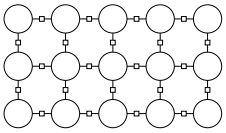
Bayesian Networks II



Lecture: Bayesian networks

Definitions: Probabilistic Programming

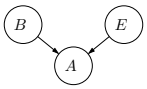
Inference: Probabilistic Inference

Inference: Forward Backward

Inference: Particle Filtering

- In this module, I will talk about probabilistic programming, a way to think about defining Bayesian networks through the lens of writing programs, which really highlights the generative process aspect of Bayesian networks.

Probabilistic programs



Joint distribution:

$$\mathbb{P}(B = b, E = e, A = a) = p(b)p(e)p(a \mid b, e)$$

Probabilistic program: alarm

$B \sim \text{Bernoulli}(\epsilon)$
$E \sim \text{Bernoulli}(\epsilon)$
$A = B \vee E$

```
def Bernoulli(epsilon):
    return random.random() < epsilon
```

Key idea: probabilistic program

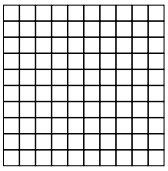
A randomized program that sets the random variables.

- Recall that a Bayesian network is given by (i) a set of random variables, (ii) directed edges between those variables capturing qualitative dependencies, (iii) local conditional distributions of each variable given its parents which captures these dependencies quantitatively, and (iv) a joint distribution which is produced by multiplying all the local conditional distributions together. Now the joint distribution is your probabilistic database, which you can answer all sorts of questions on it using probabilistic inference.
- There is another way of writing down Bayesian networks other than graphically or mathematically, and that is as a probabilistic program.
- Let's go through the alarm example. We can sample B and E independently from a Bernoulli distribution with parameter ϵ, which produces 1 (true) with probability ϵ. Then we just set $A = B \vee E$.
- In general, a **probabilistic program** is a randomized program that invokes a random number generator. Executing this program will assign values to a collection of random variables $X_1, \ldots, X_n$; that is, generating an assignment.
- We then define probability under the joint distribution of an assignment to be exactly the probability that the program generates an assignment.
- While you can run the probabilistic program to generate samples, it's important to think about it as a mathematical construct that is used to define a joint distribution.

Probabilistic program: example

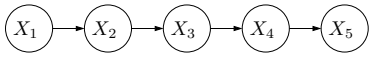
Probabilistic program: object tracking

$X_0 = (0, 0)$
For each time step $i = 1, \ldots, n$:
  if Bernoulli(α):
    $X_i = X_{i-1} + (1, 0)$ [go right]
  else:
    $X_i = X_{i-1} + (0, 1)$ [go down]



(press ctrl-enter to save)

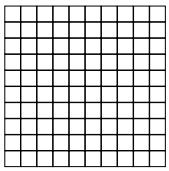
Run



- This is a more interesting example showcasing the convenience of probabilistic **programming**.
- In this program, we'll use a for loop, which allows us to compactly specify the distribution over an unboundedly large (n) set of variables.
- In the object tracking example, we define a program that generates the trajectory of an object. At each time step i, we take the previous X_{i-1} location and move it right with probability α and down with probability $1 - \alpha$, yielding X_i.
- This program is a full specification of the local conditional distribution and thus the joint distribution!
- Try clicking [Run] to run the program. Each time a new assignment of $(X_1, \ldots, X_n)$ is chosen, and recall that the probability of the program generating an assignment is the probability under the joint distribution by definition.
- We can also draw the Bayesian network, which allows us to visualize the dependencies. Here, each X_i only depends on X_{i-1}. This chain-structured Bayesian network is called a **Markov model**. However, note that the graphical representation doesn't specify the local conditional distributions.

Probabilistic inference: example

Question: what are possible trajectories given **evidence** $X_{10} = (8, 2)$?



(press ctrl-enter to save)

Run

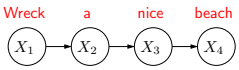
- Having used the program to define a joint distribution, we can now answer questions about that distribution.
- For example, suppose that we observe evidence $X_{10} = (8, 2)$. What is the distribution over the other variables?
- In the demo, we condition on the evidence and observe the distribution over all trajectories, which are constrained to go through $(8, 2)$ at time step 10.

Application: language modeling

Can be used to score sentences for speech recognition or machine translation

Probabilistic program: Markov model

For each position $i = 1, 2, \ldots, n$:
  Generate word $X_i \sim p(X_i \mid X_{i-1})$

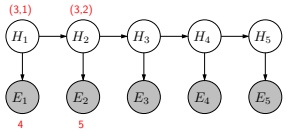


- Now I'm going to quickly go through a set of examples of Bayesian networks or probabilistic programs and talk about the applications they are used for.
- A natural language sentence can be viewed as a sequence of words, and a language model assigns a probability to each sentence, which measures the "goodness" of that sentence.
- Markov models and higher-order Markov models (called n-gram models in NLP), were the dominant paradigm for language modeling before deep learning, and for a while, they outperformed neural language models since they were computationally much easier to scale up.
- While they could be used to generate text unconditionally, they were often used in the context of a speech recognition or machine translation system to score the fluency of the output.
- A Markov model generates each word given the previous word according to some local conditional distribution $p(X_i \mid X_{i-1})$ which we're not specifying right now.

Application: object tracking

Probabilistic program: hidden Markov model (HMM)

For each time step $t = 1, \ldots, T$:
  Generate object location $H_t \sim p(H_t \mid H_{t-1})$
  Generate sensor reading $E_t \sim p(E_t \mid H_t)$



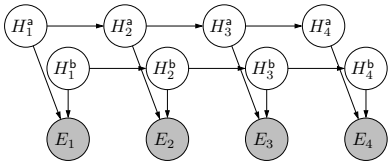
Inference: given sensor readings, where is the object?

- Markov models are limiting because they do not have a way of talking about noisy evidence (sensor readings). They can be extended to hidden Markov models, which introduce a parallel sequence of observation variables.
- For example, in object tracking, H_t denotes the true object location, and E_t denotes the noisy sensor reading, which might be (i) the location H_t plus noise, or (ii) the distance from H_t plus noise, depending on the type of sensor.
- In speech recognition, H_t would be the phonemes or words and E_t would be the raw acoustic signal.

