15 PROBABILISTIC REASONING OVER TIME

In which we try to interpret the present, understand the past, and perhaps predict the future, even when very little is crystal clear

Agents in uncertain environments must be able to keep track of the current state of the environment, just like the logical agents in Part III. The task is made more difficult by partial and noisy percepts and uncertainty about how the environment changes over time. At best, the agent will be able to obtain only a probabilistic assessment of the current situation. This chapter describes the representations and inference algorithms that make this possible, building on the ideas introduced in Chapter 14.

The basic approach is described in Section 15.1: a changing world is modelled using a random variable for each aspect of the world state *at each point in time*. The relations among these variables describe how the state evolves. Section 15.2 defines the basic inference tasks and describes the general structure of inference algorithms for temporal models. Then we describe three specific kinds of models: **hidden Markov models**, **Kalman filters**, and **dynamic Bayesian networks** (which include hidden Markov models and Kalman filters as special cases). Finally, Section 15.6 explains how temporal probability models form the core of modern speech recognition systems. Learning plays a central role in the construction of all these models, but detailed investigation of learning algorithms is left until Part VI.

15.1 TIME AND UNCERTAINTY

We have developed our techniques for probabilistic reasoning in the context of **static** worlds, in which each random variable has a single fixed value. For example, when repairing a car, we assume that whatever is broken remains broken during the process of diagnosis; our job is to infer the state of the car from observed evidence, which also remains fixed.

Now consider a slightly different problem—treating a diabetic patient. As in the case of car repair, we have evidence such as recent insulin doses, food intake, blood sugar measurements, and other physical signs. The task is to assess the current state of the patient, including actual blood sugar level and insulin level. Given this information, the doctor (or patient) makes a decision about food intake and insulin dose. Unlike the case of car repair, here the *dynamic* aspects of the problem are essential. Blood sugar levels, and measurements thereof, can change rapidly over time, depending on recent food intake and insulin doses, metabolic activity, time of day, and so on. To assess the current state from the history of evidence and to predict the outcomes of treatment actions, we must model these changes.

The same considerations arise in many other contexts, ranging from tracking the economic activity of a nation, given approximate and partial statistics, to understanding a sequence of spoken words, given noisy and ambiguous acoustic measurements. How can dynamic situations like these be modelled?

States and observations

TIME SLICE

The basic approach we will adopt is very similar to the idea underlying situation calculus, as described in Chapter 10: the process of change can be viewed as a series of snapshots, each of which describes the state of the world at a particular time. Each snapshot or **time slice** contains a set of random variables, some of which are observable and some of which are not. For simplicity, we will assume that the same subset of variables is observable in each time slice (although this is not strictly necessary in anything that follows). We will use $\mathbf{X}_t$ to denote the set of unobservable state variables at time t and $\mathbf{E}_t$ to denote the set of observable evidence variables. The observation at time t is $\mathbf{E}_t = \mathbf{e}_t$ for some set of values $\mathbf{e}_t$.

Consider the following oversimplified example. Suppose you are the security guard at some secret underground installation. You want to know if it's raining today, but your only access to the outside world occurs each morning when you see the director coming in with, or without, an umbrella. For each day t, the set $\mathbf{E}_t$ thus contains a single evidence variable U_t (whether the umbrella appears), and the set $\mathbf{X}_t$ contains a single state variable R_t (whether it is raining). Other problems may involve larger sets of variables. In the diabetes example, we might have evidence variables such as $MeasuredBloodSugar_t$, $PulseRate_t$, etc., with state variables such as $BloodSugar_t$, $StomachContents_t$, and so on.[1]

The interval between time slices also depends on the problem. For diabetes monitoring, a suitable interval might be an hour rather than a day. In this chapter, we will generally assume a fixed, finite interval; this means that times can be labelled by integers. We will assume that the state sequence starts at $t=0$; for various uninteresting reasons, we will assume that evidence starts arriving at $t=1$ rather than $t=0$. Hence our umbrella world is represented by state variables R_0, R_1, $R_2, \ldots$ and evidence variables U_1, $U_2, \ldots$. We will use the notation $a : b$ to denote the sequence of integers from a to b, and the notation $\mathbf{X}_{a:b}$ to denote the corresponding set of variables from $\mathbf{X}_a$ to $\mathbf{X}_b$. For example, $U_{1:3}$ corresponds to the variables U_1, U_2, U_3.

Stationary processes and the Markov assumption

Having decided on the set of state and evidence variables for a given problem, the next step is to specify the dependencies among the variables. We could follow the procedure laid down in Chapter 14, placing the variables in some order and asking questions about conditional

[1] Notice that $BloodSugar_t$ and $MeasuredBloodSugar_t$ are not the same variable; this is how we deal with noisy measurements of actual quantities.

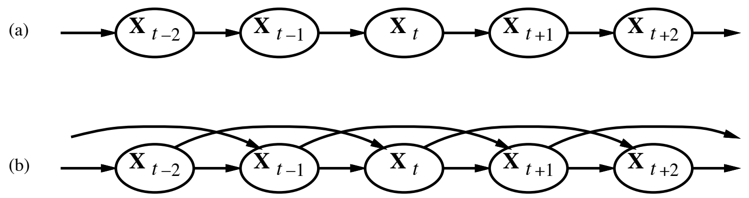


Figure 15.1 (a) Bayesian network structure corresponding to a first-order Markov process with state defined by the variables $\mathbf{X}_t$. (b) A second-order Markov process.

independence of predecessors given some set of parents. One obvious choice is to order the variables in their natural temporal order, since cause usually precedes effect and we prefer to add the variables in causal order.

We would quickly run into an obstacle, however: the set of variables is unbounded, since it includes the state and evidence variables for every time slice. This actually creates two problems: first, we might have to specify an unbounded number of conditional probability tables—one for each variable in each slice; and second, each one might involve an unbounded number of parents.

STATIONARY PROCESS

The first problem is solved by assuming that changes in the world state are caused by a **stationary process**—that is, a process of change that is governed by laws that do not themselves change over time. (Don't confuse *stationary* with *static*: in a *static* process, the state itself does not change.) In the umbrella world, then, the conditional probability that the umbrella appears, $\mathbf{P}(U_t|Parents(U_t))$, is the same for all t. Given the assumption of stationarity, therefore, we need specify conditional distributions only for the variables within a "representative" time slice.

MARKOV ASSUMPTION

MARKOV PROCESSES

MARKOV CHAINS

FIRST-ORDER MARKOV PROCESS

The second problem, that of handling the potentially infinite number of parents, is solved by making what is called a **Markov assumption**, that is, that the current state depends on only a *finite* history of previous states. Processes satisfying this assumption were first studied in depth by the Russian statistician A. A. Markov and are called **Markov processes** or **Markov chains**. They come in various flavors; the simplest is the **first-order Markov process**, in which the current state depends only on the previous state and not on any earlier states. Using our notation, the corresponding conditional independence assertion states that, for all t,

$$\mathbf{P}(\mathbf{X}_t|\mathbf{X}_{0:t-1}) = \mathbf{P}(\mathbf{X}_t|\mathbf{X}_{t-1}) \tag{15.1}$$

TRANSITION MODEL

Hence, in a first-order Markov process, the laws describing how the state evolves over time are contained entirely within the conditional distribution $\mathbf{P}(\mathbf{X}_t|\mathbf{X}_{t-1})$, which we call the **transition model**.[2] in The transition model for a second-order Markov process is the conditional distribution $\mathbf{P}(\mathbf{X}_t|\mathbf{X}_{t-2}, \mathbf{X}_{t-1})$. Figure 15.1 shows the Bayesian network structures corresponding to first-order and second-order Markov processes.

In addition to restricting the parents of the state variables $\mathbf{X}_t$, we must also restrict the

[2] The transition model is the probabilistic analogue of the Boolean update circuits in Chapter 7 and the successor-state axioms in Chapter 10.

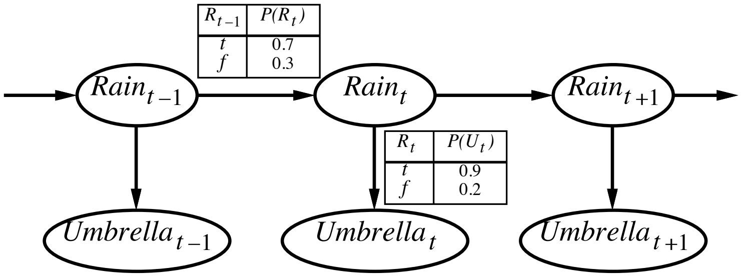


Figure 15.2 Bayesian network structure and conditional distributions describing the umbrella world. The transition model is $P(Rain_t|Rain_{t-1})$ and the sensor model is $P(Umbrella_t|Rain_t)$.

parents of the evidence variables $\mathbf{E}_t$. Typically, we will assume that the evidence variables at time t depend only on the current state:

$$\mathbf{P}(\mathbf{E}_t|\mathbf{X}_{0:t}, \mathbf{E}_{0:t-1}) = \mathbf{P}(\mathbf{E}_t|\mathbf{X}_t) \qquad (15.2)$$

SENSOR MODEL

The conditional distribution $\mathbf{P}(\mathbf{E}_t|\mathbf{X}_t)$ is called the **sensor model** (or sometimes the **observation model**), because it describes how the "sensors"—that is, the evidence variables, are affected by the actual state of the world. Notice the direction of the dependence: the "arrow" goes from state to sensor values because the state of the world *causes* the sensors to take on particular values. In the umbrella world, for example, the rain *causes* the umbrella to appear. (The inference process, of course, goes in the other direction; the distinction between the direction of modelled dependencies and the direction of inference is one of the principal advantages of Bayesian networks.)

In addition to the transition model and sensor model, we also need to specify a prior probability $\mathbf{P}(\mathbf{X}_0)$ over the states at time 0. These three distributions, combined with the the conditional independence assertions in Equations (15.1) and (15.2), give us a specification of the complete joint distribution over all the variables. For any finite t, we have

$$\mathbf{P}(\mathbf{X}_0, \mathbf{X}_1, \ldots, \mathbf{X}_t, \mathbf{E}_1, \ldots, \mathbf{E}_t) = \mathbf{P}(\mathbf{X}_0) \prod_{i=1}^{t} \mathbf{P}(X_i|\mathbf{X}_{i-1})\mathbf{P}(\mathbf{E}_i|\mathbf{X}_i) \qquad (15.3)$$

The independence assumptions correspond to a very simple structure for the Bayesian network describing the whole system. Figure 15.2 shows the network structure for the umbrella example, including the conditional distributions for the transition and sensor models.

The structure in the figure assumes a first-order Markov process, because the probability of rain is assumed to depend only on whether it rained the previous day. Whether such an assumption is reasonable depends on the domain itself. The first-order Markov assumption says that the state variables contain *all* the information needed to characterize the probability distribution for the next time slice. Sometimes the assumption is exactly true—for example,

RANDOM WALK

if a particle is executing a **random walk** along the x–axis, changing its position by ±1 at each time step, then using the x–coordinate as the state gives a first-order Markov process. Sometimes the assumption is only approximate, as in the case of predicting rain just based on whether it rained the previous day. There are two possible fixes if the approximation proves too inaccurate:

1. Increasing the order of the Markov process model. For example, we could make a second-order model by adding $Rain_{t-2}$ as a parent of $Rain_t$, which might give slightly more accurate predictions.
2. Increasing the set of state variables. For example, we could add $Temperature_t$ and $Pressure_t$ to help in predicting the weather.

Exercise 15.1 asks you to show that the first solution—increasing the order—can always be reformulated as an increase in the set of state variables, keeping the order fixed. Notice that adding state variables may improve predictive power but also increases the prediction *requirements*, since we also have to predict the new variables. Thus, we are looking for a "self-sufficient" set of variables, which really means that we have to understand the "physics" of the process being modelled. The requirement for accurate modelling of the process is obviously lessened if we can add new sensors (e.g., measurements of temperature and pressure) that provide information directly about the new state variables.

Consider, for example, the problem of tracking a robot wandering randomly on the X–Y plane. One might propose that the position and velocity are a sufficient set of state variables: one can simply use Newton's laws to calculate the new position, and the velocity may change unpredictably. If the robot is battery-powered, however, then battery exhaustion would tend to have a systematic effect on the change in velocity. Because this in turn depends on how much power was used by all previous maneuvers, the Markov property is violated. We can restore the Markov property by including the charge level $Battery_t$ as one of the state variables that comprise $\mathbf{X}_t$. This helps in predicting the motion of the robot, but in turn requires a model for predicting $Battery_t$ given $Battery_{t-1}$ and the velocity. In some cases this can be done reliably; accuracy would be improved by *adding a new sensor* that measures the battery level.

15.2 INFERENCE IN TEMPORAL MODELS

Having set up the structure of a generic temporal model, we can formulate the basic inference tasks that must be solved. They are as follows:

FILTERING
MONITORING
BELIEF STATE

◇ **Filtering** or **monitoring**: this is the task of computing the **belief state**—the posterior distribution over the current state, given all evidence to date. That is, we wish to compute $\mathbf{P}(\mathbf{X}_t|\mathbf{e}_{1:t})$, assuming that evidence arrives in a continuous stream beginning at $t=1$. In the umbrella example, this would mean computing the probability of rain today, given all the observations of the umbrella-carrier made so far. Filtering is what a rational agent needs to do in order to keep track of the current state so that rational decisions can be made (see Chapter 17). It turns out that an almost identical calculation provides the **likelihood** of the evidence sequence, i.e., $P(\mathbf{e}_{1:t})$.

PREDICTION

◇ **Prediction**: This is the task of computing the posterior distribution over the *future* state, given all evidence to date. That is, we wish to compute $\mathbf{P}(\mathbf{X}_{t+k}|\mathbf{e}_{1:t})$ for some $k > 0$.

In the umbrella example, this might mean computing the probability of rain three days from now, given all the observations of the umbrella-carrier made so far. Prediction is useful for evaluating possible courses of action.

SMOOTHING
HINDSIGHT

◇ **Smoothing** or **hindsight**: This is the task of computing the posterior distribution over a *past* state, given all evidence up to the present. That is, we wish to compute $\mathbf{P}(\mathbf{X}_k|\mathbf{e}_{1:t})$ for some k such that $0 \leq k < t$. In the umbrella example, this might mean computing the probability that it rained last Wednesday, given all the observations of the umbrella-carrier made up to today. Hindsight provides a better estimate of the state than was available at the time, because it incorporates more evidence.

◇ **Most likely explanation**: Given a sequence of observations, we may wish to find the most likely sequence of states that generated those observations. That is, we wish to compute $\arg\max_{\mathbf{x}_{1:t}} P(\mathbf{x}_{1:t}|\mathbf{e}_{1:t})$. For example, if the umbrella appears on each of the first three days and is absent on the fourth, then the most likely explanation is that it rained on the first three days and did not rain on the fourth. Algorithms for this task are useful in many applications, including speech recognition—where the aim is to find the most likely sentence, given a series of sounds—and reconstruction of bit strings transmitted over a noisy channel.

In addition to these tasks, methods are also needed for *learning* the transition and sensor models from observations. Just as with static Bayesian networks, DBN learning can be done as a by-product of inference. Inference provides an estimate of what transitions actually occurred and of what states generated the sensor readings, and these estimates can be used to update the models. The updated models provide new estimates, and the process iterates to convergence. The overall process is an instance of the **EM algorithm** (see Section 19.3). One point to note is that learning requires the full smoothing inference, rather than filtering, because it provides better estimates of the states of the process. Learning with filtering may fail to converge correctly; consider, for example, the problem of learning to solve murders—hindsight is *always* required to infer what happened at the murder scene.

Algorithms for the four inference tasks listed in the preceding paragraph can be described first at a generic level, independent of the particular kind of model employed. Further improvements specific to each family of models will be described in the corresponding sections.

