc, how to find assignment

factor graphs consists of a set of variables and set of factors

← variables $X = (X_1 \dots X_n)$ where $X_i \in Domain$
← factors $f_1 \dots f_m$ for each $f_j(x) \geq 0$
↳ expresses how good that assignment is

scope of factor f_j is the set of variables it depends on
arity of factor f_j is the # of variables in the scope
unary factors ⇒ arity = 1
factor graph specifying all the local interactions between variables
↳ assignment specifies a value for each variable
↳ each assignment is associated w/ a weight, which is just a product in each factor evaluated on that assignment

each assignment $x = (x_1 \dots x_n)$ has a weight: $Weight(x) = \prod_{j=1}^{m} f_j(x)$
↳ objective: find the maximum weight assignment = $\arg\max_x Weight(x)$

constraint satisfaction problem: - factor graph where all factors are constraints

$f_j(x) \in \{0, 1\}$ for all $j = 1 \dots m$
constraint is satisfied iff $f_j(x) = 1$

↳ an assignment x is consistent iff $Weight(x) = 1$

partial assignment ⇒ weight = product of all factor whose scope includes only assigned variables

dependent factors ⇒ $D(x, X_i)$ = set of factors depending on X_i and x but not on unassigned variables

Backtracking search: recursively takes in a partial assignment, its weight, w domains
↳ if x is complete assignment, update best & return
↳ choose unassigned variable X_i
↳ order values domain of chosen X_i
↳ for each value v in that order:
↳ $\delta \leftarrow \prod_{f_j \in D(x, X_i)} f_j(x \cup \{X_i : v\})$
↳ if $\delta = 0$: continue
↳ domains' ← domains via LookAhead
↳ backtrack $(x \cup \{X_i : v\}, w\delta, Domains')$

Lookahead ⇒ forward checking is a way to return an early lookahead
↳ assign values and preemptively remove inconsistent values from domains of neighbor values until some variable have empty domain and we stop and backtrack
↳ thus eliminate inconsistent values from domains of X_j neighbors

Choose the next unassigned variable as the most constraint / fewest current values ①
try values of newly assigned value by decreasing / impose the fewest # of conflicts ②
most constrained variable ⇒ ① ⇒ useful since factors are constraints
least constraining value ⇒ ② ⇒ useful when all factors are continuous
Arc consistency - eliminate value from domains ⇒ reduce branching
↳ eliminate any value using two variables which don't work
↳ enforcing arc consistency on X_i wrt X_j, remove from X_j with values from X_i
AC-3 - iteratively enforce arc consistency on all variables ← not always effective
↳ start w/ X_j next its neighbors
↳ if any neighbor's domain changes, choose this to enforce arc consistency next

(12) alternative ways to find maximum weight assignments efficiently w/o incurring the full cost of backtracking search

Can use greedy => sub-optimal solutions

↳ take highest weight partial and traverses downwards

beam search → keep track of more than one best partial assignment

↳ maintain candidate list => may be suboptimal

↳ tradeoff between time and accuracy

local search → modify complete assignments by changing one variable at a time

↳ Iterated Conditional Modes (ICM)

↳ initialize x to random complete assignment

↳ loop until convergence:

→ compute weight of $x_v = x \cup \{X_i : v\}$ for each v

→ $x \leftarrow x_v$ w/ highest weight

↳ may get stuck at local optimum

Gibbs sampling → utilized randomness via epsilon-greedy

↳ change so choose $x \leftarrow x_v$ w/ probability proportional to its weight

try to leverage graph properties to derive efficient algorithms which are exact

independence → let A, B be a partitioning of variables X

↳ say A, B are independent if no edges between A and B

↳ $A \perp B$

w/ conditioning => try to disconnect graph by adding factors then set fixed values of X_i

↳ graph transformation, we create new factors to condition on variables to be inserted