Application: multiple object tracking

Probabilistic program: factorial HMM

For each time step $t = 1, \ldots, T$:
  For each object $o \in \{\mathsf{a}, \mathsf{b}\}$:
    Generate location $H_t^o \sim p(H_t^o \mid H_{t-1}^o)$
  Generate sensor reading $E_t \sim p(E_t \mid H_t^{\mathsf{a}}, H_t^{\mathsf{b}})$

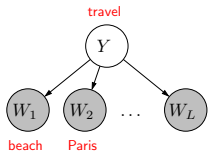


- An extension of an HMM, called a **factorial HMM**, can be used to track multiple objects.
- We assume that each object moves independently according to a Markov model, but that we get one sensor reading which is some noisy aggregated function of the true positions.
- For example, E_t could be the set $\{H_t^{\mathsf{a}}, H_t^{\mathsf{b}}\}$, which reveals where the objects are, but doesn't say which object is responsible for which element in the set.

Application: document classification

Probabilistic program: naive Bayes

Generate label $Y \sim p(Y)$
For each position $i = 1, \ldots, L$:
  Generate word $W_i \sim p(W_i \mid Y)$



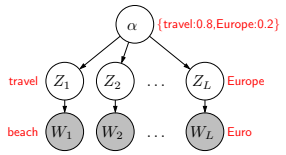
Inference: given a text document, what is it about?

- Naive Bayes is a very simple model which is often used for classification. For document classification, we generate a label and all the words in the document given that label.
- Note that the words are all generated independently, which is not a very realistic model of language, but naive Bayes models are surprisingly effective for tasks such as document classification.
- These types of models are traditionally called generative models as opposed to discriminative models for classification. Rather than thinking about how you take the input and produce the output label (e.g., using a neural network), you go the other way around: think about how the input is generated from the output (which is usually the purer, more structured form of the input).
- One advantage of using Naive Bayes for classification is that "training" is extremely easy and fast and just requires counting (as opposed to performing gradient descent).

Application: topic modeling

Probabilistic program: latent Dirichlet allocation

Generate a distribution over topics $\alpha \in \mathbb{R}^K$
For each position $i = 1, \ldots, L$:
  Generate a topic $Z_i \sim p(Z_i \mid \alpha)$
  Generate a word $W_i \sim p(W_i \mid Z_i)$



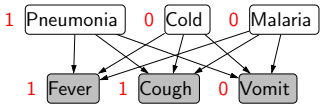
Inference: given a text document, what topics is it about?

- A more sophisticated model of text is Latent Dirichlet Allocation (LDA), which allows a document to not just be about one topic or class (which was true in naive Bayes), but about multiple topics.
- Here, the distribution over topics α is chosen per document from a Dirichlet distribution. Note that α is a continuous-valued random variable. For each position, we choose a topic according to that per-document distribution and generate a word given that topic.
- Latent Dirichlet Allocation (LDA) has been very influential for modeling not only text but images, videos, music, etc.; any sort of data with hidden structure. It is closely related to matrix factorization.

Application: medical diagnosis

Probabilistic program: diseases and symptoms

For each disease $i = 1, \ldots, m$:
  Generate activity of disease $D_i \sim p(D_i)$
For each symptom $j = 1, \ldots, n$:
  Generate activity of symptom $S_j \sim p(S_j \mid D_{1:m})$



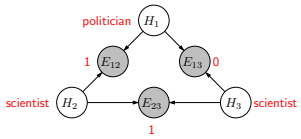
Inference: If a patient has some symptoms, what diseases do they have?

- We already saw a special case of this model. In general, we would like to diagnose many diseases and might have measured many symptoms and vitals.

Application: social network analysis

Probabilistic program: stochastic block model

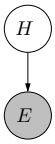
For each person $i = 1, \ldots, n$:
  Generate person type $H_i \sim p(H_i)$
For each pair of people $i \neq j$:
  Generate connectedness $E_{ij} \sim p(E_{ij} \mid H_i, H_j)$



Inference: Given a social network graph, what types of people are there?

- One can also model graphs such as social networks. A very naive-Bayes-like model is that each node (person) has a "type". Whether two people interact with each other (there is an edge between the two people) is determined solely by their types and random chance.
- Note: there are extensions called mixed membership models which, like LDA, allow each person to have multiple types.

Summary



- Probabilistic program specifies a Bayesian network
- Many different types of models
- Common paradigm: come up with stories of how the quantities of interest (output) generate the data (input)
- Opposite of how we normally do classification!

- In summary, we've seen how we can define Bayesian networks (and therefore joint distributions) by writing down probabilistic programs.
- Using this powerful tool, we then did a whirlwind tour of lots of probabilistic programs that exist in the literature (though not often introduced under this general framework).
- The common theme of these probabilistic programs is that each attempts to produce **stories** of how certain quantities of interest H (e.g., actual location of an object) generate (or give rise to) observations E (e.g., usually noisy versions).
- After defining such a model, one can do probabilistic inference to compute $\mathbb{P}(H \mid E = e)$. Note that we can see how Bayesian networks allow us to handle heterogeneous inputs (e.g., missing information). We can simply condition on partial evidence.
- Bayesian networks therefore provide quite a different paradigm compared to normal classification (e.g., neural networks). You have to think about going from the output to the input rather than input to output, which takes some getting used to.

Lecture: Bayesian networks

Definitions: Probabilistic Programming

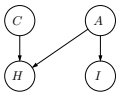
Inference: Probabilistic Inference

Inference: Forward Backward

Inference: Particle Filtering

- In this module, I will talk about a strategy for performing probabilistic inference in general Bayesian networks.

Review: Bayesian network



Random variables:

cold C, allergies A, cough H, itchy eyes I

Joint distribution:

$$\mathbb{P}(C = c, A = a, H = h, I = i) = p(c)p(a)p(h \mid c, a)p(i \mid a)$$

Definition: Bayesian network

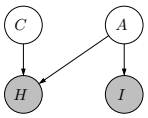
Let $X = (X_1, \ldots, X_n)$ be random variables.

A **Bayesian network** is a directed acyclic graph (DAG) that specifies a joint distribution over X as a product of local conditional distributions, one for each node:

$$\mathbb{P}(X_1 = x_1, \ldots, X_n = x_n) \stackrel{\text{def}}{=} \prod_{i=1}^{n} p(x_i \mid x_{\text{Parents}(i)})$$

- Recall that a Bayesian network is given by (i) a set of random variables, (ii) directed edges between those variables capturing qualitative dependencies, (iii) local conditional distributions of each variable given its parents which captures these dependencies quantitatively, and (iv) a joint distribution which is produced by multiplying all the local conditional distributions together.