Filtering and prediction

Let us begin with filtering. We will show that this can be done in a simple online fashion: given the result of filtering up to time t, one can easily compute the result for $t+1$ given the new evidence $\mathbf{e}_{t+1}$. That is,

$$\mathbf{P}(\mathbf{X}_{t+1}|\mathbf{e}_{1:t+1}) = f(\mathbf{e}_{t+1}, \mathbf{P}(\mathbf{X}_t|\mathbf{e}_{1:t}))$$

RECURSIVE ESTIMATION

for some function f. This process is often called **recursive estimation**. We can view the calculation as actually being composed of two parts: first, the current state distribution is projected forward from t to $t+1$, then it is updated using the new evidence $\mathbf{e}_{t+1}$. This two-part process emerges quite simply:

$$\begin{aligned}\mathbf{P}(\mathbf{X}_{t+1}|\mathbf{e}_{1:t+1}) &= \mathbf{P}(\mathbf{X}_{t+1}|\mathbf{e}_{1:t},\mathbf{e}_{t+1}) \quad \text{dividing up the evidence}\\ &= \alpha\mathbf{P}(\mathbf{e}_{t+1}|\mathbf{X}_{t+1},\mathbf{e}_{1:t})\mathbf{P}(\mathbf{X}_{t+1}|\mathbf{e}_{1:t}) \quad \text{using Bayes' rule}\\ &= \alpha\mathbf{P}(\mathbf{e}_{t+1}|\mathbf{X}_{t+1})\mathbf{P}(\mathbf{X}_{t+1}|\mathbf{e}_{1:t}) \quad \text{by the Markov property of evidence}\end{aligned}$$

The second term, $\mathbf{P}(\mathbf{X}_{t+1}|\mathbf{e}_{1:t})$ represents a one-step prediction of the next state, and the first term updates this with the new evidence; notice that $\mathbf{P}(\mathbf{e}_{t+1}|\mathbf{X}_{t+1})$ is obtainable directly from the sensor model. Now we obtain the one-step prediction for the next state by conditioning on the current state $\mathbf{X}_t$:

$$\begin{aligned}\mathbf{P}(\mathbf{X}_{t+1}|\mathbf{e}_{1:t+1}) &= \alpha\mathbf{P}(\mathbf{e}_{t+1}|\mathbf{X}_{t+1})\sum_{\mathbf{x}_t}\mathbf{P}(\mathbf{X}_{t+1}|\mathbf{x}_t,\mathbf{e}_{1:t})P(\mathbf{x}_t|\mathbf{e}_{1:t})\\ &= \alpha\mathbf{P}(\mathbf{e}_{t+1}|\mathbf{X}_{t+1})\sum_{\mathbf{x}_t}\mathbf{P}(\mathbf{X}_{t+1}|\mathbf{x}_t)P(\mathbf{x}_t|\mathbf{e}_{1:t}) \quad \text{using the Markov property} \qquad (15.4)\end{aligned}$$

Within the summation, the first factor is simply the transition model, and the second is the current state distribution. Hence, we have the desired recursive formulation. We can think of the filtered estimate $\mathbf{P}(\mathbf{X}_t|\mathbf{e}_{1:t})$ as a "message" $\mathbf{f}_{1:t}$ that is propagated forward along the sequence, modified by each transition and updated by each new observation. The process is

$$\mathbf{f}_{1:t+1} = \alpha\,\textsc{Forward}(\mathbf{f}_{1:t},\mathbf{e}_{t+1})$$

where FORWARD implements the update described in Equation (15.4).

When all the state variables are discrete, the time for each update is constant (independent of t), and the space required is also constant. (The constants depend, of course, on the size of the state space and the specific type of the temporal model in question.) *The time and space requirements for updating must be constant if an agent with limited memory is to keep track of the current state distribution over an unbounded sequence of observations.*

Let us illustrate the filtering process for two steps in the basic umbrella example (see Figure 15.2). We assume that our security guard has some prior belief as to whether it rained on day 0, just before the observation sequence begins. Let's suppose this is $\mathbf{P}(R_0) = \langle 0.5, 0.5\rangle$. Now we process the two observations as follows:

- On day 1, the umbrella appears, so $U_1 = true$. The prediction from $t = 0$ to $t = 1$ is

  $$\begin{aligned}\mathbf{P}(R_1) &= \sum_{r_0}\mathbf{P}(R_1|r_0)P(r_0)\\ &= \langle 0.7, 0.3\rangle \times 0.5 + \langle 0.3, 0.7\rangle \times 0.5 = \langle 0.5, 0.5\rangle\end{aligned}$$

  and updating with the evidence for $t = 1$ gives

  $$\begin{aligned}\mathbf{P}(R_1|u_1) &= \alpha\mathbf{P}(u_1|R_1)\mathbf{P}(R_1) = \alpha\langle 0.9, 0.2\rangle\langle 0.5, 0.5\rangle\\ &= \alpha\langle 0.45, 0.1\rangle \approx \langle 0.818, 0.182\rangle\end{aligned}$$

- On day 2, the umbrella appears, so $U_2 = true$. The prediction from $t = 1$ to $t = 2$ is

  $$\begin{aligned}\mathbf{P}(R_2|u_1) &= \sum_{r_1}\mathbf{P}(R_2|r_1)P(r_1|u_1)\\ &= \langle 0.7, 0.3\rangle \times 0.818 + \langle 0.3, 0.7\rangle \times 0.182 \approx \langle 0.627, 0.373\rangle\end{aligned}$$

and updating with the evidence for $t = 2$ gives

$$\begin{aligned}\mathbf{P}(R_2|u_1,u_2) &= \alpha\mathbf{P}(u_2|R_2)\mathbf{P}(R_2|u_1) = \alpha\langle 0.9, 0.2\rangle\langle 0.627, 0.373\rangle \\ &= \alpha\langle 0.565, 0.075\rangle \approx \langle 0.883, 0.117\rangle\end{aligned}$$

Intuitively, the probability of rain increases from day 1 to day 2 because rain persists. Exercise 15.2(a) asks you to investigate this tendency further.

The task of **prediction** can simply be seen as filtering without the addition of new evidence. In fact, the filtering process already incorporates a one-step prediction, and it is easy to derive the following recursive computation for predicting the state at $t + k + 1$ from a prediction for $t + k$:

$$\mathbf{P}(\mathbf{X}_{t+k+1}|\mathbf{e}_{1:t}) = \sum_{\mathbf{x}_{t+k}} \mathbf{P}(\mathbf{X}_{t+k+1}|\mathbf{x}_{t+k})P(\mathbf{x}_{t+k}|\mathbf{e}_{1:t}) \qquad (15.5)$$

Naturally, this computation involves only the transition model and not the sensor model.

It is interesting to consider what happens as we try to predict further and further into the future. As Exercise 15.2(b) shows, the predicted distribution for rain converges to a fixed point $\langle 0.5, 0.5\rangle$, after which it remains constant for all time. This is the **stationary distribution** of the Markov process defined by the transition model (see also page 522). A great deal is known about the properties of such distributions and about the **mixing time**—roughly, the time taken to reach the fixed point. In practical terms, this dooms to failure any attempt to predict the *actual* state for a number of steps that is more than a small fraction of the mixing time. The more uncertainty there is in the transition model, the shorter will be the mixing time and the more the future is obscured.

MIXING TIME

In addition to filtering and prediction, we can also use a forward recursion to compute the the **likelihood** of the evidence sequence, i.e., $P(\mathbf{e}_{1:t})$. This is a useful quantity if we want to compare different possible temporal models that might have produced the same evidence sequence; for example, in Section 15.6, we compare different words that might have produced the same sound sequence. For this recursion, we use a likelihood message $\boldsymbol{\ell}_{1:t} = \mathbf{P}(\mathbf{X}_t, \mathbf{e}_{1:t})$. It is a simple exercise to show that

$$\boldsymbol{\ell}_{1:t+1} = \text{FORWARD}(\ell_{1:t}, \mathbf{e}_{t+1}) \ .$$

Having computed $\boldsymbol{\ell}_{1:t}$, we obtain the actual likelihood by summing out $\mathbf{X}_t$:

$$L_{1:t} = P(\mathbf{e}_{1:t}) = \sum_{\mathbf{x}_t} \boldsymbol{\ell}_{1:t}(\mathbf{x}_t) \ . \qquad (15.6)$$

Smoothing

As we said earlier, **smoothing** is the process of computing the distribution over past states given evidence up to the present, that is, $\mathbf{P}(\mathbf{X}_k|\mathbf{e}_{1:t})$ for $1 \leq k < t$ (see Figure 15.3). This is done most conveniently in two parts—the evidence up to k and the evidence from $k + 1$ to t:

$$\begin{aligned}\mathbf{P}(\mathbf{X}_k|\mathbf{e}_{1:t}) &= \mathbf{P}(\mathbf{X}_k|\mathbf{e}_{1:k}, \mathbf{e}_{k+1:t}) \\ &= \alpha\mathbf{P}(\mathbf{X}_k|\mathbf{e}_{1:k})\mathbf{P}(\mathbf{e}_{k+1:t}|\mathbf{X}_k, \mathbf{e}_{1:k}) \quad \text{using Bayes' rule} \\ &= \alpha\mathbf{P}(\mathbf{X}_k|\mathbf{e}_{1:k})\mathbf{P}(\mathbf{e}_{k+1:t}|\mathbf{X}_k) \quad \text{using conditional independence} \\ &= \alpha\mathbf{f}_{1:k}\mathbf{b}_{k+1:t} \qquad (15.7)\end{aligned}$$

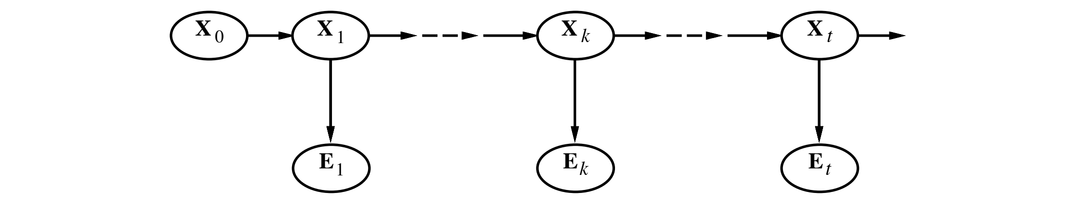


Figure 15.3 Smoothing computes $\mathbf{P}(\mathbf{X}_k|\mathbf{e}_{1:t})$, the posterior distribution of the state at some past time k given a complete sequence of observations from 1 to t.

where we have defined a "backward" message $\mathbf{b}_{k+1:t} = \mathbf{P}(\mathbf{e}_{k+1:t}|\mathbf{X}_k)$, analogous to the forward message $\mathbf{f}_{1:k}$. The forward message $\mathbf{f}_{1:k}$ can be computed by filtering forward from 1 to k, as given by Equation (15.4). It turns out that the backward message $\mathbf{b}_{k+1:t}$ can be computed by a recursive process that runs *backwards* from t:

$$\begin{aligned}
\mathbf{P}(\mathbf{e}_{k+1:t}|\mathbf{X}_k) &= \sum_{\mathbf{x}_{k+1}} \mathbf{P}(\mathbf{e}_{k+1:t}|\mathbf{X}_k, \mathbf{x}_{k+1})\mathbf{P}(\mathbf{x}_{k+1}|\mathbf{X}_k) \quad \text{conditioning on } \mathbf{X}_{k+1} \\
&= \sum_{\mathbf{x}_{k+1}} P(\mathbf{e}_{k+1:t}|\mathbf{x}_{k+1})\mathbf{P}(\mathbf{x}_{k+1}|\mathbf{X}_k) \quad \text{by conditional independence} \\
&= \sum_{\mathbf{x}_{k+1}} P(\mathbf{e}_{k+1}, \mathbf{e}_{k+2:t}|\mathbf{x}_{k+1})\mathbf{P}(\mathbf{x}_{k+1}|\mathbf{X}_k) \\
&= \sum_{\mathbf{x}_{k+1}} P(\mathbf{e}_{k+1}|\mathbf{x}_{k+1})P(\mathbf{e}_{k+2:t}|\mathbf{x}_{k+1})\mathbf{P}(\mathbf{x}_{k+1}|\mathbf{X}_k) \qquad (15.8)
\end{aligned}$$

where the last step follows by conditional independence of $\mathbf{e}_{k+1}$ and $\mathbf{e}_{k+2:t}$ given $\mathbf{X}_{k+1}$. Of the three factors in this summation, the first and third are obtained directly from the model, and the second is the "recursive call." Using the message notation, we have

$$\mathbf{b}_{k+1:t} = \text{BACKWARD}(\mathbf{b}_{k+2:t}, \mathbf{e}_{k+2:t})$$

where BACKWARD implements the update described in Equation (15.8). As with the forward recursion, the time and space required for each update are constant, independent of t.

Given this derivation, we can now see that the two terms in Equation (15.7) can both be computed by recursions through time, one running forward from 1 to k using the filtering equation (15.4) and the other running backward from t to $k+1$ using Equation (15.8). Note that the backward phase is initialized with $\mathbf{b}_{t+1:t} = \mathbf{P}(\mathbf{e}_{t+1:t}|\mathbf{X}_t) = \mathbf{1}$, where $\mathbf{1}$ is a vector of 1s. (Why?)

Let us now illustrate this algorithm for the umbrella example by computing the smoothed estimate for the probability of rain at $t = 1$, given umbrella observations on day 1 and day 2. From Equation (15.7), this is given by

$$\mathbf{P}(R_1|u_1, u_2) = \alpha\mathbf{P}(u_2|R_1)\mathbf{P}(R_1|u_1) \qquad (15.9)$$

The second term we already know to be $\langle .818, .182\rangle$, from the forward filtering process described earlier. The first term can be computed by applying the backward recursion in Equation (15.8):

$$\mathbf{P}(u_2|R_1) = \sum_{r_2} P(u_2|r_2)P(\ |r_2)\mathbf{P}(r_2|R_1)$$

$$= (0.9 \times 1 \times \langle 0.7, 0.3 \rangle) + (0.2 \times 1 \times \langle 0.3, 0.7 \rangle) = \langle 0.69, 0.41 \rangle$$

Plugging this into Equation (15.9), we find that the smoothed estimate for rain on day 1 is

$$\mathbf{P}(R_1|u_1, u_2) = \alpha\langle 0.69, 0.41 \rangle \times \langle 0.818, 0.182 \rangle \approx \langle 0.883, 0.117 \rangle$$

Thus, the smoothed estimate is *higher* than the filtered estimate (0.818) in this case. This is because the umbrella on day 2 makes it more likely to have rained on day 2; in turn, because rain tends to persist, this makes it more likely to have rained on day 1.

Both the forward and backward recursions take a constant amount of time per step, hence the time complexity of smoothing with respect to evidence $\mathbf{e}_{1:t}$ is $O(t)$. This is the complexity for smoothing at a particular time step k. If we want to smooth the whole sequence to get the correct posterior estimate of what actually happened, one obvious method is simply to run the whole smoothing process once for each time step to be smoothed. This results in a time complexity of $O(t^2)$. A better approach uses a very simple application of dynamic programming to reduce this to $O(t)$. A clue appears in the preceding analysis of the umbrella example, where we were able to reuse the results of the forward filtering phase. The key to the linear-time algorithm is to *record the results* of forward filtering over the whole sequence. Then we run the backward recursion from t down to 1, computing the smoothed estimate at each step k from the computed backward message $\mathbf{b}_{k+1:t}$ and the stored forward message $\mathbf{f}_{1:k}$. The algorithm, aptly called the **forward–backward algorithm**, is shown in Figure 15.4.

FORWARD–BACKWARD ALGORITHM

The alert reader will have spotted that the Bayesian network structure shown in Figure 15.3 is a **polytree** in the terminology of Chapter 14. This means that a straightforward application of the clustering algorithm also yields a linear-time algorithm that computes smoothed estimates for the entire sequence. One can show that the forward–backward algorithm is in fact a special case of the polytree propagation algorithm used with clustering methods.

The forward–backward algorithm forms the backbone of the computational methods used in many applications that deal with sequences of noisy observations, ranging from speech recognition to radar tracking of aircraft. As described, it has two practical drawbacks. The first is that its space complexity can be too high for applications where the state space is large and the sequences are long. It uses $O(|\mathbf{f}|t)$ space where $|\mathbf{f}|$ is the size of the representation of the forward message. The space requirement can be reduced to $O(|\mathbf{f}| \log t)$ with a concomitant increase in the time complexity by a factor of $\log t$, as shown in Exercise 15.3. In some cases (see Section 15.3), a constant-space algorithm can be used with no time penalty.

The second drawback of the basic algorithm is that it needs modification to work in an *online* setting where smoothed estimates must be computed for earlier time slices as new observations are continuously added to the end of the sequence. The most common requirement is for **fixed-lag smoothing**, which requires computing the smoothed estimate $\mathbf{P}(\mathbf{X}_{t-d}|\mathbf{e}_{1:t})$ for fixed d. That is, smoothing is done for the time slice d steps behind the current time t; as t increases, the smoothing has to keep up. Obviously, we can run the forward–backward algorithm over the d-step "window" as each new observation is added, but this seems inefficient. In Section 15.3, we will see that fixed-lag smoothing can, in some cases, be done in constant time per update, independent of the lag d.

