Move summations inwards as far as possible

- $P(B \mid j, m) = \alpha \sum_{e,a} P(B)\, P(e)\, P(a|B,e)\, P(j|a)\, P(m|a)$
- $= \alpha\, P(B) \sum_e P(e) \sum_a P(a|B,e)\, P(j|a)\, P(m|a)$

If evidence, start with factors that select that evidence

We eliminate all vars other than query + evidence

Result will be a selected joint of query and evidence

$$P(B|j,m) \propto P(B,j,m)$$

$P(B)$	$P(E)$	$P(A\|B,E)$	$P(j\|A)$	$P(m\|A)$

$$\begin{aligned}
P(B|j,m) &\propto P(B,j,m) \\
&= \sum_{e,a} P(B,j,m,e,a) && \text{marginal can be obtained from joint by summing out} \\
&= \sum_{e,a} P(B)P(e)P(a|B,e)P(j|a)P(m|a) && \text{use Bayes' net joint distribution expression} \\
&= \sum_e P(B)P(e)\sum_a P(a|B,e)P(j|a)P(m|a) && \text{use } x^*(y+z) = xy + xz \\
&= \sum_e P(B)P(e)f_1(B,e,j,m) && \text{joining on a, and then summing out gives } f_1 \\
&= P(B)\sum_e P(e)f_1(B,e,j,m) && \text{use } x^*(y+z) = xy + xz \\
&= P(B)f_2(B,j,m) && \text{joining on e, and then summing out gives } f_2
\end{aligned}$$

- The size of the largest factor determines the time and space complexity of VE
- The elimination ordering can greatly affect the size of the largest factor.
  - E.g., previous slide's example 2^{n+1} vs. 2^2

Does there always exist an ordering that only results in small factors?

- No!

Inference in Bayes' nets is NP-hard, however in polytree:

For poly-tree BNs, the complexity of VE is ***linear in the BN size*** (number of CPT entries) with the following elimination ordering:

- Convert to a factor graph
- Take Q as the root
- Eliminate from the leaves towards the root

- For i=1, 2, …, n (in topological order)
  - Sample X_i from $P(X_i \mid parents(X_i))$
- Return $(x_1, x_2, \ldots, x_n)$ Prior Sampling

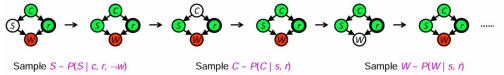
- Input: evidence $e_1,..,e_k$
- For i=1, 2, …, n Rejection
  - Sample X_i from $P(X_i \mid parents(X_i))$
  - If x_i not consistent with evidence
    - Reject: Return, and no sample is generated in this cycle
- Return $(x_1, x_2, \ldots, x_n)$

- Input: evidence $e_1,..,e_k$
- $w = 1.0$
- for i=1, 2, …, n Likelihood
  - if X_i is an evidence variable
    - x_i = observed value $_i$ for X_i
    - Set $w = w * P(x_i \mid Parents(X_i))$
  - else
    - Sample x_i from $P(X_i \mid Parents(X_i))$
- return $(x_1, x_2, \ldots, x_n)$, w

- Likelihood weighting still has weaknesses
  - The values of ***upstream*** variables are unaffected by ***downstream*** evidence
  - With many downstream evidence, we may
    - mostly get samples that are inconsistent with the evidence and thus have very small weights
    - get a few lucky samples with very large weights, which dominate the result

Gibbs Sampling Example: $P(S \mid r)$

- Step 1: Fix evidence
  - R = true
- Step 2: Initialize other variables
  - Randomly
- Step 3: Repeat
  - Choose an arbitrary non-evidence variable X
  - Resample X from $P(X \mid markov\_blanket(X))$



Sample $S \sim P(S \mid c, r, \neg w)$ Sample $C \sim P(C \mid s, r)$ Sample $W \sim P(W \mid s, r)$

Filtering: $P(X_t \mid e_{1:t})$

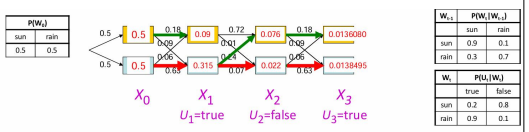
- ***belief state*** — posterior distribution over the most recent state given all evidence