Review: probabilistic inference



Question: $\mathbb{P}(C \mid H = 1, I = 1)$

Input

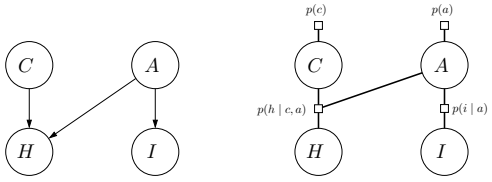
Bayesian network: $\mathbb{P}(X_1, \ldots, X_n)$
Evidence: $E = e$ where $E \subseteq X$ is subset of variables
Query: $Q \subseteq X$ is subset of variables

Output

$\mathbb{P}(Q \mid E = e)$ ⟷ $\mathbb{P}(Q = q \mid E = e)$ for all values q

- Given the joint distribution representing your probabilistic database, you can answer all sorts of questions on it using probabilistic inference.
- Given a set of evidence variables and values, a set of query variables, we want to compute the probability of the query variables given the evidence, marginalizing out all other variables.

Reduction to Markov networks



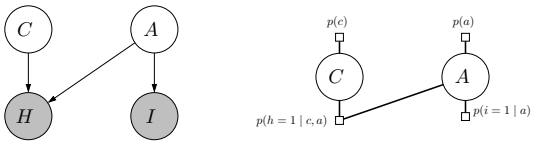
$$\mathbb{P}(C = c, A = a, H = h, I = i) = \frac{1}{Z} p(c)p(a)p(h \mid c, a)p(i \mid a)$$

Bayesian network = Markov network with normalization constant $Z = 1$

Reminder: single factor that connects **all** parents!

- Our overarching strategy for performing inference in Bayesian networks is to convert them into Markov networks.
- Recall that the joint distribution is just the product of all the local conditional distributions. The local conditional distributions (e.g., $p(a \mid b, e)$) are all non-negative so they can be interpreted as simply factors in a factor graph.
- Recall that a Markov network defines the joint distribution as the product of all the factors divided by some normalization constant Z. But in this case, $Z = 1$ because the factors are local conditional distributions of a Bayesian network! Put it another way, Bayesian networks are just instances of Markov networks where the normalization constant $Z = 1$.
- It's important to remember that there is a single factor that connects all the parents. Don't let the directed graph in the Bayesian network deceive you into thinking that there are two factors, one per arrow, which is a common mistake.
- Now we can run any inference algorithm for Markov networks (e.g., Gibbs sampling) on this so-called Markov network and obtain quantities such as $\mathbb{P}(H = 1)$. But there is one important thing that's missing, which is the ability to condition on evidence...

Conditioning on evidence



Markov network:

$$\mathbb{P}(C = c, A = a \mid H = 1, I = 1) = \frac{1}{Z} p(c)p(a)p(h = 1 \mid c, a)p(i = 1 \mid a)$$

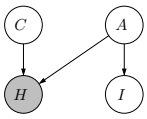
Bayesian network with evidence = Markov network with $Z = \mathbb{P}(H = 1, I = 1)$

Solution: run any inference algorithm for Markov networks (e.g., Gibbs sampling)!

[demo]

- Suppose we condition on evidence $H = 1$ and $I = 1$.
- We can define a new Markov network over the remaining variables (C and A) by simply plugging in the values to H and I. The normalization constant Z is the sum over all values of C and A, which is no longer 1, but rather the probability of the evidence $\mathbb{P}(H = 1, I = 1)$.
- To understand why this relationship holds, recall that the desired conditional probability is the joint probability over the marginal probability. The factors simply represent the joint probability, and thus the normalization constant must be the marginal probability.
- Now we can again run any inference algorithm for Markov networks (e.g., Gibbs sampling), and this allows us to do probabilistic inference in any Bayesian network.
- In the demo, we will run Gibbs sampling to compute $\mathbb{P}(C = 1 \mid H = 1, I = 1)$, and we see that it converges to the right answer (0.13).

Leveraging additional structure: unobserved leaves



Markov network:

$$\mathbb{P}(C = c, A = a, I = i \mid H = 1) = \frac{1}{Z} p(c)p(a)p(h = 1 \mid c, a)p(i \mid a),$$

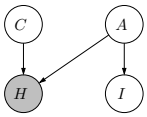
where $Z = \mathbb{P}(H = 1)$

Question: $\mathbb{P}(C = 1 \mid H = 1)$

Can we reduce the Markov network before running inference?

- We could stop there, but there are two more ways we can leverage the structure of Bayesian networks to optimize things a bit.
- Suppose we are now just conditioning on $H = 1$. As before we can form a Markov network over the remaining variables.
- But what if we knew we were only interested in $\mathbb{P}(C = 1 \mid H = 1)$?
- Is there a way to reduce the size of the Markov network before running inference?

Leveraging additional structure: unobserved leaves



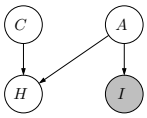
Markov network:

$$\begin{aligned}
\mathbb{P}(C = c, A = a \mid H = 1) &= \textstyle\sum_i \mathbb{P}(C = c, A = a, I = i \mid H = 1) \\
&= \textstyle\sum_i \frac{1}{Z} p(c)p(a)p(h = 1 \mid c, a)p(i \mid a) \\
&= \frac{1}{Z} p(c)p(a)p(h = 1 \mid c, a) \textstyle\sum_i p(i \mid a) \\
&= \frac{1}{Z} p(c)p(a)p(h = 1 \mid c, a)
\end{aligned}$$

Throw away any unobserved leaves before running inference!

- The answer is yes.
- Let us try marginalizing out I. We expand using the definition of marginal probability, definition of the Bayesian network, pushing the $\sum_i$ inwards past factors that don't depend on i, and noting that $\sum_i p(i \mid a) = 1$ by definition of local conditional distributions.
- But if we stare at the last equation, it is what we would have gotten if we had just ignored I in the first place!
- The general principle here is that marginalization of any unobserved leaf node produces 1, and thus all such nodes can be simply ignored. And we can keep on iterating this until all leaves are observed.
- This is practically very useful because it means that whenever we have a large Bayesian network, we might be able to remove large swaths of the network.
- This property establishes a bridge between marginalization (algebraic operations, usually involves hard work) with removal (graph operations, usually more intuitive).