FIXED-LAG SMOOTHING

```
function FORWARD-BACKWARD(ev, prior) returns a vector of probability distributions
   inputs: ev, a vector of evidence values for steps $1, \ldots, t$
           prior, the prior distribution on the initial state, $\mathbf{P}(\mathbf{X}_0)$
   local variables: fv, a vector of forward messages for steps $0, \ldots, t$
                    b, a representation of the backward message, initially all 1s
                    sv, a vector of smoothed estimates for steps $1, \ldots, t$

   fv[0] ← prior
   for $i = 1$ to $t$ do
      fv[$i$] ← FORWARD(fv[$i - 1$], ev[$i$])
   for $i = t$ downto 1 do
      sv[$i$] ← NORMALIZE(fv[$i$] × b)
      b ← BACKWARD(b, ev[$i$])
   return sv
```

Figure 15.4 The forward–backward algorithm for computing posterior probabilities of a sequence of states given a sequence of observations. The FORWARD and BACKWARD operators are defined by Equations (15.4) and (15.8) respectively.

Finding the most likely sequence

Suppose that $[true, true, false, true, true]$ is the umbrella sequence for the security guard's first five days on the job. What is the most likely weather sequence that explains this? Does the absence of the umbrella on day 3 mean that it wasn't raining, or did the director forget to bring it? If it didn't rain on day 3, perhaps (because weather tends to persist) it didn't rain on day 4 either, but the director brought the umbrella just in case. In all, there are 2^5 possible weather sequences we could pick. Is there a way to find the most likely one, short of enumerating all of them?

One approach we could try is the following linear-time procedure: use the smoothing algorithm to find the posterior distribution for the weather at each time step, then construct the sequence using the most likely weather at each step according to the posterior. Such an approach should set off alarm bells in the reader's head, because the posteriors computed by smoothing are distributions over *single* time steps, whereas to find the most likely *sequence* we must consider *joint* probabilities over all the time steps. The results may in fact be quite different (see Exercise 15.4).

There *is* a linear-time algorithm for finding the most likely sequence, but it requires a little more thought. It relies on the same Markov property that yielded efficient algorithms for filtering and smoothing. The easiest way to think about the problem is to view each sequence as a *path* through a graph whose nodes are the possible *states* at each time step. Such a graph is shown for the umbrella world in Figure 15.5(a). Now consider the task of finding the most likely path through this graph, where the likelihood of any path is the product of the transition probabilities along the path and the probabilities of the given observations at each state. Let's focus in particular on paths that reach the state $Rain_5 = true$. Because of the Markov property, we can make the following simple observation: the most likely path to the state $Rain_5 = true$ consists of the most likely path to *some* state at time 4 followed by a transition to $Rain_5 = true$; and the state at time 4 that will become part of the path to $Rain_5 = true$ is whichever maximizes the likelihood of that path. In other words, *there is a recursive relationship between most likely paths to each state* $\mathbf{x}_{t+1}$ *and most likely paths to each state* $\mathbf{x}_t$. We can write this relationship as an equation connecting the probabilities of the paths:

$$\begin{aligned} &\max_{\mathbf{x}_1 \ldots \mathbf{x}_t} \mathbf{P}(\mathbf{x}_1, \ldots, \mathbf{x}_t, \mathbf{X}_{t+1} | \mathbf{e}_{1:t+1}) \\ &= \mathbf{P}(\mathbf{e}_{t+1} | \mathbf{X}_{t+1}) \max_{\mathbf{x}_t} \left(\mathbf{P}(\mathbf{X}_{t+1} | \mathbf{x}_t) \max_{\mathbf{x}_1 \ldots \mathbf{x}_{t-1}} P(\mathbf{x}_1, \ldots, \mathbf{x}_{t-1}, \mathbf{x}_t | \mathbf{e}_{1:t}) \right) \end{aligned} \tag{15.10}$$

Equation (15.10) is *identical* to the filtering equation (15.4) except that

1. the forward message $\mathbf{f}_{1:t} = \mathbf{P}(\mathbf{X}_t | \mathbf{e}_{1:t})$ is replaced by the message

   $$\mathbf{m}_{1:t} = \max_{\mathbf{x}_1 \ldots \mathbf{x}_{t-1}} \mathbf{P}(\mathbf{x}_1, \ldots, \mathbf{x}_{t-1}, \mathbf{X}_t | \mathbf{e}_{1:t}),$$

   that is, the probabilities of the most likely path to each state $\mathbf{x}_t$; and
2. the summation over $\mathbf{x}_t$ in Equation (15.4) is replaced by the maximization over $\mathbf{x}_t$ in Equation (15.10).

Thus, the algorithm for computing the most likely sequence is very similar to filtering: it runs forward along the sequence, computing the **m** message at each time step using Equation (15.10). The progress of this computation is shown in Figure 15.5(b). At the end, it will have the probability for the most likely sequence reaching *each* of the final states. One can thus easily select the most likely sequence overall (the state outlined in bold). In order to identify the actual sequence, as opposed to just computing its probability, the algorithm will also need to keep pointers from each state back to the best state that leads to it (shown in bold); the sequence is identified by following the pointers back from the best final state.

VITERBI ALGORITHM

The algorithm we have just described is called the **Viterbi algorithm**, after its inventor. Like the filtering algorithm, its complexity is linear in t, the length of the sequence. Unlike filtering, however, its space requirement is also linear in t. This is because the Viterbi algorithm needs to keep the pointers that identify the best sequence leading to each state.

15.3 Hidden Markov Models

The preceding section developed algorithms for temporal probabilistic reasoning using a very general framework, independent of the specific form of the transition and sensor models. In this and the following two sections, we discuss more concrete models and applications that illustrate the power of the basic algorithms and in some cases allow further improvements.

HIDDEN MARKOV MODEL

We begin with the **hidden Markov model** or **HMM**. An HMM is a temporal probabilistic model in which the state of the process is described by a *single*, *discrete* random variable. The possible values of the variable are the possible states of the world. The umbrella example described in the preceding section is therefore an HMM, since it has just one state variable, $Rain_t$. Additional state variables can be added to a temporal model while staying within the

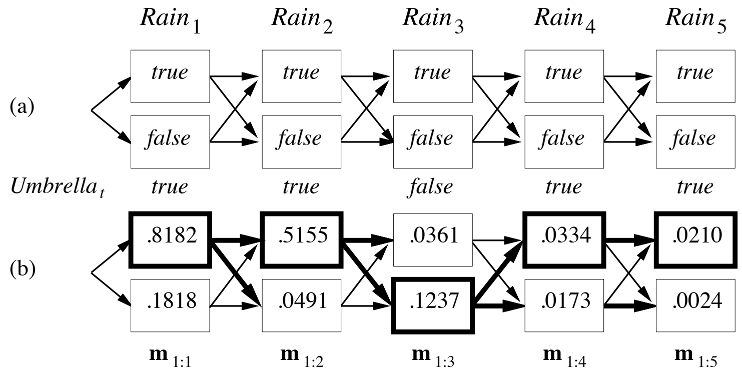


Figure 15.5 (a) Possible state sequences for $Rain_t$ can be viewed as paths through a graph of the possible states at each time step. (States are shown using square nodes to avoid confusion with nodes in a Bayesian network.) (b) Operation of the Viterbi algorithm for the umbrella observation sequence $[true, true, false, true, true]$. For each time step t, we have shown the values of the message $\mathbf{m}_{1:t}$ which gives the probability of the best sequence reaching each state at time t. Also, for each state, the bold arrow leading into it indicates its best predecessor. Following the bold arrows back from the most likely state in $\mathbf{m}_{1:5}$ gives the most likely sequence.

HMM framework, but only by combining all the state variables into a single "megavariable" whose values are all possible tuples of values of the individual state variables. HMMs usually have a single, discrete evidence variable as well, but this restriction is less important. We will see that the restricted structure of HMMs allows for a very simple and elegant matrix implementation of all the basic algorithms.[3] Section 15.6 shows how HMMs are used for speech recognition.

Simplified matrix algorithms

With a single, discrete state variable X_t, we can give concrete form to the representations of the transition model, the sensor model, and the forward and backward messages. Let the state variable X_t have values denoted by integers $1, \ldots, S$, where S is the number of possible states. The transition model $\mathbf{P}(X_t|X_{t-1})$ becomes an $S \times S$ matrix $\mathbf{T}$, where

$$\mathbf{T}_{ij} = P(X_t = j | X_{t-1} = i)$$

That is, $\mathbf{T}_{ij}$ is the probability of a transition from state i to state j. For example, the transition matrix for the umbrella world is

$$\mathbf{T} = \mathbf{P}(X_t|X_{t-1}) = \begin{pmatrix} 0.7 & 0.3 \\ 0.3 & 0.7 \end{pmatrix}$$

We also put the sensor model in matrix form. In this case, because the value of the evidence variable E_t is known to be e_t (say), we need only use that part of the model specifying the

[3] For this reason, the reader unfamiliar with basic operations on vectors and matrices might wish to consult Appendix A before continuing.

probability that e_t appears. For each time step t, we construct a diagonal matrix $\mathbf{O}_t$ whose diagonal entries are given by the values $P(e_t|X_t = i)$ and whose other entries are 0. For example, on day 1 in the umbrella world, $U_1 = true$, so from the table in Figure 15.2 we have

$$\mathbf{O}_1 = \begin{pmatrix} 0.9 & 0 \\ 0 & 0.2 \end{pmatrix}$$

Now, if we use column vectors to represent the forward and backward messages, the computations become simple matrix–vector operations. The forward equation (15.4) becomes

$$\mathbf{f}_{1:t+1} = \alpha \mathbf{O}_{t+1} \mathbf{T}^\top \mathbf{f}_{1:t} \qquad (15.11)$$

and the backward equation (15.8) becomes

$$\mathbf{b}_{k+1:t} = \mathbf{T}\mathbf{O}_{k+1}\mathbf{b}_{k+2:t} \qquad (15.12)$$

From these equations, we can see that the time complexity of the forward–backward algorithm (Figure 15.4) applied to a sequence of length t is $O(S^2 t)$, as each step requires multiplying an S-element vector by an $S \times S$ matrix. The space requirement is $O(St)$, because the forward pass stores t vectors of size S.

Besides providing an elegant description and implementation of the filtering and smoothing algorithms for HMMs, the matrix formulation also reveals opportunities for improved algorithms. The first is a simple variation on the forward–backward algorithm that allows smoothing to be carried out using *constant* space, independent of the length of the sequence. The idea is that smoothing for any particular time slice k requires the simultaneous presence of both the forward and backward messages, $\mathbf{f}_{1:k}$ and $\mathbf{b}_{k+1:t}$, according to Equation (15.7). The forward–backward algorithm achieves this by storing the **f**s computed on the forward pass so that they are available during the backward pass. Another way to achieve this is with a single pass that propagates both **f** and **b** in the same direction. For example, the "forward" message **f** can be propagated backwards if we manipulate Equation (15.11) to work in the other direction:

$$\mathbf{f}_{1:t} = \alpha' (\mathbf{T}^\top)^{-1} \mathbf{O}_{t+1}^{-1} \mathbf{f}_{1:t+1}$$

The modified smoothing algorithm works by first running the standard forward pass to compute $\mathbf{f}_{t:t}$ (forgetting all the intermediate results), then running the backward pass for both **b** and **f** together, using them to compute the smoothed estimate at each step. Since only one copy of each message is needed, the storage requirements are constant (independent of t, the length of the sequence). There is, of course, one significant restriction on this algorithm: it requires that the transition matrix be invertible and that the sensor model have no zeroes—that is, every observation is possible in every state.

A second area where the matrix formulation reveals an improvement is in *online* smoothing with a fixed lag. The fact that smoothing can be done with constant space suggests that there should exist an efficient recursive algorithm for online smoothing—that is, one whose time complexity is independent of the length of the lag. Let us suppose that the lag is d—that is, we are smoothing at time slice $t - d$ where the current time is t. By Equation (15.7), we need to compute

$$\alpha \mathbf{f}_{1:t-d} \mathbf{b}_{t-d+1:t}$$

for slice $t-d$. Then, when a new observation arrives, we need to compute

$$\alpha\mathbf{f}_{1:t-d+1}\mathbf{b}_{t-d+2:t+1}$$

for slice $t-d+1$. How can this be done incrementally? First, we can compute $\mathbf{f}_{1:t-d+1}$ from $\mathbf{f}_{1:t-d}$ using the standard filtering process, Equation (15.4).

Computing the backward message incrementally is more tricky, because there is no simple relationship between the old backward message $\mathbf{b}_{t-d+1:t}$ and the new backward message $\mathbf{b}_{t-d+2:t+1}$. Instead, we will examine the relationship between the old backward message $\mathbf{b}_{t-d+1:t}$ and the backward message at the front of the sequence, $\mathbf{b}_{t+1:t}$. To do this, we apply Equation (15.12) d times:

$$\mathbf{b}_{t-d+1:t} = \left(\prod_{i=t-d+1}^{t} \mathbf{TO}_i\right)\mathbf{b}_{t+1:t} = \mathbf{B}_{t-d+1:t}\mathbf{1} \tag{15.13}$$

where the matrix $\mathbf{B}_{t-d+1:t}$ is the product of the sequence of $\mathbf{T}$ and $\mathbf{O}$ matrices. $\mathbf{B}$ can be thought of as a "transformation operator" that transforms a later backward message into an earlier one. A similar equation holds for the new backward messages *after* the next observation arrives:

$$\mathbf{b}_{t-d+2:t+1} = \left(\prod_{i=t-d+2}^{t+1} \mathbf{TO}_i\right)\mathbf{b}_{t+2:t+1} = \mathbf{B}_{t-d+2:t+1}\mathbf{1} \tag{15.14}$$

Examining the product expressions in Equations (15.13) and (15.14), we see that they have a simple relationship: to get the second product, "divide" the first product by the first element $\mathbf{TO}_{t-d+1}$ and multiply by the new last element $\mathbf{TO}_{t+1}$. In matrix language, then, there is a simple relationship between the old and new $\mathbf{B}$ matrices:

$$\mathbf{B}_{t-d+2:t+1} = \mathbf{O}_{t-d+1}^{-1}\mathbf{T}^{-1}\mathbf{B}_{t-d+1:t}\mathbf{TO}_{t+1} \tag{15.15}$$

This equation provides an incremental update for the $\mathbf{B}$ matrix, which in turn (through Equation (15.14)) allows us to compute the new backward message $\mathbf{b}_{t-d+2:t+1}$. The complete algorithm, which requires storing and updating $\mathbf{f}$ and $\mathbf{B}$, is shown in Figure 15.6.

15.4 Kalman Filters

Imagine watching a small bird flying through dense jungle foliage at dusk: you glimpse brief, intermittent flashes of motion; you try hard to guess where the bird is and where it will appear next so that you don't lose it. Or imagine you are a WWII radar operator peering at a faint, wandering blip that appears once every ten seconds on the screen. Or, going back further still, imagine you are Kepler trying to reconstruct the motions of the planets from a collection of highly inaccurate angular observations taken at irregular and imprecisely measured time intervals. In all these cases, you are trying to estimate the state (position and velocity, for example) of a physical system from noisy observations over time. The problem can be formulated as inference in a temporal probability model, where the transition model describes the physics of motion and the sensor model describes the measurement process.

```
function FIXED-LAG-SMOOTHING($e_t$, hmm, $d$) returns a probability distribution over $\mathbf{X}_{t-d}$
  inputs: $e_t$, the current evidence for time step $t$
          hmm, a hidden Markov model with $S \times S$ transition matrix T
          $d$, the length of the lag for smoothing
  static: $t$, the current time, initially 1
          f, a probability distribution, the forward message $\mathbf{P}(X_t|e_{1:t})$, initially PRIOR[hmm]
          B, the $d$-step backward transformation matrix, initially the identity matrix
          $e_{t-d:t}$, double-ended list of evidence from $t-d$ to $t$, initially empty
  local variables: $\mathbf{O}_{t-d}, \mathbf{O}_t$, diagonal matrices containing the sensor model information

  add $e_t$ to the end of $e_{t-d:t}$
  $\mathbf{O}_t \leftarrow$ diagonal matrix containing $\mathbf{P}(e_t|X_t)$
  if $t > d$ then
      $\mathbf{f} \leftarrow$ FORWARD($\mathbf{f}, e_t$)
      remove $e_{t-d-1}$ from the beginning of $e_{t-d:t}$
      $\mathbf{O}_{t-d} \leftarrow$ diagonal matrix containing $\mathbf{P}(e_{t-d}|X_{t-d})$
      $\mathbf{B} \leftarrow \mathbf{O}_{t-d}^{-1}\mathbf{T}^{-1}\mathbf{B}\mathbf{T}\mathbf{O}_t$
  else $\mathbf{B} \leftarrow \mathbf{B}\mathbf{T}\mathbf{O}_t$
  $t \leftarrow t+1$
  if $t > d$ then return NORMALIZE($\mathbf{f} \times \mathbf{B1}$) else return null
```

Figure 15.6 An algorithm for smoothing with a fixed time lag of d steps, implemented as an online algorithm that outputs the new smoothed estimate given the observation for a new time step.