Most likely explanation: $\arg\max_{x_{0:t}} P(x_{0:t} \mid e_{1:t})$

- Ex: speech recognition, decoding with a noisy channel

$$\begin{aligned}
P(X_{t+1}|e_{1:t+1}) &= P(X_{t+1}|e_{1:t}, e_{t+1}) \\
&= \alpha\, P(e_{t+1}|X_{t+1}, e_{1:t})\, P(X_{t+1}|\, e_{1:t}) \\
&= \alpha\, P(e_{t+1}|X_{t+1})\, P(X_{t+1}|\, e_{1:t}) \\
&= \alpha\, P(e_{t+1}|X_{t+1}) \textstyle\sum_{x_t} P(x_t \mid e_{1:t})\, P(X_{t+1}|\, x_t, e_{1:t}) \\
&= \alpha\, P(e_{t+1}|X_{t+1}) \textstyle\sum_{x_t} P(x_t \mid e_{1:t})\, P(X_{t+1}|\, x_t)
\end{aligned}$$

Cost per time step: $O(|X|^2)$ where $|X|$ is the number of states



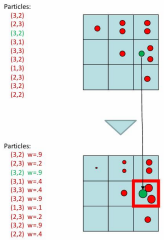
- $m_{1:t+1} = P(e_{t+1}|X_{t+1}) \max_{x_t} P(X_{t+1}|\, x_t)\, m_{1:t}[x_t]$

Every HMM is a DBN

Every discrete DBN can be represented by a HMM

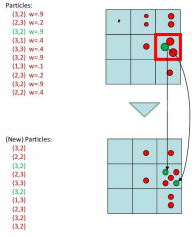
- Each HMM state is Cartesian product of DBN state variables
  - E.g., 3 binary state variables => one state variable with 2^3 possible values
- Advantage of DBN vs. HMM?
  - Sparse dependencies => exponentially fewer parameters
  - E.g., 20 binary state variables, 2 parents each; DBN has 20 x 2^{2+1} = 160 parameters, HMM has 2^{20} x 2^{20} =~ 10^{12} parameters

- Similar to likelihood weighting, weight samples based on the evidence
  - $W = P(e_t \mid x_t)$
  - Particles that fit the evidence better get higher weights, others get lower weights



- What happens if we repeat the Propagate-Observe procedure over time?
  - It is exactly likelihood weighting (if we multiply the weights)
  - Weights drop quickly… Observe

- Rather than tracking weighted samples, we ***resample***
  - Generate N new samples from our weighted sample distribution
  - Each new sample is selected from the current population of samples; the probability is proportional to its weight.
  - The new samples have weight of 1



- Now the update is complete for this time step, continue with the next one

Resample

- MDPs are non-deterministic search problems
  - One way to solve them is with expectimax search
  - We'll have a new tool soon

For MDPs, we want an optimal policy $\pi^*: S \rightarrow A$

- A policy π gives an action for each state
- An optimal policy is one that maximizes expected utility if followed
- An explicit policy defines a reflex agent

- How to discount?
  - Each time we descend a level, we multiply in the discount once

Discounting

- Why discount?
  - Sooner rewards probably do have higher utility than later rewards
  - Also helps our algorithms converge

Recursive definition of value:

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

$$Q^*(s,a) = \sum_{s'} T(s,a,s') \left[R(s,a,s') + \gamma V^*(s')\right]$$

$$V^*(s) = \max_a \sum_{s'} T(s,a,s') \left[R(s,a,s') + \gamma V^*(s')\right]$$

Problem 1: States are repeated

- Idea: Only compute needed quantities once

Problem 2: Tree goes on forever

- Idea: Do a depth-limited computation, but with increasing depths until change is small

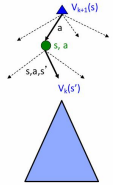
Define $V_k(s)$ to be the optimal value of s if the game ends in k more time steps

- Equivalently, it's what a depth-k expectimax would give from s

Start with $V_0(s) = 0$: no time steps left means an expected reward sum of zero

Given vector of $V_k(s)$ values, do one ply of expectimax from each state:

$$V_{k+1}(s) \leftarrow \max_a \sum_{s'} T(s,a,s') \left[R(s,a,s') + \gamma V_k(s')\right]$$