Leveraging additional structure: independence



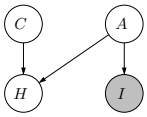
Markov network:

$$\begin{aligned}
\mathbb{P}(C = c \mid I = 1) &= \textstyle\sum_{a,h} \mathbb{P}(C = c, A = a, H = h \mid I = 1) \\
&= \textstyle\sum_{a,h} \frac{1}{Z} p(c)p(a)p(h \mid c, a)p(i = 1 \mid a) \\
&= \textstyle\sum_a \frac{1}{Z} p(c)p(a)p(i = 1 \mid a) \\
&= p(c) \textstyle\sum_a \frac{1}{Z} p(a)p(i = 1 \mid a) \\
&= p(c)
\end{aligned}$$

Throw away any disconnected components before running inference!

- There is another type of structure we can exploit, which is not specific to Bayesian networks, but shows up generally in Markov networks.
- Suppose we now condition on $I = 1$. Let us expand the marginal probability into the joint probability, expand into the local conditional probabilities, marginalize out the unobserved leaf H using the same idea we just discussed,
- Now at this point, C is completely disconnected from A and I. Algebraically, we can pull $p(c)$ out of the expression.
- We have this mess involving a and i, but this quantity does not depend on c so it is a constant. In this case, we know this constant must be 1 because both $p(c)$ and the LHS are probability distributions.
- So we can throw away any disconnected components. Note that it is advantageous to do this after removing all unobserved leaves, because removing those leaves can help disconnect the graph, as it did in this example.
- Now we have a Markov network, and we would run a standard inference algorithm on it. But in this case, it only has one factor which is already a local probability distribution, so we're done.

Summary



- Condition on evidence (e.g., $I = 1$)
- Throw away unobserved leaves, e.g., I for $\mathbb{P}(C = 1 \mid H = 1)$
- Discard disconnected components, e.g., A and I for $\mathbb{P}(C = c \mid I = 1))$
- Define Markov network out of remaining factors
- Run your favorite inference algorithm (e.g., manual, Gibbs sampling)

- In summary, we tackled the problem of how to perform probabilistic inference in Bayesian networks, by reducing the problem to that of inference in Markov networks.
- To prepare the Markov network, we condition on the evidence (substitute the values into the factors), throw away any unobserved leaves, and throw away any disconnected components.
- Then we just define the Markov network over the remaining factors. If the resulting Markov network is small enough, we can do inference manually. Otherwise, we can run an algorithm like Gibbs sampling.

Lecture: Bayesian networks

Definitions: Probabilistic Programming

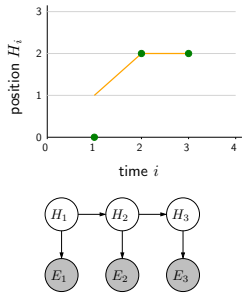
Inference: Probabilistic Inference

Inference: Forward Backward

Inference: Particle Filtering

- In this module, I will introduce the forward-backward algorithm for performing efficient and exact inference in Hidden Markov models, an important special case of Bayesian networks.

Hidden Markov models for object tracking



start

H_1

- 1/3
- 1/3
- 1/3

h_1	$p(h_1)$
0	1/3
1	1/3
2	1/3

transition

H_i

H_{i-1}

- 1/4
- 1/2
- 1/4

h_i	$p(h_i \mid h_{i-1})$
$h_{i-1} - 1$	1/4
h_{i-1}	1/2
$h_{i-1} + 1$	1/4

emission

E_i

H_i

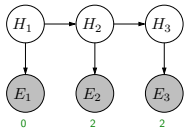
- 1/4
- 1/2
- 1/4

e_i	$p(e_i \mid h_i)$
$h_i - 1$	1/4
h_i	1/2
$h_i + 1$	1/4

$$\mathbb{P}(H = h, E = e) = \underbrace{p(h_1)}_{\text{start}} \prod_{i=2}^{n} \underbrace{p(h_i \mid h_{i-1})}_{\text{transition}} \prod_{i=1}^{n} \underbrace{p(e_i \mid h_i)}_{\text{emission}}$$

- Let us revisit our object tracking example, now through the lens of HMMs. Recall that each time i, an object is at a location H_i, and what we observe is a noisy observation E_i. The goal is to infer where the object is / was.
- We define a probabilistic story as follows: An object starts at H_1 uniformly drawn over all possible locations.
- Then at each subsequent time step, the object **transitions** from the previous time step, keeping the same location with $1/2$ probability, and moves to an adjacent location each with $1/4$ probability. For example, if $p(h_3 = 3 \mid h_2 = 3) = 1/2$ and $p(h_3 = 2 \mid h_2 = 3) = 1/4$.
- At each time step, we also **emit** a sensor reading E_i given the actual location H_i, following the same process as transitions ($1/2$ probability of the same location, $1/4$ probability of an adjacent location).
- Recall that finally, we define a joint distribution over all the actual locations $H_1, \ldots, H_n$ and sensor readings $E_1, \ldots, E_n$ by taking the product of all the local conditional probabilities.

Inference questions



Question (**filtering**):

$\mathbb{P}(H_2 \mid E_1 = 0, E_2 = 2)$

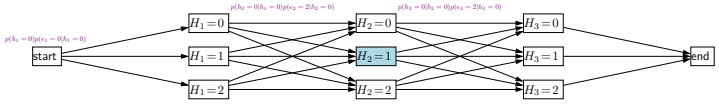
Question (**smoothing**):

$\mathbb{P}(H_2 \mid E_1 = 0, E_2 = 2, E_3 = 2)$

Note: filtering is a special case of smoothing if marginalize unobserved leaves

- In principle, you could ask any type of questions on an HMM, but there are two common ones: filtering and smoothing.
- **Filtering** asks for the distribution of some hidden variable H_i conditioned on only the evidence up until that point. This is useful when you're doing real-time object tracking, and you can't see the future.
- **Smoothing** asks for the distribution of some hidden variable H_i conditioned on all the evidence, including the future. This is useful when you have collected all the data and want to retrospectively go and figure out what H_i was.
- Note that filtering is a special case of smoothing: if we're asking for H_i given $E_1, \ldots, E_i$, then we can marginalize everything in the future (since they are just unobserved leaf nodes), reducing the problem to a smaller HMM, where we are smoothing.