↳ remove X_i if conditioning on X_i and introduce

$g_j(x) = f_j(x \cup \{X_i : v\})$

X_1 —$f(x_1, x_2)$— X_2 => X_1 — $g(x_1) = f(x_1, B)$

if condition on $X_2 = B$

conditional independence => $A \perp B \mid C$

↳ every path from A to B goes through C

Markov blanket => $C = MB(A)$

let $B = X \setminus (A \cup C)$, then $A \perp B \mid C$

↳ $MB(A)$ = neighbors of A that are not in A

↳ what to condition on to make A conditionally independent from the rest

elimination => add factors and maximize over all values of X_i

↳ this computes the best factors for all possible assignments to the Markov Blanket of X_i and is stored as new factor U

conditioning $X_2 = B$ elimination X_2

X_1 — $g(x_1) = f(x_1, B)$ X_1 — $h(x_1) = \max_{x_2} f(x_1, x_2)$

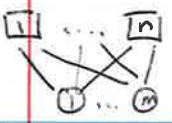
variable elimination

↳ for $i = 1, \ldots, n$: eliminate X_i (produce new factors)

for $i = n, \ldots, 1$: set X_i to maximize value in new factors

treewidth of a factor graph is the maximum arity of any factor created by variable elimination w/ the best variable ordering

HW#6

Q4) 

$\{f_1 \ldots f_n\} = F$, for all $f_j \in F$, $f_j(x) \in \{0,1\}$ ← light bulbs

constraints : n-ary constraint

variables : $\{x_1 \ldots x_n\} = X$ for all $x_i \in X$, $x_i = \begin{cases} 1 & \text{if } x_i \in T_j \\ 0 & \text{o/w} \end{cases}$

Since all light bulbs are initialized off, and each constraint is defined by n-ary, the toggle for each lightbulb should be odd so you can take the sum of all constraint corresponding to each lightbulb should be odd.

Q6) 

$x_1, x_2, x_3 \in \{0,1\}$, need both $t_1, t_2 = 1$

$x_1 x_2 x_3$	t_1	t_2
0 0 0	0	0
0 0 1	0	1
0 1 0	1	1
0 1 1	1	0
1 0 0	1	0
1 0 1	1	1
1 1 0	0	1
1 1 1	0	0

i) 2

ii) backtrack($\emptyset$, 1, $\{0,1\}$)
- ↳ $x_1 = 0$
- ↳ $\gamma \leftarrow 1$

backtrack($\{x_1: 0, x_3: 0\}$, 1, $\{0,1\}$)
- ↳ $x_2 = 1$
- ↳ $\gamma \leftarrow 1$
- ↳ complete

backtrack($\{x_1: 0, x_3: 1\}$, 1, $\{0,1\}$)
- ↳ nothing

↳ $x_1 = 1$
$\gamma = 1$

backtrack($\{x_1: 1, x_3: 1\}$, 1, $\{0,1\}$)
- ↳ $x_2 = 0$
- ↳ $\gamma \leftarrow 1$
- ↳ complete

backtrack($\{x_1: 1, x_3: 0\}$, 1, $\{0,1\}$)
- ↳ nothing

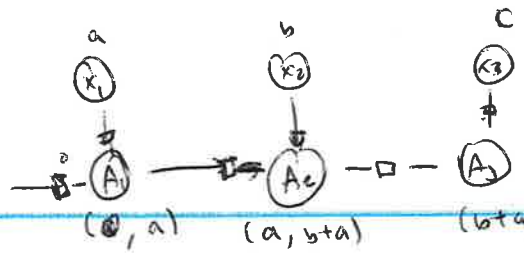
iii) backtrack($\emptyset$, 1, $\{0,1\}$)
- ↳ $x_1 = 0$
- ↳ $\gamma \leftarrow 1$

backtrack($\{x_1 = 0\}$, 1, $\{0,1\}$)

enforce arc consistency
- ↳ add $x_1 = 0$ to set
- while set not empty
  - ↳ remove $x_1 = 0$ from set
  - ↳ $D(x_2) = \{1\}$, add x_2 to set
  - ↳ remove $x_2 = 1$ from set
  - ↳ $D(x_3) = \{0\}$, add x_3 to set
  - ↳ remove x_3
  - ↳ break

complete assignment

2a)