Repeat until convergence Value Iteration

Complexity of each iteration: $O(S^2A)$

Theorem: will converge to unique optimal values

Problem 1: It's slow – $O(S^2A)$ per iteration

Problem with Value Iteration

Problem 2: The "max" at each state rarely changes

- The policy often converges long before the values

We need to do a mini-expectimax (one step)

$$\pi^*(s) = \arg\max_a \sum_{s'} T(s,a,s')[R(s,a,s') + \gamma V^*(s')]$$

This is called policy extraction, since it gets the policy implied by the values

- Value iteration: find successive (depth-limited) values
  - Start with $V_0(s) = 0$
  - Given V_k, calculate the depth k+1 values for all states:

$$V_{k+1}(s) \leftarrow \max_a \sum_{s'} T(s,a,s') \left[R(s,a,s') + \gamma V_k(s')\right]$$

Q-value iteration

- But Q-values are more useful, so compute them instead
  - Start with $Q_0(s,a) = 0$
  - Given Q_k, calculate the depth k+1 q-values for all q-states:

$$Q_{k+1}(s,a) \leftarrow \sum_{s'} T(s,a,s') \left[R(s,a,s') + \gamma \max_{a'} Q_k(s',a')\right]$$

- Policy iteration: an alternative approach for value iteration
  - Step 1: Policy evaluation: calculate utilities for some fixed (not optimal) policy
  - Step 2: Policy improvement: update policy using one-step look-ahead with resulting converged (but not optimal!) utilities as future values
  - Repeat steps until policy converges

Policy Iteration

- It's still optimal!
- Can converge (much) faster under some conditions

Idea 1: Iterative updates (like value iteration)

- Start with $V_0^\pi(s) = 0$
- Given V_k^π, calculate the depth k+1 values for all states:

$$V_{k+1}^\pi(s) \leftarrow \sum_{s'} T(s,\pi(s),s')[R(s,\pi(s),s') + \gamma V_k^\pi(s')]$$

s, π(s)

- Repeat until convergence
- Efficiency: $O(S^2)$ per iteration

Policy Evaluation

Idea 2: Without the maxes, the Bellman equations are just a linear system

$$V^\pi(s) = \sum_{s'} T(s,\pi(s),s')[R(s,\pi(s),s') + \gamma V^\pi(s')]$$

- Solvable with a linear system solver

$$\pi_{i+1}(s) = \arg\max_a \sum_{s'} T(s,a,s') \left[R(s,a,s') + \gamma V^{\pi_i}(s')\right]$$

Both value iteration and policy iteration compute the same thing (all optimal values)

In value iteration:

- Every iteration updates both the values and (implicitly) the policy
- We don't track the policy, but taking the max over actions implicitly recomputes it

In policy iteration:

- We do several passes that update utilities with fixed policy (each pass is fast because we consider only one action, not all of them)
- After the policy is evaluated, a new policy is chosen (slow like a value iteration pass)
- May converge faster

Both are dynamic programs for solving MDPs

Model-Based Idea:

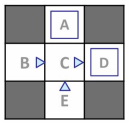
- Learn an approximate model based on experiences
- Solve for values as if the learned model was correct

Step 1: Learn empirical MDP model

- Count outcomes s' for each s, a
- Normalize to give an estimate of $\hat{T}(s,a,s')$
- Discover each $\hat{R}(s,a,s')$ when we experience (s, a, s')

Step 2: Solve the learned MDP

- For example, use value iteration, as before



Assume: $\gamma = 1$

Observed Episodes (Training)

Episode 1	Episode 2
B, east, C, -1 C, east, D, -1 D, exit, x, +10	B, east, C, -1 C, east, D, -1 D, exit, x, +10
Episode 3	**Episode 4**
E, north, C, -1 C, east, D, -1 D, exit, x, +10	E, north, C, -1 C, east, A, -1 A, exit, x, -10

Learned Model

$\hat{T}(s,a,s')$

T(B, east, C) = 1.00
T(C, east, D) = 0.75
T(C, east, A) = 0.25
…

$\hat{R}(s,a,s')$

R(B, east, C) = -1
R(C, east, D) = -1
R(D, exit, x) = +10
…

Passive RL: Model Free

- Act according to π Direct Evaluation
- Every time you visit a state, write down what the sum of discounted rewards turned out to be
- Average those samples

It wastes information about state connections

Each state must be learned separately

So, it takes a long time to learn

We can't use policy eval don't know T R.