Lattice representation

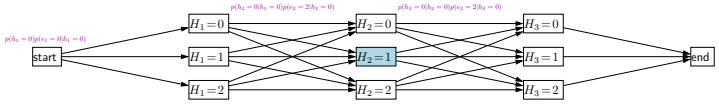


- Edge $\boxed{\text{start}} \Rightarrow \boxed{H_1 = h_1}$ has weight $p(h_1)p(e_1 \mid h_1)$
- Edge $\boxed{H_{i-1} = h_{i-1}} \Rightarrow \boxed{H_i = h_i}$ has weight $p(h_i \mid h_{i-1})p(e_i \mid h_i)$
- Each path from $\boxed{\text{start}}$ to $\boxed{\text{end}}$ is an assignment with weight equal to the product of edge weights

Key: $\mathbb{P}(H_i = h_i \mid E = e)$ is the weighted fraction of paths through $\boxed{H_i = h_i}$

- The forward-backward algorithm is based on a form of dynamic programming.
- To develop this, we consider a **lattice representation** of HMMs. Consider a directed graph (not to be confused with the HMM) with a start node, an end node, and a node for each assignment of a value to a variable $H_i = v$. The nodes are arranged in a lattice, where each column corresponds to one variable H_i and each row corresponds to a particular value v. Each path from the start to the end corresponds exactly to a complete assignment to the nodes.
- Each edge has a weight (a single number) determined by the local conditional probabilities (more generally, the factors in a factor graph). For each edge into $\boxed{H_i = h_i}$, we multiply by the transition probability into h_i and emission probability $p(e_i \mid h_i)$.
- This defines a weight for each path (assignment) in the graph equal to the joint probability $P(H = h, E = e)$.
- Note that the lattice contains $O(n|\text{Domain}|)$ nodes and $O(n|\text{Domain}|^2)$ edges, where n is the number of variables and |Domain| is the number of values in the domain of each variable (3 in our example).
- Now comes the key point. Recall we want to compute a smoothing question $\mathbb{P}(H_i = h_i \mid E = e)$. This quantity is simply the weighted fraction of paths that pass through $\boxed{H_i = h_i}$. This is just a way of visualizing the definition of the smoothing question.
- There are an exponential number of paths, so it's intractable to enumerate all of them. But we can use dynamic programming...

Forward and backward messages



Forward: $F_i(h_i) = \sum_{h_{i-1}} F_{i-1}(h_{i-1})\text{Weight}(\boxed{H_{i-1} = h_{i-1}}, \boxed{H_i = h_i})$

sum of weights of paths from $\boxed{\text{start}}$ to $\boxed{H_i = h_i}$

Backward: $B_i(h_i) = \sum_{h_{i+1}} B_{i+1}(h_{i+1})\text{Weight}(\boxed{H_i = h_i}, \boxed{H_{i+1} = h_{i+1}})$

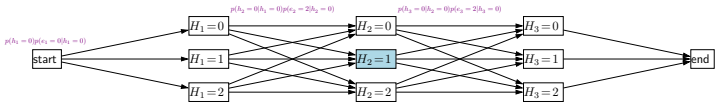
sum of weights of paths from $\boxed{H_i = h_i}$ to $\boxed{\text{end}}$

Define $S_i(h_i) = F_i(h_i)B_i(h_i)$:

sum of weights of paths from $\boxed{\text{start}}$ to $\boxed{\text{end}}$ through $\boxed{H_i = h_i}$

- First, define the forward message $F_i(v)$ to be the sum of the weights over all paths from the start node to $\boxed{H_i = v}$. This can be defined recursively: any path that goes $\boxed{H_i = h_i}$ will have to go through some $\boxed{H_{i-1} = h_{i-1}}$, so we can sum over all possible values of h_{i-1}.
- Analogously, let the backward message $B_i(v)$ be the sum of the weights over all paths from $\boxed{H_i = v}$ to the end node.
- Finally, define $S_i(v)$ to be the sum of the weights over all paths from the start node to the end node that pass through the intermediate node $\boxed{X_i = v}$. This quantity is just the product of the weights of paths going into $\boxed{H_i = h_i}$ ($F_i(h_i)$) and those leaving it ($B_i(h_i)$).
- This is analogous to factoring: $(a + b)(c + d) = ab + ad + bc + bd$.
- Note: $F_1(h_1) = p(h_1)p(e_1 = 0 \mid h_1)$ and $B_n(h_n) = 1$ are base cases, which don't require the recurrence.

Putting everything together



$$\mathbb{P}(H_i = h_i \mid E = e) = \frac{S_i(h_i)}{\sum_v S_i(v)}$$

Algorithm: forward-backward algorithm

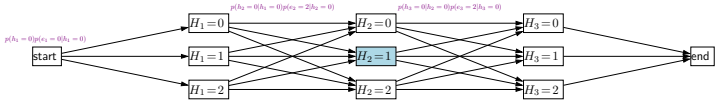
Compute $F_1, F_2, \ldots, F_n$
Compute $B_n, B_{n-1}, \ldots, B_1$
Compute S_i for each i and normalize

Running time: $O(n|\text{Domain}|^2)$

[demo]

- Now the smoothing question $\mathbb{P}(H_i = h_i \mid E = e)$ is just equal to the normalized version of S_i.
- The algorithm is thus as follows: for each node $\boxed{H_i = h_i}$, we compute three numbers: $F_i(h_i), B_i(h_i), S_i(h_i)$. First, we sweep forward to compute all the F_i's recursively. At the same time, we sweep backward to compute all the B_i's recursively. Then we compute S_i by pointwise multiplication.
- The running time of the algorithm is $O(n|\text{Domain}|^2)$, which is the number of edges in the lattice.
- In the demo, we are running the variable elimination algorithm, which is a generalization of the forward-backward algorithm for arbitrary Markov networks. As you step through the algorithm, you can see that the algorithm first computes a forward message F_2 and then a backward message B_2, and then it multiplies everything together and normalizes to produce $\mathbb{P}(H_2 \mid E_1 = 0, E_2 = 2, E_3 = 2)$. The names and details don't match up exactly, so you don't need to look too closely.
- Implementation note: we technically can normalize S_i to get $\mathbb{P}(H_i \mid E = e)$ at the very end but it's useful to normalize F_i and B_i at each step to avoid underflow. In addition, normalization of the forward messages yields $\mathbb{P}(H_i = v \mid E_1 = e_1, \ldots, E_i = e_i)$ which are exactly the filtering queries!