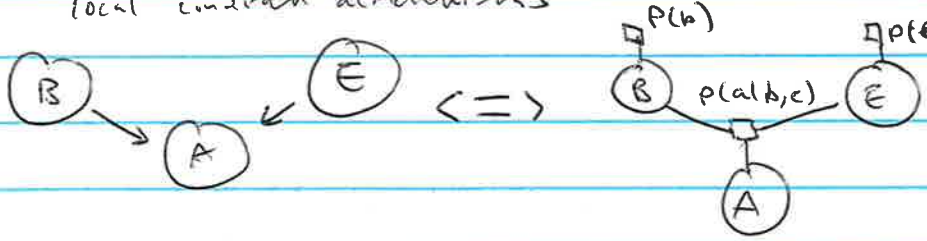
auxiliary shared hold (past, sum)
↳ domain = {0, sum}

(13) bayesian networks → factor graphs instead w/ probability
↳ reasoning under uncertainty
joint distribution specifies probability for two r.v's
marginal distribution focuses on one r.v while also taking into account the other r.v
conditional distribution select a column or row r.v & normalize the probabilities of the other r.v

get sum joint distribution ⇒ $P(S, R, T, A)$
$P(R \mid T=1, A=1)$, w/ S marginalized out } probability inference
(R = query, T=1, A=1 = condition)

modeling - specify a joint distribution ⇒ bayesian networks (factor graphs to specify joint distributions)
inference - compute queries efficiently ⇒ variable elimination, Gibbs sampling, particle filtering
reasoning under uncertainty ⇒ given many unknowns, how does evidence about some affect our beliefs about other unknowns

$P(B=b, E=e, A=a) \overset{def}{=} p(b)\,p(e)\,P(a \mid b, e)$
↳ connect variables as directed edges ⇒ directed acyclic graphs
↳ local conditional distribution ⇒ factor of the variable given its parent variables
↳ define joint distribution over all random variables as the product of all local conditional distributions



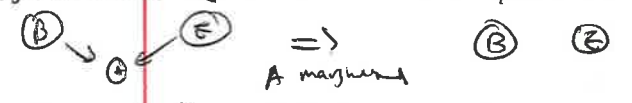
↳ can be used to capture common reasoning patterns and causality
explaining away ⇒ suppose two causes positively influence an effect. conditioned on the effects, conditioning on one cause reduces probability of the other cause

bayesian network - let $X = (X_1, \ldots X_n)$ be r.v's
↳ a bayesian network is a directed acyclic graph (DAG) that specifies a joint distribution over X as a product of local conditional distributions. $P(X_1 = x_1, \ldots X_n = x_n) = \prod p(x_i \mid x_{parents(i)})$
↳ graph structure specifies what other variables a given variable depends on
↳ specify local conditional distribution for X_i ; which is a function that specifies a distribution over X_i given an assignment $x_{parents(i)}$ to its parents in the graph, the joint distribution is simply defined to be the product of all the local conditional distributions
locally normalized ⇒ all factors (local conditional distributions) sum to 1

$$\sum_{x_i} p(x_i \mid x_{parents(i)}) = 1 \quad \text{for each } (\text{each } i)$$

↳ implies consistency of sub-matrix as conditional distribution

marginalization of a latent node yields a bayesian network w/o node
↳ useful for larger bayesian networks



↳ local conditional distributions are the true conditional distributions

$P(D=d \mid A=a, B=b) = p(d \mid a,b)$

$p(A=a, B=b, D=d) = p(a)\,p(b)\,p(d \mid a,b)$ since A, B are parent to D

probabilistic program: a randomized program that invokes a random number generator to make random choices

when variables only depend on preceding variables, => chain-structured Bayesian network
↳ Markov model

↳ basic model => for each $i = 1 \ldots n$
↳ generate word $x_i \sim p(x_i \mid x_{i-1})$

hidden Markov model => for each step $t = 1 \ldots T$
↳ generate $H_t \sim p(H_t \mid H_{t-1})$
↳ generate $E_t \sim p(E_t \mid H_t)$