Temporal difference learning of values

- (Policy still fixed, still doing evaluation!)
- Move the value towards the sample

Sample of V(s): $sample = R(s,\pi(s),s') + \gamma V^\pi(s')$

Update to V(s): $V^\pi(s) \leftarrow (1-\alpha)V^\pi(s) + (\alpha)sample$

Same update: $V^\pi(s) \leftarrow V^\pi(s) + \alpha(sample - V^\pi(s))$

TD value leaning is a model-free way to do policy evaluation, mimicking Bellman updates with running sample averages

However, if we want to turn values into a (new) policy… Don't know T R!

Q-value iteration

$$Q_{k+1}(s,a) \leftarrow \sum_{s'} T(s,a,s') \left[R(s,a,s') + \gamma \max_{a'} Q_k(s',a')\right]$$

Q-Learning: learn Q(s,a) values as you go

- Receive a sample (s,a,s',r)
- Consider your old estimate: $Q(s,a)$
- Consider your new sample estimate:

$$sample = R(s,a,s') + \gamma \max_{a'} Q(s',a')$$

- Incorporate the new estimate into a running average:

$$Q(s,a) \leftarrow (1-\alpha)Q(s,a) + (\alpha)\,[sample]$$

Amazing result: Q-learning converges to optimal policy -- even if you're acting suboptimally!

This is called off-policy learning

Several schemes for forcing exploration

- Simplest: random actions (ε-greedy)
  - Every time step, flip a coin
  - With (small) probability ε, act randomly
  - With (large) probability 1-ε, act on current policy
- Problems with random actions?
  - You do eventually explore the space, but keep thrashing around once learning is done
- One solution: lower ε over time
- Another solution: exploration functions

- When to explore?
  - Explore states that haven't been sufficiently explored
  - Eventually stop exploring
- Idea: select actions based on modified Q-value
  - Exploration function: takes a Q-value estimate u and a visit count n, and returns an optimistic utility, e.g. $f(u,n) = u + k/n$

Exploration Functions

  - Q-Update

Regular Update: $Q(s,a) \leftarrow_{\alpha} R(s,a,s') + \gamma \max_{a'} Q(s',a')$

Modified Update: $Q(s,a) \leftarrow_{\alpha} R(s,a,s') + \gamma \max_{a'} f(Q(s',a'), N(s',a'))$

This propagates the "bonus" back to states that lead to under-explored states

Regret is a measure of your total mistake cost: the difference between your (expected) rewards, including youthful suboptimality, and optimal (expected) rewards

Minimizing regret goes beyond learning to be optimal – it requires optimally learning to be optimal

Using a feature representation, we can write a q function (or value function) for any state using a few weights:

$$V(s) = w_1 f_1(s) + w_2 f_2(s) + \ldots + w_n f_n(s)$$

$$Q(s,a) = w_1 f_1(s,a) + w_2 f_2(s,a) + \ldots + w_n f_n(s,a)$$

Disadvantage: states may share features but actually be very different in value!

Q-learning with linear Q-functions:

transition $= (s,a,r,s')$

difference $= \left[r + \gamma \max_{a'} Q(s',a')\right] - Q(s,a)$

$Q(s,a) \leftarrow Q(s,a) + \alpha$ [difference] Exact Q's

$w_i \leftarrow w_i + \alpha$ [difference] $f_i(s,a)$ Approximate Q's (based on online least squares)

Intuitive interpretation:

- Adjust weights of active features
- E.g., if something unexpectedly bad happens, blame the features that were on: disprefer all states with that state's features

Idea: learn policies that maximize rewards, not the values that predict them

Policy Search

Policy search: start with an OK solution (e.g., approximate Q-learning), then fine-tune feature weights to find a better policy

Using empirical rate will overfit the training data!

Why does overfitting occur?