Summary



- Lattice representation: paths are assignments
- Dynamic programming: compute sums efficiently
- Forward-backward algorithm: compute all smoothing questions, share intermediate computations

- In summary, we have presented the forward-backward algorithm for probabilistic inference in HMMs, in particular smoothing queries.
- The algorithm is based on the lattice representation in which each path is an assignment, and the weight of path is the joint probability.
- Smoothing is just then asking for the weighted fraction of paths that pass through a given node.
- Dynamic programming can be used to compute this quantity efficiently.
- This is formalized using the forward-backward algorithm, which consists of two sets of recurrences.
- Note that the forward-backward algorithm gives you the answer to all the smoothing questions ($\mathbb{P}(H_i = h_i \mid E = e)$ for all i), because the intermediate computations are all shared.

Lecture: Bayesian networks

Definitions: Probabilistic Programming

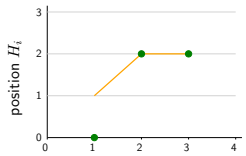
Inference: Probabilistic Inference

Inference: Forward Backward

Inference: Particle Filtering

- In this module, I will present the particle filtering algorithm for performing approximate inference in Hidden Markov models which is useful when the size of the domain of the variables is large.

Review: Hidden Markov models for object tracking



start

H_1

h_1	$p(h_1)$
0	1/3
1	1/3
2	1/3

transition

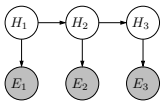
H_i

h_i	$p(h_i \mid h_{i-1})$
$h_{i-1}-1$	1/4
h_{i-1}	1/2
$h_{i-1}+1$	1/4

emission

E_i

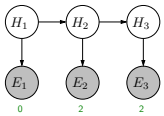
e_i	$p(e_i \mid h_i)$
h_i-1	1/4
h_i	1/2
h_i+1	1/4



$$\mathbb{P}(H=h, E=e) = \underbrace{p(h_1)}_{\text{start}} \prod_{i=2}^{n} \underbrace{p(h_i \mid h_{i-1})}_{\text{transition}} \prod_{i=1}^{n} \underbrace{p(e_i \mid h_i)}_{\text{emission}}$$

- Recall that HMM for object tracking.
- Each each point in time, an object has an position H_i, which gives rise to a sensor reading E_i. We start with H_1 uniform over positions, transition from H_{i-1} to H_i with $1/2$ probability on the same location and $1/4$ probability on an adjacent location. We emit the sensor reading analogously. Multiply everything together to form the joint distribution over locations $H_1, \ldots, H_n$ and sensor readings $E_1, \ldots, E_n$.

Review: inference in Hidden Markov models



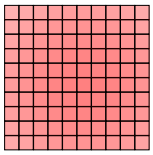
Filtering questions:

$\mathbb{P}(H_1 \mid E_1 = 0)$

$\mathbb{P}(H_2 \mid E_1 = 0, E_2 = 2)$

$\mathbb{P}(H_3 \mid E_1 = 0, E_2 = 2, E_3 = 2)$

Problem: many possible location values for H_i

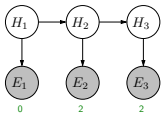


Forward-backward is too slow ($O(n|\text{Domain}|^2)$)...

- Recall that the two common types of inference questions we ask on HMMs are filtering and smoothing.
- Particle filtering, as the name might suggest, performs filtering, so let us focus on that. Filtering asks for the probability distribution over object location H_i at a current time step i given the past observations $E_1 = e_1, \ldots, E_i = e_i$.
- Last time, we saw that the forward-backward algorithm could already solve this. But it runs in $O(n|\text{Domain}|^2)$, where $|\text{Domain}|$ is the number of possible values (e.g., locations) that H_i can take on. On this example, $H_i \in \{0, 1, 2\}$ but for real applications, there could easily be hundreds of thousands of values, not to mention what happens if H_i is continuous. This could be a very large number, which makes the forward-backward algorithm very slow (even if it's not exponentially so).
- The motivation of particle filtering is to perform **approximate probabilistic inference**, and leverages the fact that most of the locations are very improbable given evidence.
- Particle filtering actually applies to general factor graphs, but we will present them for hidden Markov models for concreteness.

Beam search for HMMs

Idea: keep $\leq K$ partial assignments (**particles**)



Algorithm: beam search

Initialize $C \leftarrow [\{\}]$

For each $i = 1, \ldots, n$:

Extend:

$C' \leftarrow \{h \cup \{H_i : v\} : h \in C, v \in \text{Domain}_i\}$

Prune:

$C \leftarrow K$ particles of C' with highest weights

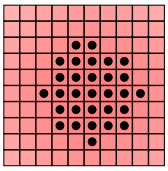
Normalize weights to get approximate $\hat{\mathbb{P}}(H_1, \ldots, H_n \mid E = e)$

Sum probabilities to get any approximate $\hat{\mathbb{P}}(H_i \mid E = e)$

[demo: `beamSearch({K:3})`]

- Our starting point for motivating particle filtering is beam search, an algorithm for finding an approximate maximum weight assignment in arbitrary constraint satisfaction problems (CSPs).
- Since HMMs are Bayesian networks, which are Markov networks, which have an underlying factor graph, we can simply apply beam search to HMMs (for now putting aside the goal of finding the maximum weight assignment).
- Recall that beam search maintains a list of candidate partial assignments to the first i variables. There are two phases. In the first phase, we **extend** all the existing candidates C to all possible assignments to H_i; this results in $K = |\text{Domain}|$ candidates C'. We then take the subset of K candidates with the highest weight, where the weight of a partial assignment is simply the product of all the factors (transitions, emissions) that can be computed on the partial assignment.
- In the demo, we start with partial assignments to H_1, whose weights are given by $p(h_1)p(e_1 = 0 \mid h_1)$. In the next step, we can multiply in factors $p(h_2 \mid h_1)p(e_2 = 2 \mid h_2)$, and so on.
- At the very end, we obtain $K = 3$ complete assignments, each with a weight (equal to the joint probability of the assignment and observations). We can normalize these weights to form an approximate distribution over all assignments (conditioned on the observations). From here, we can manually compute any marginal probabilities (e.g., $\mathbb{P}(H_3 = 2 \mid E = e)$) by summing the probabilities of assignments satisfying the given condition (e.g., $H_3 = 2$).