This section describes the special representations and inference algorithms that have been developed to solve these sorts of problems; the method we will describe is called **Kalman filtering** after its inventor.

KALMAN FILTERING

Clearly, we will need several *continuous* variables to specify the state of the system. For example, the bird's flight might be specified by position (X, Y, Z) and velocity $(\dot{X}, \dot{Y}, \dot{Z})$ at each point in time. We will also need suitable conditional densities to represent the transition and sensor models; as in Chapter 14, we will use **linear Gaussian** distributions. This means that the next state $\mathbf{X}_{t+1}$ must be a linear function of the current state $\mathbf{X}_t$, plus some Gaussian noise. This turns out to be quite reasonable in practice. Consider, for example, the X–coordinate of the bird, ignoring the other coordinates for now. Let the interval between observations be Δ, and let us assume constant velocity; then the position update is given by

$$X_{t+\Delta} = X_t + \Delta\dot{X}$$

If we add Gaussian noise to account for variation in velocity and so on, then we have a linear Gaussian transition model:

$$P(X_{t+\Delta} = x_{t+\Delta}|X_t = x_t, \dot{X}_t = \dot{x}_t) = N(x_t + \Delta\dot{x}_t, \sigma)(x_{t+\Delta})$$

The Bayesian network structure for a system with position $\mathbf{X}_t$ and velocity $\dot{\mathbf{X}}_t$ is shown in Figure 15.7. Note that this is a very specific form of linear Gaussian model; the general form will be described later in this section, and covers a vast array of applications beyond the simple motion examples of the first paragraph. The reader may wish to consult Appendix A for some of the mathematical properties of Gaussian distributions; for our immediate purposes, the most important is that a **multivariate Gaussian** distribution for d variables is specified by a d-element mean $\boldsymbol{\mu}$ and a $d \times d$ covariance matrix $\boldsymbol{\Sigma}$.

MULTIVARIATE GAUSSIAN

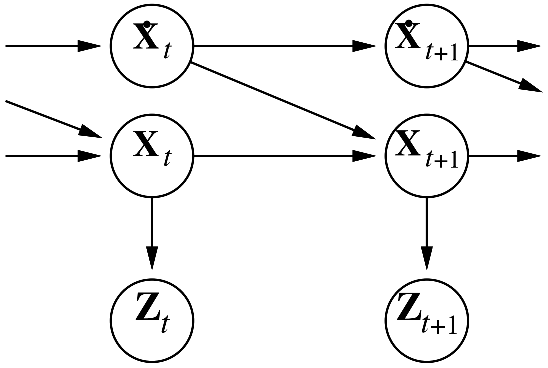


Figure 15.7 Bayesian network structure for a linear dynamical system with position $\mathbf{X}_t$, velocity $\dot{\mathbf{X}}_t$, and position measurement $\mathbf{Z}_t$.

Updating Gaussian distributions

We alluded in Chapter 14 to a key property of the linear Gaussian family of distributions: it remains closed under the standard Bayesian network operations. Here, we make this claim precise in the context of filtering in a temporal probability model. The required properties correspond to the two-step filtering calculation in Equation (15.4):

1. If the current distribution $\mathbf{P}(\mathbf{X}_t|\mathbf{e}_{1:t})$ is Gaussian and the transition model $\mathbf{P}(\mathbf{X}_{t+1}|\mathbf{x}_t)$ is linear Gaussian, then the one-step predicted distribution given by

$$\mathbf{P}(\mathbf{X}_{t+1}|\mathbf{e}_{1:t}) = \int_{\mathbf{x}_t} \mathbf{P}(\mathbf{X}_{t+1}|\mathbf{x}_t)P(\mathbf{x}_t|\mathbf{e}_{1:t})\,d\mathbf{x}_t \qquad (15.16)$$

is also a Gaussian distribution.

2. If the predicted distribution $\mathbf{P}(\mathbf{X}_{t+1}|\mathbf{e}_{1:t})$ is Gaussian and the sensor model $\mathbf{P}(\mathbf{e}_{t+1}|\mathbf{X}_{t+1})$ is linear Gaussian, then, after conditioning on the new evidence, the updated distribution

$$\mathbf{P}(\mathbf{X}_{t+1}|\mathbf{e}_{1:t+1}) = \alpha\mathbf{P}(\mathbf{e}_{t+1}|\mathbf{X}_{t+1})\mathbf{P}(\mathbf{X}_{t+1}|\mathbf{e}_{1:t}) \qquad (15.17)$$

is also a Gaussian distribution.

Thus, the FORWARD operator for Kalman filtering takes a Gaussian forward message $\mathbf{f}_{1:t}$, specified by a mean $\boldsymbol{\mu}_t$ and covariance matrix $\boldsymbol{\Sigma}_t$, and produces a new multivariate Gaussian forward message $\mathbf{f}_{1:t+1}$, specified by a mean $\boldsymbol{\mu}_{t+1}$ and covariance matrix $\boldsymbol{\Sigma}_{t+1}$. So, if we start with a Gaussian prior $\mathbf{f}_{1:0} = \mathbf{P}(\mathbf{X}_0) = N(\boldsymbol{\mu}_0, \boldsymbol{\Sigma}_0)$, filtering with a linear Gaussian model produces a Gaussian state distribution for all time.

This seems to be a nice, elegant result, but why is it so important? The reason is that, except for a few special cases such as this, *filtering with continuous or hybrid (discrete and* *continuous) networks generates state distributions whose representation grows without bound over time*. This is not easy to prove in general, but Exercise 15.5 shows what happens for a simple example.

A simple, one-dimensional example

We have said that the FORWARD operator for the Kalman filter maps a Gaussian into a new Gaussian. This translates into computing a new mean and covariance matrix from the previous mean and covariance matrix. Deriving the update rule in the general (multivariate) case requires rather a lot of linear algebra, so will stick to a very simple, univariate case for now; later we will give the results for the general case. Even for the univariate case, the calculations are somewhat tedious, but we feel they are worth seeing because the usefulness of the Kalman filter is tied so intimately to the mathematical properties of Gaussian distributions.

RANDOM WALK

The temporal model we will consider describes a **random walk** of a single continuous state variable X_t with a noisy observation Z_t. An example might be the "consumer confidence" index, which can be modelled as undergoing a random, Gaussian-distributed change each month and is measured by a random consumer survey that also introduces Gaussian sampling noise. The prior distribution is assumed to be Gaussian with variance σ_0^2:

$$P(x_0) = \alpha e^{-\frac{1}{2}\left(\frac{(x_0-\mu_0)^2}{\sigma_0^2}\right)}$$

(For simplicity, we will use the same symbol α for all normalizing constants in this section.) The transition model simply adds a Gaussian perturbation of constant variance σ_x^2 to the current state:

$$P(x_{t+1}|x_t) = \alpha e^{-\frac{1}{2}\left(\frac{(x_{t+1}-x_t)^2}{\sigma_x^2}\right)}$$

and the sensor model assumes Gaussian noise with variance σ_z^2:

$$P(z_t|x_t) = \alpha e^{-\frac{1}{2}\left(\frac{(z_t-x_t)^2}{\sigma_z^2}\right)}$$

Now, given the prior $P(X_0)$, we can compute the one-step predicted distribution using Equation (15.16):

$$\begin{aligned} P(x_1) &= \int_{-\infty}^{\infty} P(x_1|x_0)P(x_0)\,dx_0 = \alpha \int_{-\infty}^{\infty} e^{-\frac{1}{2}\left(\frac{(x_1-x_0)^2}{\sigma_x^2}\right)} e^{-\frac{1}{2}\left(\frac{(x_0-\mu_0)^2}{\sigma_0^2}\right)}\,dx_0 \\ &= \alpha \int_{-\infty}^{\infty} e^{-\frac{1}{2}\left(\frac{\sigma_0^2(x_1-x_0)^2+\sigma_x^2(x_0-\mu_0)^2}{\sigma_0^2\sigma_x^2}\right)}\,dx_0 \end{aligned}$$

This integral looks rather hairy. The key to progress is to notice that the exponent is the sum of two expressions that are *quadratic* in x_0, and hence is itself a quadratic in x_0. A simple trick known as **completing the square** allows the rewriting of any quadratic $ax_0^2 + bx_0 + c$ as the sum of a squared term $a(x_0 - \frac{-b}{2a})^2$ and a residual term $c - \frac{b^2}{4a}$ that is independent of x_0. The residual term can be taken outside the integral, giving us

COMPLETING THE SQUARE

$$P(x_1) = \alpha e^{-\frac{1}{2}\left(c-\frac{b^2}{4a}\right)} \int_{-\infty}^{\infty} e^{-\frac{1}{2}\left(a(x_0-\frac{-b}{2a})^2\right)}\,dx_0$$

Now the integral is just the integral of a Gaussian over its full range, which is simply 1. Thus, we are left with just the residual term from the quadratic.

The second key step is to notice that the residual term has to be a quadratic in x_1; in fact, after simplification, we obtain

$$P(x_1) = \alpha e^{-\frac{1}{2}\left(\frac{(x_1-\mu_0)^2}{\sigma_0^2+\sigma_x^2}\right)}$$

That is, the one-step predicted distribution is a Gaussian with the same mean μ_0 and a variance equal to the sum of the original variance σ_0^2 and the transition variance σ_x^2. A momentary exercise of intuition reveals that this is intuitively reasonable.

To complete the update step, we need to condition on the observation at the first time step, namely z_1. From Equation (15.17), this is given by

$$\begin{aligned} P(x_1|z_1) &= \alpha P(z_1|x_1)P(x_1) \\ &= \alpha e^{-\frac{1}{2}\left(\frac{(z_1-x_1)^2}{\sigma_z^2}\right)} e^{-\frac{1}{2}\left(\frac{(x_1-\mu_0)^2}{\sigma_0^2+\sigma_x^2}\right)} \end{aligned}$$

Once again, we combine the exponents and complete the square (Exercise 15.6), obtaining

$$P(x_1|z_1) = \alpha e^{-\frac{1}{2}\left(\frac{(x_1 - \frac{(\sigma_0^2+\sigma_x^2)z_1+\sigma_z^2\mu_0}{\sigma_0^2+\sigma_x^2+\sigma_z^2})^2}{(\sigma_0^2+\sigma_x^2)\sigma_z^2/(\sigma_0^2+\sigma_x^2+\sigma_z^2)}\right)} \qquad (15.18)$$

Thus, after one update cycle, we have a new Gaussian distribution for the state variable.

From the Gaussian formula in Equation (15.18), we can see that the new mean and standard deviation can be calculated from the old mean and standard deviation as follows:

$$\begin{aligned} \mu_{t+1} &= \frac{(\sigma_t^2+\sigma_x^2)z_{t+1}+\sigma_z^2\mu_t}{\sigma_t^2+\sigma_x^2+\sigma_z^2} \\ \sigma_{t+1}^2 &= \frac{(\sigma_t^2+\sigma_x^2)\sigma_z^2}{\sigma_t^2+\sigma_x^2+\sigma_z^2} \end{aligned} \qquad (15.19)$$

Figure 15.8 shows one update cycle for particular values of the transition and sensor models.

The preceding pair of equations plays exactly the same role as the general filtering equation (15.4) or the HMM filtering equation (15.11). Because of the special nature of Gaussian distributions, however, the equations have some interesting additional properties. First, we can interpret the calculation for the new mean μ_{t+1} as simply a *weighted mean* of the new observation z_{t+1} and the old mean μ_t. If the observation is unreliable, then σ_z^2 is large and we pay more attention to the old mean; if the old mean is unreliable (σ_t^2 is large) or the process is highly unpredictable (σ_x^2 is large), then we pay more attention to the observation. Second, notice that the update for the variance σ_{t+1}^2 is *independent of the observation*. We can therefore compute in advance what the sequence of variance values will be. Third, the sequence of variance values quickly converges to a fixed value that depends only on σ_x^2 and σ_z^2, thereby substantially simplifying the subsequent calculations (see Exercise 15.7).

The general case

The preceding derivation, painful as it was, illustrates the key property of Gaussian distributions that allows Kalman filtering to work: the fact that the exponent is a quadratic form.

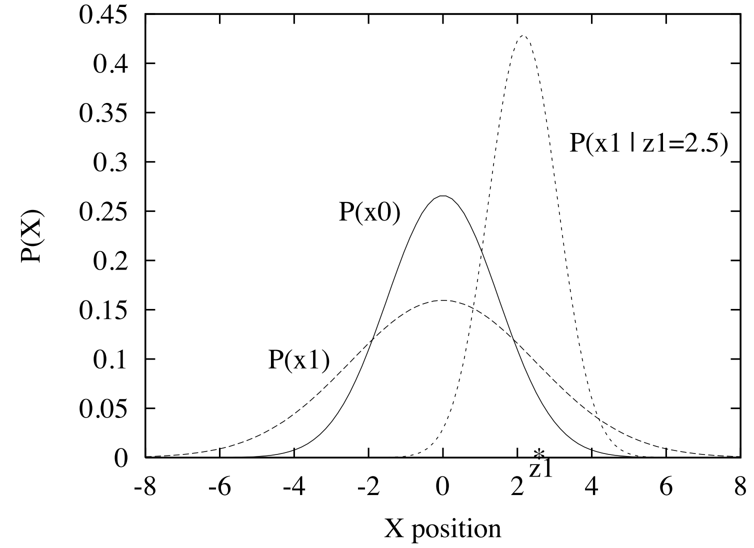


Figure 15.8 Stages in the Kalman filter update cycle for a random walk with a prior given by $\mu_0 = 0.0$ and $\sigma_0 = 1.0$, transition noise given by $\sigma_x = 2.0$, sensor noise given by $\sigma_z = 1.0$, and a first observation $z_1 = 2.5$. Notice how the prediction $P(x_1)$ is flattened out, relative to $P(x_0)$, by the transition noise. Notice also that the mean of the posterior $P(x_1|z_1)$ is slightly to the left of the observation z_1 because the mean is a weighted average of the prediction and the observation.

This is true not just for the univariate case. The full multivariate Gaussian distribution has the form

$$N(\boldsymbol{\mu},\boldsymbol{\Sigma})(\mathbf{x}) = \alpha e^{-\frac{1}{2}\left((\mathbf{x}-\boldsymbol{\mu})^\top\boldsymbol{\Sigma}^{-1}(\mathbf{x}-\boldsymbol{\mu})\right)}$$

Multiplying out the terms in the exponent, it is clear that the exponent is also a quadratic function of the random variables x_i in $\mathbf{x}$. As in the univariate case, the filtering update preserves the Gaussian nature of the state distribution.

Let us first define the general temporal model used with Kalman filtering. Both the transition model and the sensor model allow for a *linear* transformation with additive Gaussian noise. Thus, we have

$$\begin{aligned} P(\mathbf{x}_{t+1}|\mathbf{x}_t) &= N(\mathbf{F}\mathbf{x}_t,\boldsymbol{\Sigma}_x)(\mathbf{x}_{t+1}) \\ P(\mathbf{z}_t|\mathbf{x}_t) &= N(\mathbf{H}\mathbf{x}_t,\boldsymbol{\Sigma}_z)(\mathbf{z}_t) \end{aligned} \qquad (15.20)$$

where $\mathbf{F}$ and $\boldsymbol{\Sigma}_x$ are matrices describing the linear transition model and transition noise covariance, and $\mathbf{H}$ and $\boldsymbol{\Sigma}_z$ are the corresponding matrices for the sensor model. Now the update equations for the mean and covariance, in their full, hairy horribleness, are as follows:

$$\begin{aligned} \boldsymbol{\mu}_{t+1} &= \mathbf{F}\boldsymbol{\mu}_t + \mathbf{K}_{t+1}(\mathbf{z}_{t+1} - \mathbf{H}\mathbf{F}\boldsymbol{\mu}_t) \\ \boldsymbol{\Sigma}_{t+1} &= (\mathbf{I} - \mathbf{K}_{t+1})(\mathbf{F}\boldsymbol{\Sigma}_t\mathbf{F}^\top + \boldsymbol{\Sigma}_x) \end{aligned} \qquad (15.21)$$

KALMAN GAIN MATRIX

where $\mathbf{K}_{t+1} = (\mathbf{F}\boldsymbol{\Sigma}_t\mathbf{F}^\top + \boldsymbol{\Sigma}_x)\mathbf{H}^\top(\mathbf{H}(\mathbf{F}\boldsymbol{\Sigma}_t\mathbf{F}^\top + \boldsymbol{\Sigma}_x) + \boldsymbol{\Sigma}_z)^{-1}$ is called the **Kalman gain matrix**. Believe it or not, these equations make some intuitive sense. For example, consider the update for the mean state estimate $\boldsymbol{\mu}$. The term $\mathbf{F}\boldsymbol{\mu}_t$ is the *predicted* state at $t+1$, so $\mathbf{HF}\boldsymbol{\mu}_t$ is the *predicted* observation. Therefore the term $\mathbf{z}_{t+1} - \mathbf{HF}\boldsymbol{\mu}_t$ represents the error in the predicted observation. This is multiplied by $\mathbf{K}_{t+1}$ to correct the predicted state; therefore $\mathbf{K}_{t+1}$ is a measure of *how seriously to take the new observation* relative to the prediction. As in Equation (15.19), we also have the property that the variance update is independent of the observations. The sequence of values for $\mathbf{\Sigma}_t$ and $\mathbf{K}_t$ can therefore be computed offline, and the actual calculations required during online tracking are quite modest.