- Training data is not representative of the true data distribution
  - Too few training samples
  - Training data is noisy
- Too many attributes, some of them irrelevant to the classification task
- The model is too expressive
  - Ex: the model is capable of memorizing all the spam emails in the training set

Avoid overfitting

- Acquire more training data (not always possible)
- Remove irrelevant attributes (not always possible)
- Limit the model expressiveness by regularization, early stopping, pruning, etc.

Laplace's estimate (extended):

- Pretend you saw every outcome k extra times

$$P_{LAP,k}(x) = \frac{c(x)+k}{N+k|X|}$$

- k is the strength of the prior

Laplace for conditionals:

- Smooth each condition independently:

$$P_{LAP,k}(x|y) = \frac{c(x,y)+k}{c(y)+k|X|}$$

In practice, Laplace often performs poorly for P(X|Y):

- When |X| is very large
- When |Y| is very large

- Classify with current weights

$$y = \begin{cases} +1 & \text{if } w \cdot f(x) \geq 0 \\ -1 & \text{if } w \cdot f(x) < 0 \end{cases}$$

If wrong: adjust the weight vector by adding or subtracting the feature vector.

- If correct (i.e., y=y*), no change!

$$w = w + y^* \cdot f$$

Convergence: if the training is separable, perceptron will eventually converge (binary case)

Mistake Bound: the maximum number of mistakes (binary case) related to the *margin* or degree of separability

$$\text{mistakes} < \frac{k}{\delta^2}$$

LASSO (Least Absolute Shrinkage and Selection Operator)

$$L(\boldsymbol{w}) = \sum_i (y_i - \boldsymbol{w}^T \boldsymbol{x}_i)^2 + \lambda \sum_k |w_k|$$

Ridge Regression

$$L(\boldsymbol{w}) = \sum_i (y_i - \boldsymbol{w}^T \boldsymbol{x}_i)^2 + \lambda \sum_k w_k^2$$

"Ockham's razor": prefer the simplest hypothesis consistent with the data

$$w = \frac{n\sum_{i=1}^n x_i y_i - \sum_{i=1}^n x_i \sum_{i=1}^n y_i}{n\sum_{i=1}^n x_i^2 - (\sum_{i=1}^n x_i)^2}$$

$$b = \frac{\sum_{i=1}^n y_i \sum_{i=1}^n x_i^2 - \sum_{i=1}^n x_i \sum_{i=1}^n x_i y_i}{n\sum_{i=1}^n x_i^2 - (\sum_{i=1}^n x_i)^2}$$

What could "similar" mean?

- One option: small (squared) Euclidean distance

$$\text{dist}(x,y) = (x-y)^T(x-y) = \sum_i (x_i - y_i)^2$$

- Many other options, often domain specific

For each point, re-assign to closest mean:

K-mean

$$a_i = \arg\min_k \text{dist}(x_i, c_k)$$

Move each mean to the average of its assigned points:

$$c_k = \frac{1}{|\{i : a_i = k\}|} \sum_{i:a_i=k} x_i$$

两步都不能使得总距离 phi 函数增加

Objective: $\text{argmax}_\theta \prod_j \sum_{i=1}^k P(y_j=i, x_j|\theta) = \sum_j \log \sum_{i=1}^k P(y_j=i, x_j|\theta)$

Data: $\{x_j \mid j=1 \ldots n\}$

Notation a bit inconsistent

Parameters = θ=λ

- **E-step**: Compute expectations to "fill in" missing y values according to current parameters, θ
  - For all examples j and values i for y, compute: $P(y_j=i \mid x_j, \theta)$
- **M-step**: Re-estimate the parameters with "weighted" MLE estimates
  - Set $\theta = \text{argmax}_\theta \sum_j \sum_{i=1}^k P(y_j=i \mid x_j, \theta) \log P(y_j=i, x_j|\theta)$

Iterate: On the *t*'th iteration let our estimates be

$$\theta^{(t)} = \{\mu_1^{(t)}, \mu_2^{(t)} \ldots \mu_k^{(t)}, \Sigma_1^{(t)}, \Sigma_2^{(t)} \ldots \Sigma_k^{(t)}, \pi_1^{(t)}, \pi_2^{(t)} \ldots \pi_k^{(t)}\}$$