Beam search problems



Algorithm: beam search

Initialize $C \leftarrow [\{\}]$
For each $i = 1, \ldots, n$:
  Extend:
    $C' \leftarrow \{h \cup \{H_i : v\} : h \in C, v \in \text{Domain}_i\}$
  Prune:
    $C \leftarrow K$ particles of C' with highest weights

- Extend: slow because requires considering every possible value for H_i
- Prune: greedily taking best K doesn't provide diversity

Particle filtering solution (3 steps): **propose, weight, resample**

- There are two problems with beam search.
- First, beam search can be slow if Domain is large, since we might have to try every single candidate value h_i to assign H_i. In some cases, we can efficiently generate only the values h_i that have nonzero transition probability ($p(h_i \mid h_{i-1} > 0)$, for example, if we know that h_i must be within a certain distance of h_{i-1} (can't teleport). But if we wanted to track the object to high resolution, there might still be too many values to consider.
- Second, beam search greedily takes the K highest weight candidates at each time step. This could be dangerous, since we might end up with many assignments that are only slightly different, and not truly representative of the actual distribution. You can think of this as a form of overfitting.
- Particle filtering addresses both of these problems. It has three steps: propose, which extends the current partial assignment, and reweight + resample, which redistributes resources on the particles based on evidence.

Step 1: propose

Old particles: $\approx \mathbb{P}(H_1, H_2 \mid E_1 = 0, E_2 = 2)$

$\{H_1 : 0, H_2 : 1\}$
$\{H_1 : 1, H_2 : 2\}$

Key idea: proposal distribution

For each old particle (h_1, h_2), sample $H_3 \sim p(h_3 \mid h_2)$.

h_i	$p(h_i \mid h_{i-1})$
$h_{i-1} - 1$	1/4
h_{i-1}	1/2
$h_{i-1} + 1$	1/4

New particles: $\approx \mathbb{P}(H_1, H_2, H_3 \mid E_1 = 0, E_2 = 2)$

$\{H_1 : 0, H_2 : 1, H_3 : 1\}$
$\{H_1 : 1, H_2 : 2, H_3 : 2\}$

- At each stage of the particle filtering, we can think of our set of particles C as approximating a certain distribution.
- Suppose we have a set of particles that approximates the filtering distribution over H_1, H_2. The first step is to extend each current partial assignment (particle) from $(h_1, \ldots, h_{i-1})$ to $(h_1, \ldots, h_i)$.
- To do this, we simply go through each particle and sample a new value h_i using the transition probability $p(h_i \mid h_{i-1})$.
- We can think of advancing each particle according to the dynamics of the HMM. These extended particles approximate the probability of H_1, H_2, H_3, but still conditioned on the same evidence.
- In some cases (e.g., the transitions are Gaussian), sampling h_3 is very easy compared to enumerating all possible of h_3. (Indeed, the advantages of particle filtering are clearer in continuous state spaces.).

Step 2: weight

Old particles: $\approx \mathbb{P}(H_1, H_2, H_3 \mid E_1 = 0, E_2 = 1)$

$\{H_1 : 0, H_2 : 1 : H_3 : 1\}$
$\{H_1 : 1, H_2 : 2 : H_3 : 2\}$

Key idea: weighting based on evidence

For each old particle (h_1, h_2, h_3), weight it by $p(e_3 = 2 \mid h_3)$.

h_3	$p(e_3 = 2 \mid h_3)$
0	0
1	1/4
2	1/2

New particles: $\approx \mathbb{P}(H_1, H_2, H_3 \mid E_1 = 0, E_2 = 1, E_3 = 2)$

$\{H_1 : 0, H_2 : 1 : H_3 : 1\}$ (1/4)
$\{H_1 : 1, H_2 : 2 : H_3 : 2\}$ (1/2)

- Having generated a set of K candidates, we need to now take into account the new evidence $E_i = e_i$. This is a deterministic step that simply weights each particle by the probability of generating $E_i = e_i$, which is the emission probability $p(e_i \mid h_i)$.
- Intuitively, the proposal was just a guess about where the object will be H_3, but we need to fact check this guess.
- In this example, we observed $E_3 = 2$, so we need to weight the two particles by $p(e_3 = 2 \mid h_3 = 1) = 1/4$ and $p(e_3 = 2 \mid h_3 = 2) = 1/2$, respectively.

Step 3: resample

Old particles: $\approx \mathbb{P}(H_1, H_2, H_3 \mid E_1 = 0, E_2 = 2, E_3 = 2)$

$\{H_1 : 0, H_2 : 1 : H_3 : 1\}$ (1/4) $\Rightarrow$ 1/3
$\{H_1 : 1, H_2 : 2 : H_3 : 2\}$ (1/2) $\Rightarrow$ 2/3

Key idea: resampling

Normalize weights and draw K samples to redistribute particles to more promising areas.

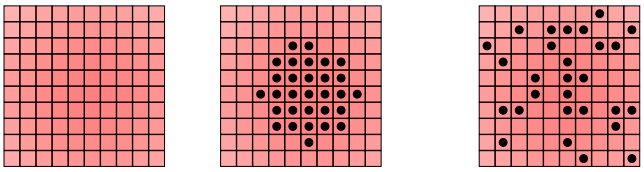
New particles: $\approx \mathbb{P}(H_1, H_2, H_3 \mid E_1 = 0, E_2 = 2, E_3 = 2)$

$\{H_1 : 1, H_2 : 2 : H_3 : 2\}$
$\{H_1 : 1, H_2 : 2 : H_3 : 2\}$

- At this point, we have a set of weighted particles representing the desired filtering distribution.
- However, if some of the weights are small, this could be wasteful. In the extreme case, any particle with zero weight should just be thrown out.
- The K particles can be viewed as our limited resources for representing the distribution, the resampling step attempts to redistribute these precious resources to places in the distribution that are more promising.
- To this end, we will normalize the weights to form a distribution over the particles (similar to what we did at the end of beam search). Then we sample K times from this distribution.
- In this example, we happened to get two occurrences of the second particle, but we might have easily gotten one of each or even two of the first.