To illustrate these equations at work, we have applied them to the problem of tracking an object moving on the X–Y plane. The state variables are $\mathbf{X} = (X, Y, \dot{X}, \dot{Y})^{\top}$ so $\mathbf{F}$, $\mathbf{\Sigma}_x$, $\mathbf{H}$, and $\mathbf{\Sigma}_z$ are 4×4 matrices. Figure 15.9(a) shows the true trajectory, a series of noisy observations, and the trajectory estimated by Kalman filtering, along with the covariances indicated by the one-standard-deviation contours. The filtering process does a reasonably good job of tracking the actual motion, and, as expected, the variance quickly reaches a fixed point.

As one might expect, one can also derive equations for *smoothing* as well as filtering with linear Gaussian models. The smoothing results are shown in Figure 15.9(b). Notice how the variance in the position estimate is sharply reduced, except at the ends of the trajectory (why?); and that the estimated trajectory is much smoother.

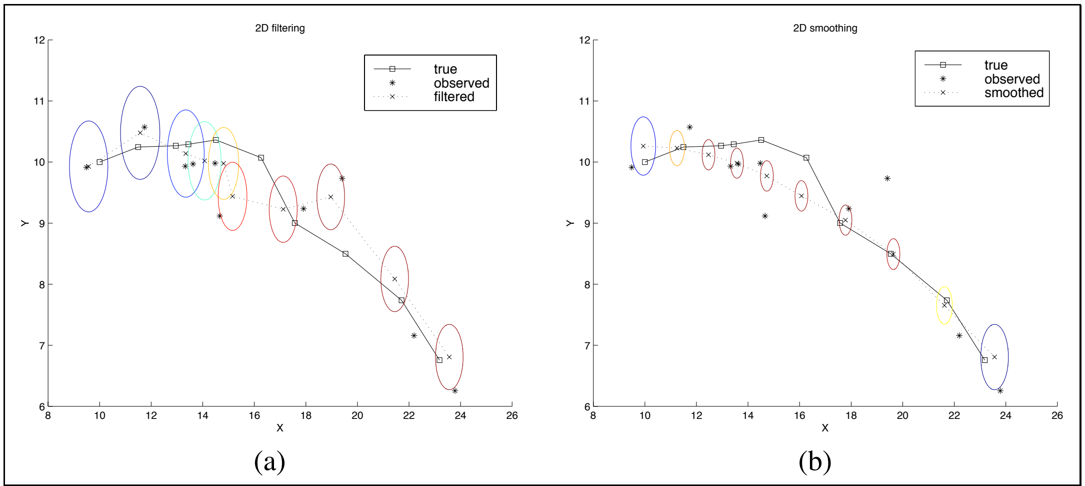


Figure 15.9 (a) Results of Kalman filtering for an object moving on the X–Y plane, showing the true trajectory (left-to-right), a series of noisy observations, and the trajectory estimated by Kalman filtering. (b) The results of Kalman smoothing for the same observation sequence.

Applicability of Kalman filtering

The Kalman filter and its elaborations are used in a vast array of applications. The “classical” application is in radar tracking of aircraft and missiles. Related applications include acoustic tracking of submarines and ground vehicles and visual tracking of vehicles and people. In a slightly more esoteric vein, Kalman filters are used to reconstruct particle trajectories from bubble chamber photographs and ocean currents from satellite surface measurements. The range of application is much larger than just the tracking of motion: any system characterized by continuous state variables and noisy measurements will do. Such systems include pulp mills, chemical plants, nuclear reactors, plant ecosystems, and national economies.

The fact that Kalman filtering can be applied to a system does not mean that the results will be valid or useful. The assumptions made—linear Gaussian transition and sensor models—are very strong. The **extended Kalman filter** or EKF attempts to overcome nonlinearities in the system being modelled. A system is nonlinear if the transition model cannot be described as a matrix multiplication of the state vector, as in Equation (15.20). The EKF works by modelling the system as *locally* linear in $\mathbf{x}_t$ in the region of $\mathbf{x}_t = \boldsymbol{\mu}_t$, the mean of the current state distribution. This works well for smooth, well-behaved systems, and allows the tracker to maintain and update a Gaussian state distribution that is a reasonable approximation to the true posterior.

EXTENDED KALMAN FILTER

What does it mean for a system to be "unsmooth" or "poorly behaved"? Technically, this means that there is significant nonlinearity in system response within the region that is "close" (according to the covariance $\boldsymbol{\Sigma}_t$) to the current mean $\boldsymbol{\mu}_t$. To understand this in nontechnical terms, consider the example of trying to track a bird as it flies through the jungle. The bird appears to be heading at high speed straight for a tree-trunk. The Kalman filter, whether regular or extended, can only make a Gaussian prediction of the location of the bird, and the mean of this Gaussian will be centered on the trunk, as shown in Figure 15.10(a). A reasonable model of the bird, on the other hand, would predict evasive action to one side or the other, resulting in the prediction shown in Figure 15.10(b). Such a model is highly nonlinear because the bird's decision varies sharply depending on its precise location relative to the trunk.

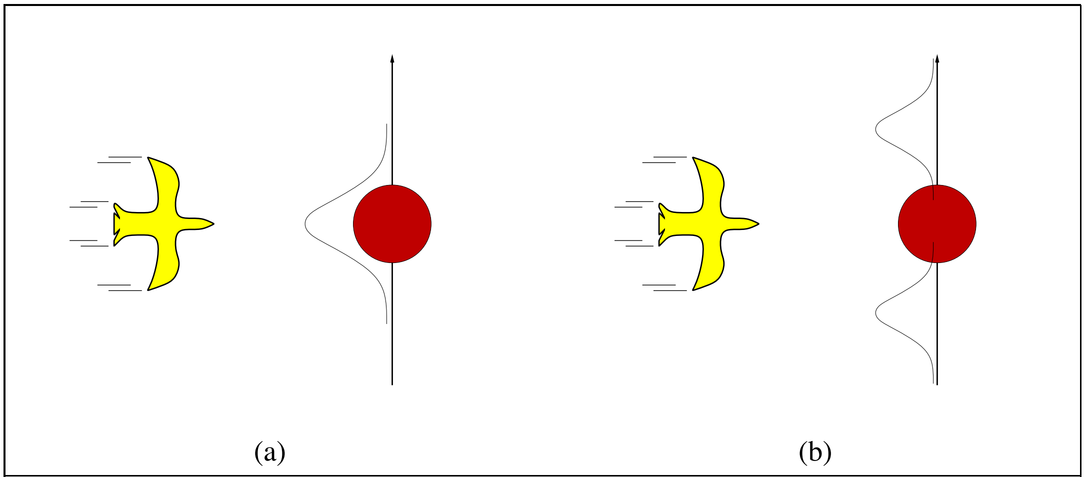


Figure 15.10 A bird flying toward a tree (top views). (a) A Kalman filter will predict the location of the bird using a single Gaussian centered on the obstacle. (b) A more realistic model allows for the bird's evasive action, predicting that it will fly to one side or the other.

SWITCHING KALMAN FILTER

In order to handle such examples, we clearly need a more expressive language for representing the behavior of the system being modelled. Within the control theory community, where problems such as evasive maneuvering by aircraft raise the same kinds of difficulties, the standard solution is the **switching Kalman filter**. In this approach, multiple Kalman filters run in parallel, each using a different model of the system—for example, one for straight flight, one for sharp left turns, one for sharp right turns. A weighted sum of predictions is used, where the weight depends on how well each filter fits the current data. We will see in the next section that this is simply a special case of the general dynamic Bayesian network model, obtained in this case by adding a discrete "maneuver" state variable to the network shown in Figure 15.7. Switching Kalman filters are discussed further in Exercise 15.5.

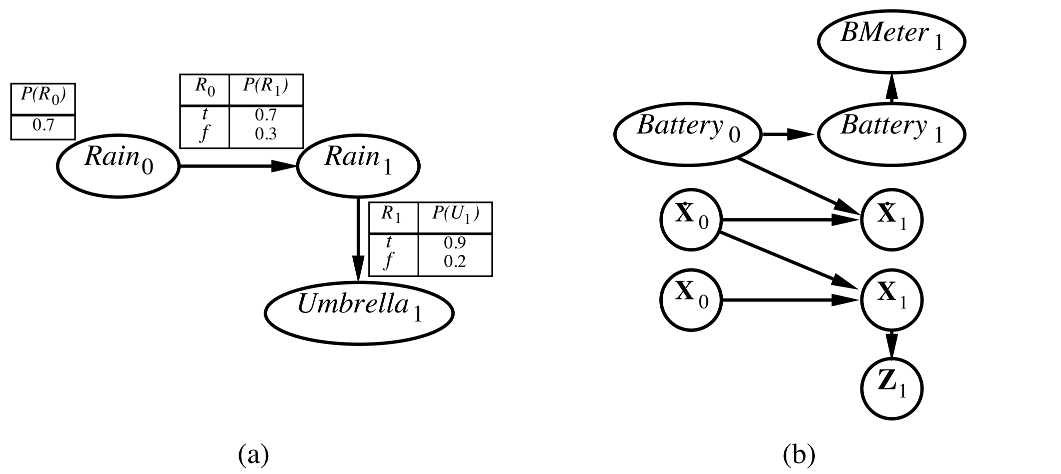
15.5 Dynamic Bayesian Networks

DYNAMIC BAYESIAN NETWORK

A **dynamic Bayesian network** or **DBN** is a Bayesian network that represents a temporal probability model of the kind described in Section 15.1. We have already seen examples of DBNs: the umbrella network in Figure 15.2 and the Kalman filter network in Figure 15.7. In general, each slice of a DBN can have any number of state variables $\mathbf{X}_t$ and evidence variables $\mathbf{E}_t$. For simplicity, we will assume that the variables and their links are exactly replicated from slice to slice, and that the DBN represents a first-order Markov process, so that each variable can have parents only in its own slice or the immediately preceding slice.

It should be clear that every hidden Markov model can be represented as a DBN with a single state variable and a single evidence variable. It is also the case that every discrete-variable DBN can be represented as an HMM: as explained in Section 15.3, we can combine all the state variables in the DBN into a single state variable whose values are all possible tuples of values of the individual state variables. Now if every HMM is a DBN and every DBN can be translated into an HMM, what's the difference? The difference is that, *by decomposing the state of a complex system into its constituent variables, the DBN is able to take advantage of* sparseness *in the temporal probability model.* Suppose, for example, that a DBN has 20 Boolean state variables, each of which has three parents in the preceding slice. Then the DBN transition model has $20 \times 2^3 = 160$ probabilities, whereas the corresponding HMM has 2^{20} states and therefore 2^{40}, or roughly a trillion, probabilities in the transition matrix. This bad for at least two reasons: first, the HMM itself requires much more space; second, the huge transition matrix makes HMM inference much more expensive; and third, the problem of learning such a huge number of parameters makes the pure HMM model unsuitable for large problems. The relationship between DBNs and HMMs is roughly analogous to the relationship between ordinary Bayesian networks and full tabulated joint distributions.

We have already explained that every Kalman filter model can be represented in a DBN with continuous variables and linear Gaussian conditional distributions (Figure 15.7). It should be clear from the discussion at the end of the preceding section that *not* every DBN can be represented by a Kalman filter model. In a Kalman filter, the current state distribution is always a single multivariate Gaussian distribution—that is, a single "bump" in a particular



$P(R_0)$
0.7

R_0	$P(R_1)$
t	0.7
f	0.3

R_1	$P(U_1)$
t	0.9
f	0.2

Figure 15.11 (a) Specification of the prior, transition model, and sensor model for the umbrella DBN. All subsequent slices are assumed to be copies of slice 1. (b) A simple DBN for robot motion in the X–Y plane.

location. DBNs, on the other hand, can handle arbitrary distributions. For many real-world applications, this flexibility is essential. Consider, for example, the current location of my keys. They might be in my pocket, on the bedside table, on the kitchen counter, or dangling from the front door. A single Gaussian bump that included all these places would have to allocate significant probability to the keys being in mid-air in the front hall. Aspects of the real world such as purposive agents, obstacles, and pockets introduce "nonlinearities" and "discontinuities" that necessitate complex combinations of discrete and continuous variables in order to get reasonable models.

Constructing DBNs

To construct a DBN, one must specify three kinds of information: the prior distribution over the state variables, $\mathbf{P}(\mathbf{X}_0)$; the transition model $\mathbf{P}(\mathbf{X}_{t+1}|\mathbf{X}_t)$; and the sensor model $\mathbf{P}(\mathbf{E}_t|\mathbf{X}_t)$. To specify the transition and sensor models, one must also specify the topology of the connections between successive slices and between the state and evidence variables. Because the transition and sensor models are assumed to be stationary—i.e., the same for all t—it is most convenient simply to specify them for the first slice. For example, the complete DBN specification for the umbrella world is given by the three-node network shown in Figure 15.11(a). From this specification, the complete (semi-infinite) DBN can be constructed as needed by copying the first slice.

Let us now consider a more interesting example: monitoring a battery-powered robot moving in the X–Y plane, as introduced in Section 15.1. First, we need state variables, which will include both $\mathbf{X}_t = (X_t, Y_t)$ for position and $\dot{\mathbf{X}}_t = (\dot{X}_t, \dot{Y}_t)$ for velocity. We will assume some method of measuring position—perhaps a fixed camera or onboard GPS (Global Positioning System)—yielding measurements $\mathbf{Z}_t$. The position at the next time step depends on the current position and velocity, as in the standard Kalman filter model. The velocity at the next step depends on the current velocity and the state of the battery. We add $Battery_t$ to represent the actual battery charge level, which has as parents the previous battery level and the velocity, and we add $BMeter_t$, which measures the battery charge level. This gives us the basic model shown in Figure 15.11(b).

It is worth looking in more depth at the nature of the sensor model for $BMeter_t$. Let us suppose, for simplicity, that both $Battery_t$ and $BMeter_t$ can take on discrete values 0 through 5—rather like the battery meter on a typical laptop computer. If the meter is always accurate, then the CPT $\mathbf{P}(BMeter_t|Battery_t)$ should have probabilities of 1.0 "along the diagonal" and probabilities of 0.0 elsewhere. In reality, noise always creeps into measurements. For continuous measurements, a Gaussian distribution with a small variance might be used instead.[4] For our discrete variables, we can approximate a Gaussian using a distribution in which the probability of error drops off in the appropriate way, so that the probability of a large error is very small. We will use the term **Gaussian error model** to cover both the continuous and discrete versions.

GAUSSIAN ERROR MODEL

Anyone with hands-on experience of robotics, computerized process control, or other forms of automatic sensing will readily testify to the fact that small amounts of measurement noise are often the least of one's problems. Real sensors *fail*. When a sensor fails, it does not necessarily send a signal saying, "Oh, by the way, the data I'm about to send you is a load of nonsense." Instead, it simply sends the nonsense. The simplest kind of failure is called a **transient failure**, where the sensor occasionally decides to send some nonsense. For example, the battery level sensor might have a habit of sending a zero when someone bumps the robot, even if the battery is fully charged.