E-step

Compute label distribution of each data point

$$P\left(y_j = i \left| x_j, \theta^{(t)}\right.\right) \propto \pi_i^{(t)} N\left(x_j \left| \mu_i^{(t)}, \Sigma_i^{(t)}\right.\right)$$

M-step

Compute weighted MLE of parameters given label distributions

$$\mu_i^{(t+1)} = \frac{\sum_j P\left(y_j = i \left| x_j, \theta^{(t)}\right.\right) x_j}{\sum_{j'} P\left(y_{j'} = i \left| x_{j'}, \theta^{(t)}\right.\right)} \quad \Sigma_i^{(t+1)} = \frac{\sum_j P\left(y_j = i \left| x_j, \theta^{(t)}\right.\right) [x_j - \mu_i^{(t+1)}][x_j - \mu_i^{(t+1)}]^T}{\sum_{j'} P\left(y_{j'} = i \left| x_{j'}, \theta^{(t)}\right.\right)}$$

$$\pi_i^{(t+1)} = \frac{\sum_j P\left(y_j = i \left| x_j, \theta^{(t)}\right.\right)}{m}$$

EM degrades to k-means if we assume

- All the Gaussians are spherical and have identical weights and covariances
  - i.e., the only parameters are the means
- The label distributions computed at E-step are point-estimations
  - i.e., hard-assignments of data points to Gaussians
  - Alternatively, assume the variances are close to zero

Can be used to learn any model with hidden variables (missing data)

Alternate: EM in General

- Compute distributions over hidden variables based on current parameter values
- Compute new parameter values to maximize expected log likelihood based on distributions over hidden variables

$$\min_{\boldsymbol{y}} \frac{\boldsymbol{y}^T(D-W)\boldsymbol{y}}{\boldsymbol{y}^T D \boldsymbol{y}}, \boldsymbol{y} \in \mathcal{R}^n, \boldsymbol{y}^T D \mathbf{1} = 0$$

$$(D-W)\boldsymbol{y} = \lambda D \boldsymbol{y}$$

$$(D-W)\boldsymbol{y} = \lambda D^{\frac{1}{2}} D^{\frac{1}{2}} \boldsymbol{y}$$

$$D^{-\frac{1}{2}}(D-W)D^{-\frac{1}{2}} D^{\frac{1}{2}}\boldsymbol{y} = \lambda D^{\frac{1}{2}}\boldsymbol{y}$$

$$D^{-\frac{1}{2}}(D-W)D^{-\frac{1}{2}}\boldsymbol{z} = \lambda \boldsymbol{z}$$

$$k = \frac{\sum_{x_i>0} d_i}{\sum_i d_i}$$

$$\boldsymbol{x} \in [1,-1]^n, x_i = \begin{cases} 1 & i \in A \\ -1 & i \in B \end{cases}$$

$$\boldsymbol{y} = (\mathbf{1} + \boldsymbol{x}) - b(\mathbf{1} - \boldsymbol{x}) \qquad b = \frac{k}{1-k}$$

Output: weighted sum of the values, dependent on each query

- The weight is computed via the dot-product.

$$A(q, K, V) = \sum_i \frac{e^{q \cdot k_i}}{\sum_j e^{q \cdot k_j}} v_i$$

LayerNorm(x + Sublayer(x))

$$A(Q, K, V) = \text{softmax}\left(\frac{QK^T}{\sqrt{d_k}}\right)V \qquad output = \frac{\boldsymbol{x} - \mu}{\sqrt{\sigma + \epsilon}} \times \gamma + \beta$$

$$MultiHead(Q,K,V) = Concat(head_1, \ldots, head_h)W^O$$

where: $head_i = Attention(QW_i^Q, KW_i^K, VW_i^V)$

$$PE_{(pos,2i)} = \sin(pos/10000^{2i/d_{model}}) \qquad PE = [0, 1, 2, \ldots, T-1]$$

$$PE_{(pos,2i+1)} = \cos(pos/10000^{2i/d_{model}}) \qquad PE = [\frac{0}{T}, \frac{1}{T}, \frac{2}{T}, \ldots, \frac{T-1}{T}]$$