Why sampling?

distribution | K with highest weight | K sampled from distribution



not representative | more representative

Sampling is especially important when there is high uncertainty!

- You might wonder why we are resampling, leaving the result of the algorithm up to chance.
- To see why resampling can be more favorable than beam search, consider the setting where we start with a set of particles on the left where the weights are given by the shade of red (darker is more weight). Notice that the weights are all quite similar (i.e., the distribution is close to the uniform distribution).
- Beam search chooses the K locations with the highest weight, which would clump all the particles near the mode. This is risky, because we have no support out farther from the center, where there is actually substantial probability.
- However, if we sample from the distribution which is proportional to the weights, then we can hedge our bets and get a more representative set of particles which cover the space more evenly.
- In cases where the original weights much more skewed towards a few particles, then taking the highest weight particles is fine and perhaps even slightly better than resampling.

Particle filtering

Algorithm: particle filtering

Initialize $C \leftarrow [\{\}]$
For each $i = 1, \ldots, n$:
  Propose:
    $C' \leftarrow \{h \cup \{H_i : h_i\} : h \in C, h_i \sim p(h_i \mid h_{i-1})\}$
  Weight:
    Compute weights $w(h) = p(e_i \mid h_i)$ for $h \in C'$
  Resample:
    $C \leftarrow K$ particles drawn independently from $\frac{w(h)}{\sum_{h' \in C} w(h')}$

[demo: `particleFiltering({K:100})`]

- We now present the final particle filtering algorithm, which is structurally similar to beam search. We go through all the variables $H_1, \ldots, H_n$.
- For each candidate $h \in C$, we propose h_i according to the transition distribution $p(h_i \mid h_{i-1})$.
- We then weight this particle using the emission probability $w(h) = p(e_i \mid h_i)$.
- Finally, we normalize the weights $\{w(h) : h \in C\}$ and sample K particles independently from this distribution.
- In the demo, we can go through the extend (propose) and prune (weight + resample) steps, ending with a final set of full assignments, which can be used to approximate the filtering distribution $\mathbb{P}(H_3 \mid E = e)$.

Particle filtering: implementation

For filtering questions, can optimize:

- Keep only value of last H_i for each particle
- Store count for each unique particle

$$\begin{array}{ll} \{H_1:0, H_2:1:H_3:1\} & 1 \\ \{H_1:0, H_2:1:H_3:1\} & 1 \\ \{H_1:1, H_2:2:H_3:2\} & 2 \\ \{H_1:1, H_2:1:H_3:2\} & 2 \\ \{H_1:1, H_2:2:H_3:2\} & 2 \end{array} \quad \Rightarrow \quad \begin{array}{l} 1 \text{ (2x)} \\ 2 \text{ (3x)} \end{array}$$

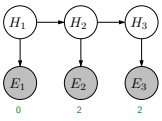
- So far, we have presented a version of particle filtering where each particle at the end is a full assignment to all the variables. This allows us to approximately answer a variety of different questions based on the induced distribution.
- However, if we're only interested in filtering questions, then we can perform two optimizations.
- First, in tracking applications, we only care about the last location H_i, and future steps only depend on the value of H_i. Therefore, we often just store the value of H_i rather than the entire trajectory.
- Second, since we have discrete variables, many particles might have the same value of H_i, so we can just store the counts of each value rather than storing duplicate values.

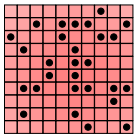
Particle filtering demo

[see web version]

- Now let us visualize particle filtering in a more realistic, interactive object tracking setting.
- Consider an object is moving around in a grid and we are trying to figure out its location $H_i \in \{1, \ldots, \text{grid-width}\} \times \{1, \ldots, \text{grid-height}\}$.
- The transition distribution places a uniform distribution over moving north, moving south, moving east, moving west, or staying put.
- The emission distribution places a uniform distribution over locations E_i that are within 3 steps (both vertically and horizontally) of the actual position H_i. In the textbox, you can change the emission distribution dynamically (`observeFactor`).
- When you hit ctrl-enter, you can see the noisy sensor readings (visualized as a yellow dot bouncing around).
- If you increase the number of particles, you can see a red cloud representing where the particles are, where the intensity of a square is proportional to the number of particles in that square.
- You can now set `showTruePosition = true` to see the actual H_i that generated E_i. You can see that the cloud is able to track the true location reasonably well, although there are occasional errors.

Summary





$$\mathbb{P}(H_3 \mid E_1 = 0, E_2 = 2, E_3 = 2)$$

- Use particles to represent an approximate distribution

Propose (transitions) **Weight** (emissions) **Resample**

- Can scale to large number of locations (unlike forward-backward)
- Maintains better particle diversity (compared to beam search)

- In summary, we have presented particle filtering, an inference algorithm for HMMs that approximately computes filtering questions of the form: where is the object currently given all the past noisy sensor readings?
- Particle filtering represents distributions over hidden variables with a set of particles. To advance the particles to the next time step, it proposes new positions based on transition probabilities. It then weights these guesses based on evidence from the emission probabilities. Finally, it resamples from the normalized weights to redistribute the precious particle resources.
- Compared to the forward-backward algorithm, both beam search and particle filtering can scale up to a large number of locations (assuming most of them are unlikely). Unlike beam search, however, particle filtering uses randomness to ensure better diversity of the particles.
- Particle filtering is also called sequential Monte Carlo and there are many more sophisticated extensions that I'd encourage you to learn about.

Overall Summary: Bayesian Networks II

- Probabilistic programs as equivalent to Bayesian Networks
- Gibbs sampling is an algorithm for estimating marginal probabilities
- Forward Backward algorithm: Dynamic programming for inference (filtering and smoothing)
- Particle Filtering: Approximate inference for HMMs with large domains
- Next: learning the parameters of Bayesian networks

- In summary, we started by outlining probabilistic programs, which are a programmatic way to describing Bayesian networks.
- Next, we covered Gibbs sampling for computing marginal probabilities of a Markov network
- Then, we discussed the forward-backward algorithm, which implements inference as an application of dynamic programming for filtering and smoothing
- Next Lecture, we will finish inference and cover algorithms for learning the parameters of Bayesian networks