TRANSIENT FAILURE

Let's see what happens when a transient failure occurs with a Gaussian error model that doesn't accommodate such failures. Suppose, for example, that the robot is sitting quietly and observes twenty consecutive battery readings of 5. Then the battery meter has a temporary seizure and the next reading is $BMeter_{21} = 0$. What will the simple Gaussian error model lead us to believe about $Battery_{21}$? According to Bayes' rule, the answer depends on both the sensor model $\mathbf{P}(BMeter_{21} = 0|Battery_{21})$ and the prediction $\mathbf{P}(Battery_{21}|BMeter_{1:20})$. If the probability of a large sensor error is significantly less likely than the probability of a transition to $Battery_{21} = 0$, even if the latter is very unlikely, then the posterior distribution will assign high probability to the battery being empty. A second reading of zero at $t = 22$ will make this conclusion almost certain. If the transient failure then disappears and the reading returns to 5 from $t = 23$ onwards, the estimate for the battery level will quickly return to 5, as if by magic. This course of events is illustrated in the upper curve of Figure 15.12(a), which shows the expected value of $Battery_t$ over time using a discrete Gaussian error model.

Despite the recovery, there is a time ($t = 22$) when the robot is convinced its battery is empty; presumably, then, it should send out a mayday signal and shut down. Alas, its oversimplified sensor model has led it astray. How can this be fixed? Consider a familiar

[4] Strictly speaking, a Gaussian distribution is problematic because it assigns nonzero probability to large negative charge levels. The **beta distribution** is sometimes a better choice for a variable whose range is restricted.

example from everyday human driving: on sharp curves or steep hills, one's "fuel tank empty" warning light sometimes turns on. Rather than looking for the emergency phone, one simply recalls that the fuel gauge sometimes gives a very large error when the fuel is sloshing around in the tank. The moral of the story is the following: *in order for the system to handle sensor failure properly, the sensor model must include the possibility of failure.*

The simplest kind of failure model for a sensor allows a certain probability that the sensor will return some completely incorrect value, regardless of the true state of the world. For example, if the battery meter fails by returning 0, we might say that

$$P(BMeter_t = 0 | Battery_t = 5) = 0.03$$

TRANSIENT FAILURE MODEL

which is presumably much larger than the probability assigned by the simple Gaussian error model. Let's call this the **transient failure model**. How does it help when we are faced with a reading of 0? Provided that the *predicted* probability of an empty battery, according to the readings so far, is much less than 0.03, then the best explanation of the observation $BMeter_{21} = 0$ is that the sensor has temporarily failed. Intuitively, we can think of the belief about the battery level as having a certain amount of "inertia" that helps to overcome temporary blips in the meter reading. The upper curve in Figure 15.12(b) shows that the transient failure model can handle transient failures without a catastrophic change in beliefs.

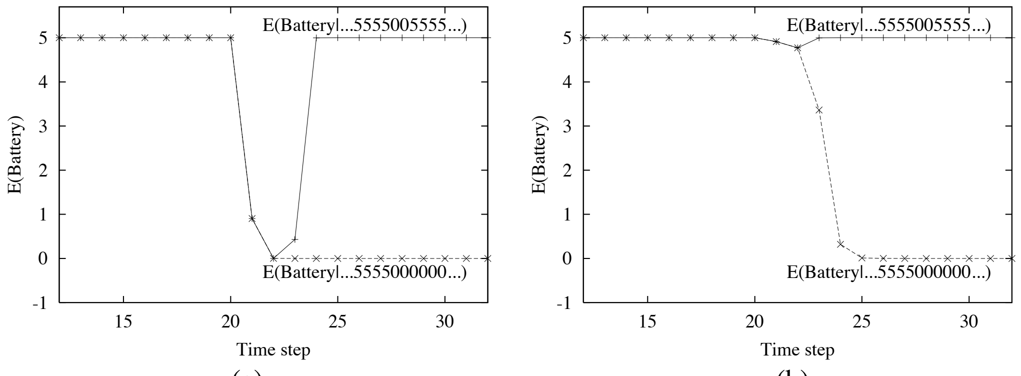
So much for temporary blips. What about a persistent sensor failure? Sadly, failures of this kind are all too common. If the sensor returns 20 readings of 5 followed by 20 readings of 0, then the transient sensor failure model described in the preceding paragraph will result in the robot gradually coming to believe that its battery is empty, when in fact it may be that the meter has failed. The lower curve in Figure 15.12(b) shows the belief "trajectory" for this case. By $t = 25$—five readings of 0—the robot is convinced that its battery is empty. Obviously, we would prefer the robot to believe that its battery meter is broken—if indeed this is the more likely event.

PERSISTENT FAILURE MODEL

Unsurprisingly, to handle persistent failure we will need a **persistent failure model** that describes how the sensor behaves under normal conditions and after failure. To do this, we need to augment the hidden state of the system with an additional variable, say $BMBroken$, that describes the status of the battery meter. The persistence of failure must be modelled by an arc linking $BMBroken_0$ to $BMBroken_1$. This **persistence arc** has a CPT that gives a small probability of failure in any given time step, say 0.001, but specifies that the sensor stays broken once it breaks. When the sensor is OK, the sensor model for $BMeter$ is identical to the transient failure model; when the sensor is broken, it says $BMeter$ is always 0, regardless of the actual battery charge.

PERSISTENCE ARC

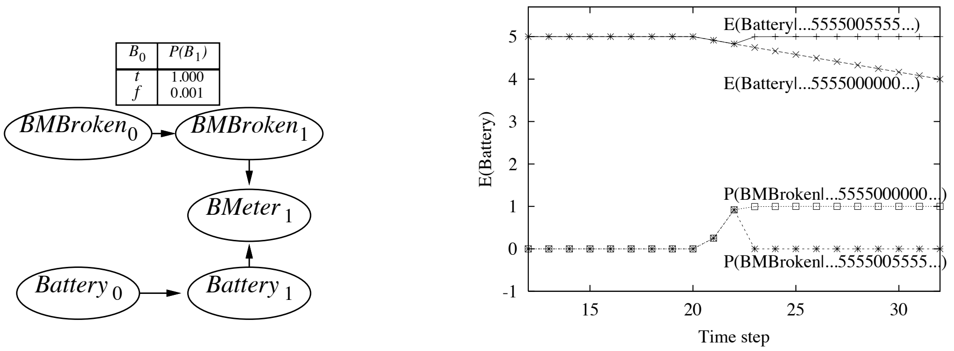
The persistent failure model for the battery sensor is shown in Figure 15.13(a). Its performance on the two data sequences (temporary blip and persistent failure) is shown in Figure 15.13(b). There are several things to notice about these curves. First, in the case of the temporary blip, the probability that the sensor is broken rises significantly after the second 0 reading, but immediately drops back to zero once a 5 is observed. Second, in the case of persistent failure, the probability that the sensor is broken rises quickly to almost 1 and stays there. Finally, once the sensor is known to be broken, the robot can only assume that its battery discharges at the "normal" rate, as shown by the gradually descending level of



(a)

(b)

Figure 15.12 (a) Upper curve: trajectory of the expected value of $Battery_t$ for an observation sequence consisting of all 5s except for 0s at $t=21$ and $t=22$, using a simple Gaussian error model. Lower curve: trajectory when the observation remains at 0 from $t=21$ onwards. (b) The same experiment run using the transient failure model. Notice that the transient failure is handled well but the persistent failure results in excessive pessimism.



(a)

(b)

Figure 15.13 (a) A DBN fragment showing the sensor status variable required for modelling persistent failure of the battery sensor. (b) Upper curves: trajectories of the expected value of $Battery_t$ for the "transient failure" and "permanent failure" observations sequences. Lower curves: probability trajectories for $BMBroken$ given the two observation sequences.

$E(Battery_t|\ldots)$. A more refined model would include the influence of the robot's activities on the battery level, which we have so far ignored.

So far, we have only scratched the surface of the problem of representing complex processes. The variety of transition models is huge, encompassing topics as disparate as modelling of the human endocrine system and modelling multiple vehicles driving on a freeway. Sensor modelling is also a vast subfield in itself, but even subtle phenomena, such as sensor drift, sudden decalibration, and the effects of exogenous conditions (such as weather) on sensor readings, can be handled by explicit representation within dynamic Bayesian networks.

Exact inference in DBNs

Having sketched some ideas for representing complex processes as DBNs, we now turn to the question of inference. In a sense, this question has already been answered: dynamic Bayesian networks *are* Bayesian networks, and we already have algorithms for inference in Bayesian networks. Given a sequence of observations, one can construct the full Bayesian network representation of a DBN by replicating slices until the network is large enough to accommodate the observations, as in Figure 15.14. This is called **unrolling**. (Technically, the DBN is equivalent to the semi-infinite network obtained by unrolling for ever. Slices added beyond the last observation have no effect on inferences within the observation period and can be omitted.) Once the DBN is unrolled, one can use any of the inference algorithms—variable elimination, join-tree methods, and so on—described in Chapter 14.

UNROLLING

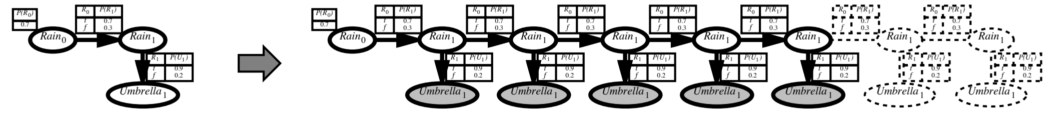


Figure 15.14 Unrolling a dynamic Bayesian network: slices are replicated to accommodate the observation sequence (shaded nodes). Further slices have no effect on inferences within the observation period.

Unfortunately, a naive application of unrolling would not be particularly efficient. If we want to perform filtering or smoothing with a long sequence of observations $\mathbf{e}_{1:t}$, the unrolled network would require $O(t)$ space and thus grows without bound as more observations are added. Moreover, if we simply run the inference algorithm anew each time an observation is added, the inference time per update will also increase as $O(t)$.

Looking back to Section 15.2, we see that constant time and space per filtering update can be achieved if the computation can be done in a recursive fashion. Essentially, the filtering update in Equation (15.4) works by *summing out* the state variables of the previous time step to get the distribution for the new time step. Summing out variables is exactly what the **variable elimination** (Figure 14.10) algorithm does, and it turns out that running variable elimination with the variables in temporal order exactly mimics the operation of the recursive filtering update in Equation (15.4). The modified algorithm keeps at most two slices in memory at any one time: starting with slice 0, we add slice 1, then sum out slice 0, then add slice 2, then sum out slice 1, and so on. In this way, we can achieve constant space and time per filtering update. (The same performance can be achieved by making suitable modifications to the join tree algorithm.) Exercise 15.10 asks you to verify this fact for the umbrella network.

So much for the good news; now for the bad news. It turns out that the "constant" for the per-update time and space complexity is, in almost all cases, exponential in the number of state variables. What happens is that as the variable elimination proceeds, the factors grow to include all the state variables (or, more precisely, all those state variables that have parents in the previous time slice). The maximum factor size is $O(d^{n+1})$ and the update cost is $O(d^{n+2})$.

This is much less than the cost of HMM updating, which is $O(d^{2n})$, but it is still infeasible for large numbers of variables. This grim fact is somewhat hard to accept. What it means is that *even though we can use DBNs to* represent *very complex temporal processes with many sparsely connected variables, we cannot* reason *efficiently and exactly about those processes*. The DBN model itself, which represents the prior joint distribution over all the variables, is factorable into its constituent CPTs, but the posterior joint distribution conditioned on an observation sequence—that is, the forward message—is generally *not* factorable. So far, no-one has found a way around this problem, despite the fact that many important areas of science and engineering would benefit enormously from its solution. Thus, we must fall back on approximate methods.

Approximate inference in DBNs

Chapter 14 described two approximation algorithms: likelihood weighting (Figure 14.14) and Markov chain Monte Carlo (MCMC, Figure 14.15). Of the two, the former is most easily adapted to the DBN context. We will see, however, that several improvements are required over the standard likelihood weighting algorithm before a practical method emerges.

Recall that likelihood weighting works by sampling the nonevidence nodes of the network in topological order, weighting each sample by the likelihood it accords to the observed evidence variables. As with the exact algorithms, we could apply likelihood weighting directly to an unrolled DBN, but this would suffer from the same problems in terms of increasing time and space requirements per update as the observation sequence grows. The problem is that the standard algorithm runs each sample in turn all the way through the network. Instead, we can simply run all N samples together through the DBN one slice at a time. The modified algorithm fits the general pattern of filtering algorithms, with the set of N samples as the forward message. The first key innovation, then, is to *use the samples themselves as an approximate representation of the current state distribution*. This meets the requirement of a "constant" time per update, although the constant depends on the number of samples required to maintain a reasonable approximation to the true posterior distribution. There is also no need to unroll the DBN, because we need only the current slice and the next slice in memory.

In our discussion of likelihood weighting in Chapter 14, we pointed out that the algorithm's accuracy suffers if the evidence variables are "downstream" of the variables being sampled, because in that case the samples are generated without any influence from the evidence. Looking at the typical structure of a DBN—say, the umbrella DBN in Figure 15.14—we see that indeed the early state variables will be sampled without the benefit of the later evidence. In fact, looking more carefully, we see that *none* of the state variables has *any* evidence variables among its ancestors! Hence, although the weight of each sample will depend on the evidence, the actual set of samples generated will be *completely independent* of the evidence. For example, even if the boss brings in the umbrella every day, the sampling process may still hallucinate endless days of sunshine. What this means in practice is that the fraction of samples that remain reasonably close to the actual series of events drops exponentially with t, the length of the observation sequence; in other words, to maintain a given level of accuracy, we need to increase the number of samples exponentially with t. Figure 15.15(a) shows some experimental results for likelihood weighting applied to the umbrella network. Clearly we need a better solution.

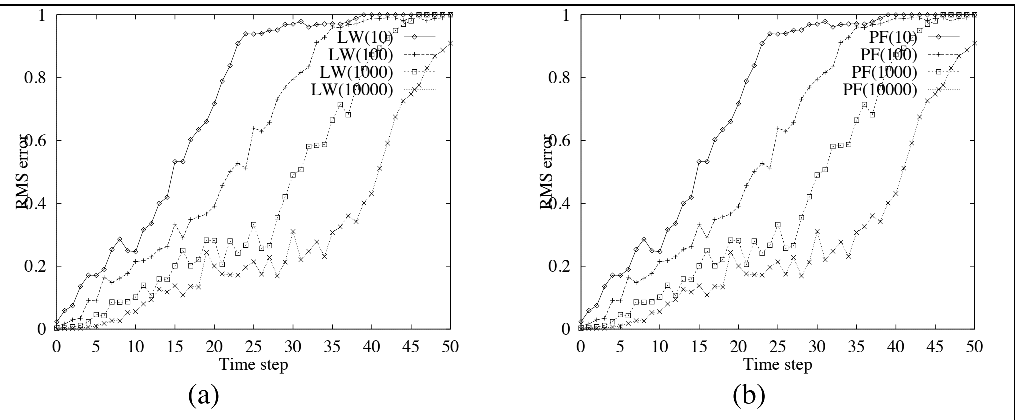


Figure 15.15 (a) Performance of likelihood weighting on the umbrella DBN, showing the root-mean-squared error in the probability of rain as a function of time step, averaged over 100 observation sequences generated from the model itself. (b) Performance of particle filtering on the same observation sequences.[[real data to be provided]]

The second key innovation is to *focus the set of samples on the high-probability regions of the state space*. This can be done by throwing away samples that have very low weight, according to the observations, while multiplying those that have high weight. In this way, the population of samples will stay reasonably close to reality. If we think of samples as a resource for modelling the posterior distribution, then it makes sense to use more samples in regions of the state space where the posterior is higher.

PARTICLE FILTERING

A family of algorithms called **particle filtering** is designed to do just this. Particle filtering works as follows. First, a population of N samples is created by sampling from the prior distribution at time 0, $\mathbf{P}(\mathbf{X}_0)$. Then the update cycle is repeated for each time step:

- Each sample is propagated forward by sampling the next state value $\mathbf{x}_{t+1}$ given the current value $\mathbf{x}_t$ for the sample, using the transition model $\mathbf{P}(\mathbf{X}_{t+1}|\mathbf{x}_t)$.
- Each sample is weighted by the likelihood it assigns to the new evidence, $P(\mathbf{e}_{t+1}|\mathbf{x}_{t+1})$.
- The population is *resampled* to generate a new population of N samples. Each new sample is selected from the current population; the probability that a particular sample is selected is proportional to its weight. The new samples are unweighted.

The algorithm is shown in detail in Figure 15.16, and its operation for the umbrella DBN is illustrated in Figure 15.17.