↳ introduce parallel sequence of observation variables

factorial HMM => for each $t = 1, \ldots T$
for each object $o \in \{a, b\}$:
generate location $H_t^o \sim p(H_t^o \mid H_{t-1}^o)$
generate sensor reading $E_t \sim p(E_t \mid H_t^a, H_t^b)$

↳ track multiple objects w/ assume each object moves independently
acting as a Markov model

naive bayes => generate label $Y \sim p(Y)$
for each $i = 1 \ldots L$
generate $W_i \sim p(W_i \mid Y)$

latent Dirichlet allocation => generate a distribution over topics $\phi \in \mathbb{R}^K$
for each position $i = 1 \ldots L$
generate topic $Z_i \sim p(Z_i, \phi)$
generate word $W_i \sim p(W_i, Z_i)$

given input = bayesian network, evidence, query
output = $p(Q = q, E = e)$ for all q

general probabilistic inference strategy
↳ remove/marginal variables that are not ancestors of Q or E
↳ convert bayesian network to factor graph
↳ condition on $E = e$
↳ remove rows disconnected from Q
↳ run probabilistic inference algorithm

Beam search

(+) forward-backward algorithm can be used to compute typical queries of interest
filtering => asks for the distribution for some hidden variable H_i conditioned
on only the evidence up until that point => real time
smoothing => ask for the distribution of some hidden variable H_i conditioned
on all evidence => retrospectively figure out H_i

given $H_1 \to H_2 \to H_3$, unfold lattice representation => start → $H_1=1$, $H_2=1$, $H_3=1$ → ... → end

↳ where edge start → $H_1=1$ has weight $p(h_1)p(e_1 \mid h_1)$

↳ $H_{i-1} = h_{i-1} \to H_i = h_i$ has weight $p(h_i \mid h_{i-1})p(e_i \mid h_i)$

↳ each path is an assignment w/ weight equal to the product of edge weights

Forward: $F_i(h_i) = \sum_{h_{i-1}} F(h_{i-1}) w(h_{i-1}, h_i)$

sum of weights of paths from start to h_i

Backward: $B_i(h_i) = \sum_{h_{i+1}} B_{i+1}(h_{i+1}) w(h_i, h_{i+1})$

sum of weights from $H_i = h_i$ to end

define $S_i(h_i) = F_i(h_i) B_i(h_i)$

= sum of the wghts over all paths from start node to any node that goes h_i

normalize S_i

forward-backward algorithm

= share intermediate computation across different queries

particle filtering => perform approximate probabilistic inference (similar to beam search)

= propose, then extends the current partial assignment

= weight/resample reallocates resources on the particles based on evidence

propose => extend each current partial assignment (particles stochastically)

weight => weight each particle by $p(e_i | h_i) = w(h_i, \ldots h_i)$

resample => want to prune smaller weights or based on distribution

= if resampled on higher weights, keep diversity

= this can resample based on distribution of entire weights

particle filtering

= initialize $C = [\{\}]$

for $i = 1 \ldots n$

propose

$C' \leftarrow \{h \cup \{H_i : h_i\} : h \in C, h_i \sim p(h_i | h_{-i})\}$

reweight

compute weights $w(h) = p(e_i | h_i)$ for $h \in C'$

resample

$C \leftarrow k$ elements drawn independently from $\propto w(h)$

Gibbs resampling

= initialize x to random complete assignment

loop $i = 1 \ldots n$ over variables

{= compute weight of $b \cup \{X_i : v\}$ for each v

{= choose $b \cup \{X_i : v\}$ w/ probability proportional to weight

probabilistic inference

set $X_i = v$ w/ prob $P(X_i = v \mid X_{-i} = x_{-i})$

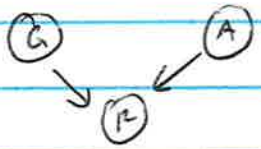
= performs a random walk in space of all possible assignments

(15) supervised learning on bayesian networks =>

use parameters => collection of distributions $\theta = \{p_d : d \in D\}$

for each variable x_i is sampled from p_{d_i}:

$$P(X_1 = x_1, \ldots X_n = x_n) = \hat{\prod_{i=1}} p_{d_i}(x_i \mid x_{\text{parents}(i)})$$



$$P(h = g, A = a, R = r) = p_G(g)\, p_A(a)\, p_R(r \mid g, a)$$

parameter sharing => local conditional distributions of different r.v. use the same parameters => more flexible expressive less expressive model

maximum likelihood for bayesian networks

↳ count

↳ for each $x \in Data$

for each variable x_i:

increment $count_{d_i}(x_{parent(i)}, x_i)$

↳ normalize

for each d and local assignment $x_{parent(i)}$:

set $p_d(x_i \mid x_{parent(i)}) \propto count_d(x_{parent(i)}, x_i)$

↳ max likelihood objective → try to find θ to maximize probability of training examples

$$= \max_\theta \prod_{x \in Data} P(X = x; \theta)$$

Laplace smoothing: regularized by adding λ to all counts and normalized to get probability estimates

if we don't observe value for every variable, we expectation maximization

↳ given H is hidden, $\forall E = e$ if observed

initialize θ

$E \Rightarrow$ ↳ compute $q(h) = P(H = h \mid E = e; \theta)$ for each h

↳ create weighted points (h, e) w/ weight $q(h)$

M-step) compute maximum likelihood to get θ

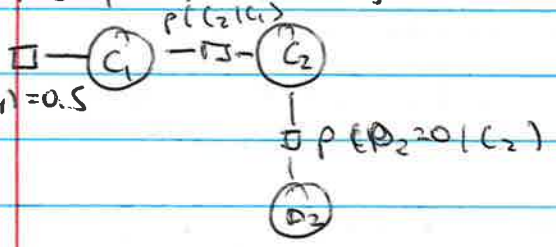
repeat until convergence

ours: bayesian network with many variables

↳ learn: maximum likelihood (Laplace smoothing + EM)

↳ $Q \mid E \Rightarrow$ parameter $\theta \Rightarrow P(Q \mid E; \theta)$

HW7) (a) $P(C_1 = 1 \mid D_2 = 0)$



$$P(C_2 \mid C_1) = \begin{cases} C_1 & C_2 & \\ 0 & 0 & 1-\epsilon \\ 0 & 1 & \epsilon \\ 1 & 0 & \epsilon \\ 1 & 1 & 1-\epsilon \end{cases}$$

$$P(C_1 = 1 \mid D_2 = 0) = p(c_1) \cdot P(c_2 \mid c_2)$$

(b) $P(C_2 = 1 \mid D_2 = 0, D_3 = 1)$

conditioning on $D_2 = 0 \Rightarrow$ $\begin{cases} C_2 & D_2 & \\ 0 & 0 & 0.5(1-n) \\ 1 & 0 & 0.5\,n \end{cases}$

$$p(C_3 \mid C_2) = \begin{cases} C_2 & C_3 & \\ 0 & 0 & 1-\epsilon \\ 0 & 1 & \epsilon \\ 1 & 0 & \epsilon \\ 1 & 1 & 1-\epsilon \end{cases}$$

$p(C_2, C_3, D_2=0) = \begin{cases} D_2 & C_2 & C_3 & \\ 0 & 0 & 0 & 0.5(1-\epsilon)(1-n) \\ 0 & 0 & 1 & 0.5(\epsilon)(1-n) \\ 0 & 1 & 0 & 0.5(\epsilon)\,n \\ 0 & 1 & 1 & 0.5(1-\epsilon)\,n \end{cases}$

conditioning on $D_3 = 1$

$p(C_2, C_3, D_2=0, D_3=1) = \begin{cases} D_2 & D_3 & C_2 & C_3 & \\ 0 & 1 & 0 & 0 & 0.5(1-\epsilon)(1-n)n \\ 0 & 1 & 0 & 1 & 0.5\,\epsilon(1-n)^2 \\ 0 & 1 & 1 & 0 & 0.5\,\epsilon n^2 \\ 0 & 1 & 1 & 1 & 0.5\,\epsilon(1-n)n \end{cases}$