We can show that this algorithm is consistent—gives the correct probabilities as N tends to infinity—by considering what happens during one update cycle. We will assume the sample population starts with a correct representation of the forward message $\mathbf{f}_{1:t}$ at time t:

```
function PARTICLEFILTERING($\mathbf{e}$,$N$,$dbn$) returns a set of samples for the next time step
  inputs: $\mathbf{e}$, the new incoming evidence
          $N$, the number of samples to be maintained
          $dbn$, a DBN with slice 0 variables $\mathbf{X}_0$ and slice 1 variables $\mathbf{X}_1$ and $\mathbf{E}_1$
  static: $S$, a vector of samples of size $N$
  local variables: $W$, a vector of weights of size $N$

  if $\mathbf{e}$ is empty then   /* initialization phase */
     for $i$ = 1 to $N$ do
        $S[i] \leftarrow$ sample from $\mathbf{P}(\mathbf{X}_0)$
  else do                         /* update cycle */
     for $i$ = 1 to $N$ do
        $S[i] \leftarrow$ sample from $\mathbf{P}(\mathbf{X}_1|\mathbf{X}_0 = S[i])$
        $W[i] \leftarrow \mathbf{P}(\mathbf{e}|\mathbf{X}_1 = S[i])$
     $S \leftarrow$ WEIGHTEDSAMPLEWITHREPLACEMENT($N$,$S$,$W$)
  return $S$
```

Figure 15.16 The particle filtering algorithm implemented as a recursive update operation with state (the set of samples). Each of the sampling steps involves sampling the relevant slice variables in topological order, much as in PRIOR-SAMPLE. The WEIGHTED-SAMPLE-WITH-REPLACEMENT operation can be implemented to run in $O(N)$ expected time.

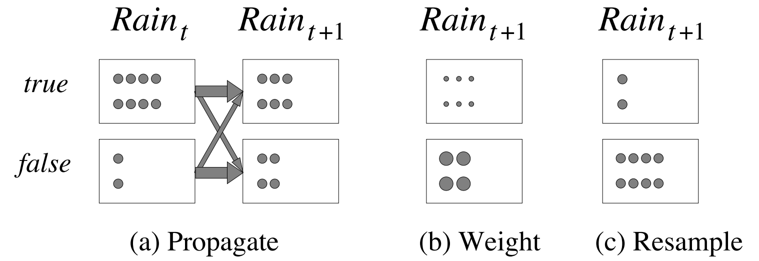


Figure 15.17 The particle filtering update cycle for the umbrella DBN with $N = 10$, showing the sample populations of each state. (a) At time t, 8 samples indicate $Rain$ and 2 indicate $\neg Rain$. Each is propagated forward by sampling the next state using the transition model. At time $t+1$, 7 samples indicate $Rain$ and 3 indicate $\neg Rain$. (b) $\neg Umbrella$ is observed at $t+1$. Each sample is weighted by its likelihood for the observation, as indicated by the size of the circles. (c) A new set of 10 samples is generated by weighted random selection from the current set, resulting in 4 samples that indicate $Rain$ and 6 that indicate $\neg Rain$.

writing $N(\mathbf{x}_t|\mathbf{e}_{1:t})$ for the number of samples occupying state $\mathbf{x}_t$ after observations $\mathbf{e}_{1:t}$ have been processed, we therefore have

$$N(\mathbf{x}_t|\mathbf{e}_{1:t})/N = P(\mathbf{x}_t|\mathbf{e}_{1:t}) \qquad (15.22)$$

for large N. Now we propagate each sample forward by sampling the state variables at $t+1$ given the values for the sample at t. The number of samples reaching state $\mathbf{x}_{t+1}$ from each

$\mathbf{x}_t$ is the transition probability times the population of $\mathbf{x}_t$; hence the total number of samples reaching $\mathbf{x}_{t+1}$ is

$$N(\mathbf{x}_{t+1}|\mathbf{e}_{1:t}) = \sum_{\mathbf{x}_t} P(\mathbf{x}_{t+1}|\mathbf{x}_t)N(\mathbf{x}_t|\mathbf{e}_{1:t})$$

Now we weight each sample by its likelihood for the evidence at $t+1$. A sample in state $\mathbf{x}_{t+1}$ receives weight $P(\mathbf{e}_{t+1}|\mathbf{x}_{t+1})$. The total weight of the samples in $\mathbf{x}_{t+1}$ after seeing $\mathbf{e}_{t+1}$ is therefore

$$W(\mathbf{x}_{t+1}|\mathbf{e}_{1:t+1}) = P(\mathbf{e}_{t+1}|\mathbf{x}_{t+1})N(\mathbf{x}_{t+1}|\mathbf{e}_{1:t})$$

Now for the resampling step. Since each sample is replicated with probability proportional to its weight, the number of samples in state $\mathbf{x}_{t+1}$ after resampling is proportional to the total weight in $\mathbf{x}_{t+1}$ before resampling:

$$\begin{aligned}
N(\mathbf{x}_{t+1}|\mathbf{e}_{1:t+1})/N &= \alpha W(\mathbf{x}_{t+1}|\mathbf{e}_{1:t+1}) \\
&= \alpha P(\mathbf{e}_{t+1}|\mathbf{x}_{t+1})N(\mathbf{x}_{t+1}|\mathbf{e}_{1:t}) \\
&= \alpha P(\mathbf{e}_{t+1}|\mathbf{x}_{t+1})\sum_{\mathbf{x}_t} P(\mathbf{x}_{t+1}|\mathbf{x}_t)N(\mathbf{x}_t|\mathbf{e}_{1:t}) \\
&= \alpha N P(\mathbf{e}_{t+1}|\mathbf{x}_{t+1})\sum_{\mathbf{x}_t} P(\mathbf{x}_{t+1}|\mathbf{x}_t)P(\mathbf{x}_t|\mathbf{e}_{1:t}) \quad \text{by Equation (15.22)} \\
&= \alpha' P(\mathbf{e}_{t+1}|\mathbf{x}_{t+1})\sum_{\mathbf{x}_t} P(\mathbf{x}_{t+1}|\mathbf{x}_t)P(\mathbf{x}_t|\mathbf{e}_{1:t}) \\
&= P(\mathbf{x}_{t+1}|\mathbf{e}_{1:t+1}) \quad \text{by Equation (15.4)}
\end{aligned}$$

Therefore the sample population after one update cycle correctly represents the forward message at time $t+1$.

Particle filtering is *consistent*, therefore, but is it *efficient*? In practice, it seems the answer is yes—particle filtering seems to maintain a good approximation to the true posterior using a constant number of samples. There are, as yet, no theoretical guarantees; particle filtering is currently an area of intensive study. Many variants and improvements have been proposed and the et of applications is growing rapidly. Because it is a sampling algorithm, particle filtering can be used easily with hybrid and continuous DBNs, allowing it to be applied to areas such as tracking complex motion patterns in video (Isard and Blake, 1996) and predicting the stock market (de Freitas *et al.*, 1999).

15.6 SPEECH RECOGNITION

SPEECH RECOGNITION

In this section, we look at one of the most important applications of temporal probability models—**speech recognition**. The task is to identify the sequence of words uttered by a speaker, given the acoustic signal. Speech is the dominant modality for communication between humans, and reliable speech recognition by machines would be immensely useful. Still more useful would be **speech understanding**—the identification of the *meaning* of the utterance. For this, we must wait until Chapter 22.

Speech provides our first contact with the raw, unwashed world of real sensor data. These data are *noisy*, quite literally; there can be background noise as well as artifacts introduced by the digitization process; there is variation in the way that words are pronounced, even by the same speaker; different words can sound the same; and so on. For these reasons, speech recognition has come to be viewed as a problem of probabilistic inference.

At the most general level, we can define the probabilistic inference problem as follows. Let $Words$ be a random variable ranging over all possible sequences of words that might be uttered, and let $signal$ be the observed acoustic signal sequence. Then the most likely interpretation of the utterance is the value of $Words$ that maximizes $P(words|signal)$. As is often the case, applying Bayes' rule is helpful:

$$P(words|signal) = \alpha P(signal|words)P(words)$$

ACOUSTIC MODEL

HOMOPHONES

LANGUAGE MODEL

$P(signal|words)$ is the **acoustic model**. It describes the sounds of words—for example, that "ceiling" begins with a soft "c" and sounds very similar to "sealing". (Words that sound the same are called **homophones**.) $P(words)$ is known as the **language model**. It specifies the prior probability of each utterance—for example, that "high ceiling" is a much more likely word sequence than "high sealing."

BIGRAM MODEL

PHONOLOGY

PHONES

The language models used in speech recognition systems are usually very simple. The **bigram model** that we describe later in this section gives the probability of each word following each other word. The acoustic model is much more complex. At its heart is an important discovery made in the field of **phonology** (the study of how language sounds), namely, that all human languages use a limited repertoire of about 40 or 50 sounds, called **phones**. Roughly speaking, a phone is the sound that corresponds to a single vowel or consonant, but there are some complications: combinations of letters such as "th" and "ng" produce single phones, and some letters produce different phones in different contexts (for example, the "a" in *rat* and *rate*. Figure 15.18 lists all the phones in English with an example of each.

PRONUNCIATION

The existence of phones makes it possible to divide the acoustic model into two parts. The first part deals with **pronunciation** and specifies, for each word, a probability distribution over possible phone sequences. For example, "ceiling" is pronounced [s iy l ih ng], or sometimes [s iy l ix ng], or sometimes even [s iy l en]. The phones are not directly observable, so, roughly speaking, speech is represented as a hidden Markov model whose state variable X_t specifies which phone is being uttered at time t.

SIGNAL PROCESSING

The second part of the acoustic model deals with the way that phones are realized as acoustic signals—that is, the evidence variable E_t for the hidden Markov model gives the observed features of the acoustic signal at time t, and the acoustic model specifies $P(E_t|X_t)$, where X_t is the current phone. This model must allow for variations in pitch, speed, and volume, and relies on techniques from **signal processing** to provide signal descriptions that are reasonably robust against these kinds of variations.

The remainder of the section describes the models and algorithms from the bottom up, beginning with acoustic signals and phones, then individual words, and finally entire sentences. We conclude with a description of how all these models are trained and how well the resulting systems work.

Vowels		Consonants B-N		Consonants P-Z	
Phone	Example	Phone	Example	Phone	Example
[iy]	b**ea**t	[b]	**b**et	[p]	**p**et
[ih]	b**i**t	[ch]	**Ch**et	[r]	**r**at
[eh]	b**e**t	[d]	**d**ebt	[s]	**s**et
[æ]	b**a**t	[f]	**f**at	[sh]	**sh**oe
[ah]	b**u**t	[g]	**g**et	[t]	**t**en
[ao]	b**ough**t	[hh]	**h**at	[th]	**th**ick
[ow]	b**oa**t	[hv]	**h**igh	[dh]	**th**at
[uh]	b**oo**k	[jh]	**j**et	[dx]	bu**tt**er
[ey]	b**ai**t	[k]	**k**ick	[v]	**v**et
[er]	B**er**t	[l]	**l**et	[w]	**w**et
[ay]	b**uy**	[el]	bott**le**	[wh]	**wh**ich
[oy]	b**oy**	[m]	**m**et	[y]	**y**et
[axr]	din**er**	[em]	bott**om**	[z]	**z**oo
[aw]	d**ow**n	[n]	**n**et	[zh]	mea**s**ure
[ax]	**a**bout	[en]	butt**on**		
[ix]	ros**e**s	[ng]	si**ng**		
[aa]	c**o**t	[eng]	Wash**ing**ton	[-]	*(silence)*

Figure 15.18 The DARPA phonetic alphabet, or **ARPAbet**, listing all the phones used in American English. There are several alternative notations, including an International Phonetic Alphabet (IPA), which contains the phones in all known languages.

Speech sounds

SAMPLING RATE

QUANTIZATION FACTOR

Sound waves are periodic changes in pressure that propagate through the air. Sound can be measured by a microphone whose diaphragm is displaced by the pressure changes and generates a continuously varying current. An analog-to-digital converter measures the size of the current—which corresponds to the amplitude of the sound wave—at discrete intervals determined by the **sampling rate**. For speech, a sampling rate between 8 and 16 kHz (i.e., 8 to 16,000 times per second) is typical. The precision of each measurement is determined by the **quantization factor**; speech recognizers typically keep 8 to 12 bits. That means that a low-end system, sampling at 8 kHz with 8-bit quantization, would require nearly half a megabyte per minute of speech. It would be impractical to construct and manipulate $P(signal|phone)$ distributions with so much signal information; therefore, we need to develop more concise descriptions of the signal.

FRAMES

FEATURES

First, we observe that although the sound frequencies in speech may be several kHz, the *changes* in the content of the signal occur much less often, perhaps at no more than 100 Hz. Therefore, speech systems summarize the properties of the signal over extended intervals called **frames**. A frame length of about 10 msecs (i.e., 80 samples at 8 kHz) is short enough to ensure that few short-duration phenomena will be smudged out by the summarization process. Within each frame, we represent what is happening with a vector of **features**. For example, we might want to characterize the amount of energy at each of several frequency ranges. Other important features include overall energy in a frame, and the difference from the previous frame. Picking out features from a speech signal is like listening to an orchestra and saying "here the French horns are playing loudly and the violins are playing softly." Figure 15.19 shows the sequence of transformations from the raw sound to a sequence of frames. Note that the frames overlap; this prevents us from losing information if an important acoustic event just happens to fall on a frame boundary.

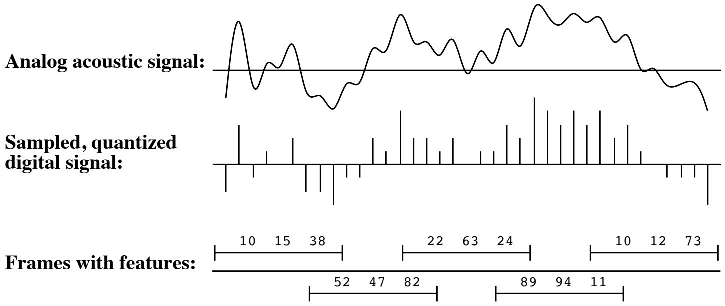


Figure 15.19 Translating the acoustic signal into a sequence of frames; each frame is described by the values of three acoustic features.

In our example, we have shown frames with just three features. Real systems may have tens or even hundreds of features. If there are n features and each has, say, 256 possible values, then a frame is described by a point in n-dimensional space and there are 256^n possible frames. For $n > 2$ it would be impractical to represent the distribution $P(features|phone)$ as an explicit table, so we need further compression. There are two possible approaches:

VECTOR QUANTIZATION

- The method of **vector quantization** or VQ divides the n-dimensional space into, say, 256 regions labelled C1 through C256. Each frame can then be represented with a single label rather than a vector of n numbers. Thus, the tabulated distribution $P(VQ\ label|phone)$ has 256 probabilities specified for each phone. Vector quantization is no longer popular in large-scale systems.
- Instead of discretizing the feature space, we can use a parameterized continuous distribution to describe $P(features|phone)$. For example, we could use a Gaussian distribution with a different mean and covariance matrix for each phone. This works well if the acoustic realizations of each phone are clustered in a single region of feature space. In practice, the sounds can be spread over several regions, and a **mixture of Gaussians** must be used. A mixture is a weighted sum of k individual distributions, so $P(features|phone)$ has k weights, k mean vectors of size n, and k covariance matrices of size n^2—that is, $O(kn^2)$ parameters for each phone.

MIXTURE OF GAUSSIANS

Of course, some information is lost in going from the full speech signal to a VQ label or a set of mixture parameters. The art of signal processing lies in choosing features and regions (or Gaussians) so that the loss of *useful* information is minimized. A given speech sound can be pronounced so many ways: loud or soft, fast or slow, high-pitched or low, against a background of silence or noise, and by any of millions of different speakers each with different accents and vocal tracts. Signal processing hopes to eliminate the variations while keeping the commonalities that define the sound.

There are two more refinements we need to make to the simple model we have described so far. The first deals with the temporal structure of phones. In normal speech, most phones have a duration of 50-100 milliseconds, or 5-10 frames. The probability model $P(features|phone)$ is the same for all these frames, whereas most phones have a good deal of internal structure. For example, [t] is one of several **stop consonants** in which the flow of air is cut off for a short period before a sharp release. Examining the acoustic signal, we find that [t] has a silent beginning, a small explosion in the middle, and (usually) a hissing at the end. This internal structure of phones can be captured by the **three-state phone** model; each phone has Onset, Mid, and End states, and each state has its own distribution over features.