$p(C_2, D_2=0, D_3=1) = \begin{cases} D_2 & D_3 & C_2 & \\ 0 & 1 & 0 & 0.5(1-\epsilon)((1-n)n + 0.5\epsilon(1-n)^2 \\ 0 & 1 & 1 & 0.5\,\epsilon n^2 + 0.5\,\epsilon(1-n)n \end{cases}$

$p(C_2=1 \mid D_2=0, D_3=1) = \dfrac{0.5\epsilon n^2 + 0.5\epsilon(1-n)n}{0.5(1-\epsilon)(1-n)n + 0.5\epsilon(1-n)^2 + 0.5\epsilon n^2 + 0.5(1-n)n}$

16) logical inference: apply set of truth-preserving rules to arrive at the answer
↳ contrasts model checking → try to directly find assignment
logical language ⇒ formal such as propositional logic, first order logic
↳ captures declarative knowledge
↳ want to represent knowledge and reason with knowledge
syntax → defines a set of valid formulas (validity of expressions)
semantics → specifies meaning of a formula, is a set of configurations of the world in which formula holds
inference rules → operate drawing on the syntax using a set of rules
syntax of propositional logic ⇒
↳ symbols (atomic formulas) or atoms ⇒ A, B, C, ...
↳ logic connectives are used to combine atoms recursively to build formulas {¬: not, ∧: conj. and, ∨: disj. or, ⇒: implies, ...}
need semantics to give meaning
↳ model w in propositional logic is an assignment of truth values to propositional symbols
↳ interpretation function: take formula and model and return whether model satisfy formula ⇒ $I(f, w)$
↳ models is a set of model that satisfy $I(f, w) = 1$
knowledge base ⇒ set of formula representing their conjunction (KB)
entailment ⇒ KB entails f iff $M(KB) \subseteq M(f)$ [$KB \models f$]
contradiction ⇒ KB contradicts f iff $M(KB) \cap M(f) = \emptyset$
contingent ⇒ nontrivial overlap
Ask and Tell are operations on formulas wrt KB
satisfiability ⇒ $M(KB) \neq \emptyset$
model checking ⇒ input: KB, output: exists satisfying model
modus ponens inference rule ⇒ $\dfrac{P,\ P \Rightarrow q}{q}$ (premise) / (conclusion) } capture of the reasoning pattern
formal inference ⇒ input set of rules, apply until no change to KB,
(choose set of formulas $f_1 \ldots f_n \in KB$, if matching rule $f_1 \ldots f_n \Rightarrow g$ exists, add g to KB)
derivation ⇒ $f(KB \vdash f)$ iff f gets added to KB
soundness ⇒ $\{f : KB \vdash f\} \subseteq \{f : KB \models f\}$ → nothing but the truth
completeness ⇒ $\{f : KB \vdash f\} \supseteq \{f : KB \models f\}$ → whole truth
definite clause ⇒ if a conjunction of symbols holds, then some other symbol holds
Horn clauses ⇒ definite clause or goal clause

17) ground resolution inference rule => resolution = $\frac{f_1 \vee \dots \vee f_n \vee p, \ \neg p \vee g_1 \vee \dots \vee g_m}{f_1 \vee \dots \vee f_n \vee g_1 \vee \dots \vee g_m}$

↳ takes two ground clauses

Conjuctive normal form => conjunction of clauses

limitations of propositional logic => missing objects & predicates and quantifiers

first order logic => constants, variables, function of term (Terms) and variables

↳ atomic formulas, connectives, quantifiers (formulas)

$\forall$ (universal quantifier) - conjunction : $\forall x P(x) = P(A) \wedge P(B) \wedge \dots$

$\exists$ (existential quantifier) - disjunction : $\exists x P(x) = P(A) \vee P(B) \vee \dots$

a model in first order logic maps constants to objects, predicate to tuples of objects

relations => unique name and domain closure (at most and at least one constant symbol)

propositionalization => one to one mapping from common law propositional logic