STOP CONSONANTS

THREE-STATE PHONE

The second refinement deals with the context in which the phone is uttered. The sound of a given phone can change depending on the surrounding phones.[5] For example, the [t] in “tar” has a short hiss at the end, prior to the voiced [aa r], whereas the [t] in “star” does not. Both of these [t] sounds are produced by closing the tongue against the roof of the mouth just behind the teeth, whereas the [t] in “eighth” is often produced with the tongue pressed against the front teeth because it is followed immediately by a [th] sound. These contextual effects are partially captured by the **triphone** model, in which the acoustic model for each phone is allowed to depend on the preceding and succeeding phones. Thus, the [t] in “star” is written [t(s,aa)], i.e., [t] with left-context [s] and right-context [aa].

TRIPHONE

The combined effect of the three-state and triphone models is to increase the number of possible states of the temporal process from n phones in the original phone alphabet ($n \approx 50$ for the ARPAbet) to $3n^3$. Experience shows that the improved accuracy more than offsets the extra expense in terms of inference and learning.

Words

We can think of each word as specifying a distinct probability distribution $\mathbf{P}(X_{1:t}|word)$, where X_i specifies the phone state in the ith frame. Typically, we separate this distribution into two parts. The **pronunciation model** gives a distribution over phone sequences (ignoring metric time and frames), while the **phone model** describes how a phone maps into a sequence of frames.

Consider the word “tomato.” It is well-known that you say [t ow m ey t ow] and I say [t ow m aa t ow], so the pronunciation model has to account for dialects. The top of Figure 15.20 shows a transition model that provides for this variation. There are only two possible paths through the model, one corresponding to the phone sequence [t ow m ey t ow] and the other to [t ow m aa t ow]. The probability of a path is the product of the probabilities on the arcs that make up the path:

$$P([towmeytow]|\text{“tomato”}) = P([towmaatow]|\text{“tomato”}) = 0.5$$

[5] In this sense, the “phone model” of speech should be thought of as a useful approximation rather than an immutable law.

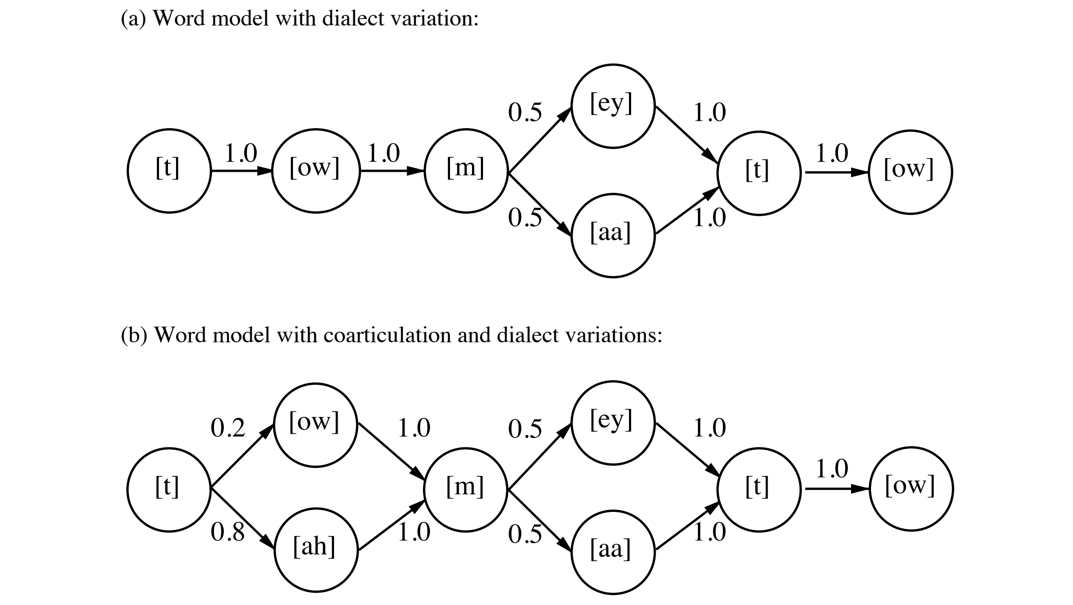


Figure 15.20 Two pronunciation models of the word "tomato." Each model is shown as a transition diagram with states as circles and arrows showing allowed transitions with their associated probabilities. (a) A model allowing for dialect differences. The 0.5 numbers are estimates based on the two authors' preferred pronunciations. (b) A model with a coarticulation effect on the first vowel, allowing either the [ow] or the [ah] phone.

COARTICULATION

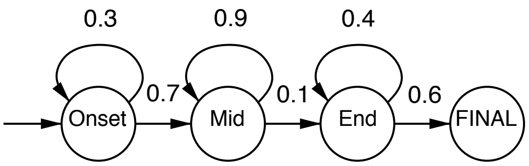
The second source of phonetic variation is **coarticulation**. Remember that speech sounds are produced by moving the tongue and jaw and forcing air through the vocal tract. When the speaker is talking slowly and deliberately, there is time to place the tongue in just the right spot before producing a phone. But when the speaker is talking quickly (or sometimes even at a normal pace), the movements slur together. For example, the [t] phone is produced with the tongue at the top of the mouth, whereas the [ow] has the tongue near the bottom. When spoken quickly, the tongue often goes to an intermediate position, and we get [t ah] rather than [t ow]. The bottom half of Figure 15.20 gives a more complicated pronunciation model for "tomato" that takes this coarticulation effect into account. In this model there are four distinct paths and we have

$$P([towmeytow]|\text{“tomato”}) = P([towmaatow]|\text{“tomato”}) = 0.1$$
$$P([tahmeytow]|\text{“tomato”}) = P([tahmaatow]|\text{“tomato”}) = 0.4$$

Similar models can be constructed for every word we want to be able to recognize.

The model for a three-state phone is shown as a state transition diagram in Figure 15.21. The model is for a particular phone, [m], but all phones will have models with similar topology. For each phone state, we show the associated acoustic model assuming that the signal is represented by a VQ label. For example, the model asserts that $P(E_t = C_1|X_t = [\text{m}]_{\text{Onset}}) = 0.5$. Notice the self-loops in the figure; for example, the $[\text{m}]_{\text{Mid}}$ state persists with probability 0.9. This means that the $[\text{m}]_{\text{Mid}}$ state has an expected duration of 10 frames. In this way,

Phone HMM for [m]:



Output probabilities for the phone HMM:

Onset:	Mid:	End:
C1: 0.5	C3: 0.2	C4: 0.1
C2: 0.2	C4: 0.7	C6: 0.5
C3: 0.3	C5: 0.1	C7: 0.4

Figure 15.21 An HMM for the three-state phone [m]. Each state has several possible outputs, each with its own probability. The VQ labels C_1 through C_7 are arbitrary.

we can specify the relative durations of phones; of course, the probabilistic model allows for variations, such as arise with fast and slow speech.

We can construct similar models for each phone, possibly depending on the triphone context. Each word model, when combined with the phone models, gives a complete specification of an HMM. The model specifies the transition probabilities between phone states from frame to frame, as well as the acoustic feature probabilities for each phone state.

ISOLATED WORDS

If we want to recognized **isolated words**—that is, words spoken without any surrounding context and with clear boundaries—then we need to find the word that maximizes

$$P(word|e_{1:t}) = \alpha P(e_{1:t}|word)P(word)$$

The prior probability $P(word)$ can be obtained from actual text data, as described later. The quantity $P(e_{1:t}|word)$ is the likelihood of the sequence of acoustic features according to the word model. Section 15.2 covered the computation of such likelihoods; in particular, Equation (15.6) gives a simple recursive computation whose cost is linear in t and in the number of states of the Markov chain. To find the most likely word, we can perform this calculation for each possible word model, multiply by the prior, and select the best word accordingly.

Sentences

CONTINUOUS SPEECH

To have a conversation with a human, a machine needs to be able to recognize **continuous speech** rather than just isolated words. One might think that continuous speech is nothing more than a sequence of words, to each of which we can apply the algorithm from the previous section. This approach fails for two reasons. First, we have already seen (page 552) that the sequence of most likely words is not the most likely sequence of words. For example, in the movie *Take the Money and Run*, a bank teller interprets Woody Allen's sloppily written hold-up note as saying "I have a gub." A good language model would suggest "I have a gun" as a much more likely sequence, even though the last word looks more like "gub" than "gun". The

Word	Unigram count	Previous words							
		of	in	is	on	to	from	model	agent
the	33508	3833	2479	832	944	1365	597	28	24
on	2573	1	0	33	2	1	0	0	6
of	15474	0	0	29	1	0	0	88	7
to	11527	0	4	450	21	4	16	9	82
is	10566	3	6	1	4	2	1	47	127
model	752	8	1	0	1	14	0	6	4
agent	2100	10	3	3	2	3	0	0	36
idea	241	0	0	0	0	0	0	0	0

Figure 15.22 A partial table of unigram and bigram counts for the words in this book. There are 513,893 total words; "the" is the most common at 33,508. The bigram "of the" is the most common at 15,474. That is, one out of every 15 words is "the" and one out of every 33 word pairs is "of the." Some counts are higher than expected (e.g. 4 for "on is") because the bigram counts ignore punctuation—one sentence might end with "on" and the next begin with "is."

SEGMENTATION

second issue we must face with continuous speech is **segmentation**, the problem of deciding where one word ends and the next begins. Anyone who has tried to learn a foreign language will appreciate this problem: at first all the words seem to run together. Gradually, one learns to pick out words from the jumble of sounds. In this case, first impressions are correct; a spectrographic analysis shows that in fluent speech, the words really *do* run together with no silence between them. We learn to identify word boundaries despite the lack of silence.

Let us begin with the language model, whose job in speech recognition is to specify the probability of each possible sequence of words. Using the notation $w_1 \cdots w_n$ to denote a string of n words and w_i to denote the ith word of the string, we can write an expression for the probability of a string using the chain rule as follows:[6]

$$\begin{aligned} P(w_1 \cdots w_n) &= P(w_1)\,P(w_2|w_1)\,P(w_3|w_1 w_2)\,\cdots\,P(w_n|w_1 \cdots w_{n-1}) \\ &= \textstyle\prod_{i=1}^{n} P(w_i|w_1 \cdots w_{i-1}) \end{aligned}$$

BIGRAM

Most of these terms are quite complex and difficult to estimate or compute. Fortunately, we can approximate this formula with something simpler and still capture a large part of the language model. One simple, popular, and effective approach is the **bigram** model. This model approximates $P(w_i|w_1 \cdots w_{i-1})$ with $P(w_i|w_{i-1})$. In other words, it makes a first-order Markov assumption for word sequences.

A big advantage of the bigram model is that it is easy to train the model by counting the number of times each word pair occurs in a representative corpus of strings and using the counts to estimate the probabilities. For example, if "a" appears 10,000 times in the training

[6] Strictly speaking, the probability of a word sequence depends strongly on the *context* of the utterance; for example, "I have a gun" is much more common on notes passed to a bank teller than it is in, say, the Wall Street Journal. Few speech recognizers handle context, other than by training a special-purpose language model for a particular task.

corpus and it is followed by "gun" 37 times, then $\hat{P}(gun_i|a_{i-1}) = 37/10,000$, where by $\hat{P}$ we mean the estimated probability. After such training one would expect "I have" and "a gun" to have relatively high estimated probabilities, while "I has" and "an gun" would have low probabilities. Figure 15.22 shows some bigram counts derived from the words in this chapter.

TRIGRAM

It is possible to go to a **trigram** model that provides values for $P(w_i|w_{i-1}w_{i-2})$. This is a more powerful language model, capable of determining that "ate a banana" is more likely than "ate a bandana." For trigram models, and to a lesser extent for bigram and unigram models, thre is a problem with counts of zero. We wouldn't want to say that a combination of words that didn't happen to appear in the training corpus is improbable. The process of **smoothing** gives a small non-zero probability to such combinations. It is discussed on page 817.

Bigram or trigram models are not as sophisticated as some of the grammar models we will see in Chapters 22 and 23, but they account for local context-sensitive effects better, and manage to capture some local syntax. For example, the fact that the word pairs "I has" and "man have" get low scores is reflective of subject-verb agreement. The problem is that these relationships can only be detected locally: "the man have" gets a low score, but "the man over there have" is not penalized.

Now we consider how to combine the language model with the word models, so that we can handle word sequences properly. We'll assume a bigram language model for simplicity. With such a model, we can combine all the word models (which are comprised in turn of pronunciation models and phone models) into one large HMM model. A state in a single-word HMM is a frame labelled by the current phone and phone state (for example, $[\mathrm{m}]_{\mathrm{onset}}$); a state in a continuous-speech HMM is also labelled with a word, as in $[\mathrm{m}]^{\mathrm{tomato}}_{\mathrm{onset}}$. If each word has an average of p three-state phones in its pronunciation model, and there are W words, then the continuous-speech HMM has $3Wp$ states. Transitions can occur between phone states within a given phone; between phones in a given word, and between the final state of one word and the initial state of another. The transitions between words occur with probabilities specified by the bigram model.

Once we have constructed the combined HMM, we can use it to analyze the continuous speech signal. In particular, the Viterbi algorithm embodied in Equation (15.10) can be used to find the most likely state sequence. From this state sequence, we can extract a word sequence simply by reading the word labels from the states. Thus, the Viterbi algorithm solves the word segmentation problem by using dynamic programming to consider (in effect) all possible word sequences and word boundaries simultaneously.

Notice that we didn't say "we can extract *the most likely* word sequence." The most likely word sequence is not necessarily the one that contains the most likely state sequence. This is because the probability of a word sequence is the sum of probabilities over all possible state sequences consistent with that word sequence. Comparing two word sequences, say "a back" and "aback," it might be that case that there are ten alternative state sequences for "a back," each with probability 0.03, but just one state sequence for "aback," with probability 0.20. Viterbi chooses "aback," but "a back" is actually more likely.

In practice, this difficulty is not life-threatening, but it is serious enough that other

A* DECODER

approaches have been tried. The most common is the **A* decoder**, which makes ingenious use of A* search (see Chapter 4) to find the most likely word sequence. The idea is to view each word sequence as a path through a graph whose nodes are labelled with words. The successors of a node are all the words that can come next; thus, the graph for all sentences of length n or less has n layers, each of width at most W, where W is the number of possible words. With a bigram model, the cost $g(w_1, w_2)$ of an arc between nodes label w_1 to w_2 is given by $-\log P(w_2|w_1)$; in this way, the total path cost of a sequence $w_1 \cdots w_n$ is

$$\sum_{i=1}^{n} -\log P(w_i|w_{i-1}) = -\log \prod_{i=1}^{n} P(w_i|w_{i-1}) .$$

With this definition of path cost, finding the shortest path is exactly equivalent to finding the most likely word sequence. For the process to be efficient, we also need a good heuristic $h(w_i)$ to estimate the cost of completing the word sequence. Obviously, this has something to do with how much of the speech signal is not yet covered by the words on the current path. As yet, no especially interesting heuristics have been devised for this problem.

Building a speech recognizer

The quality of a speech recognition system depends on the quality of all its components—the language model, the word pronunciation models, the phone models, and the signal processing algorithms used to extract spectral features from the acoustic signal. We have discussed how the language model may be constructed, and we leave the details of signal processing to other textbooks. That leaves the pronunciation and phone models. The *structure* of the pronunciation models—such as the tomato models in Figure 15.20—is usually developed by hand. Large pronunciation dictionaries are now available for English and other languages, although their accuracy varies greatly. The structure of the three-state phone models is the same for all phones, as shown in Figure 15.21. That leaves the probabilities themselves. How are these to be obtained, given that the models may require hundreds of thousands or millions of parameters?

The only plausible method is to learn the models from actual speech data, of which there is certainly no shortage. The next question is how to do the learning. We give the answer in full in Chapter 19, but we can give the main ideas here. Consider the bigram language model; we explained how to learn it by looking at frequencies of word pairs in actual text. Can we do the same for, say, phone transition probabilities in the pronunciation model? The answer is yes, but only if someone goes to the trouble of annotating every occurrence of each word with the right phone sequence. This is a difficult and error-prone task, but has been carried out for some standard data sets containing several hours of speech. If we know the phone sequences, we can estimate transition probabilities for the pronunciation models from frequencies of phone pairs. Similarly, if we are given the phone state for each frame—an even more excruciating manual labelling task—then we can estimate transition probabilities for the phone models. Given the phone state and the acoustic features in each frame, we can also estimate the acoustic model, either directly from frequencies (for VQ models) or using statistical fitting methods (for mixture-of-Gaussian models; see Chapter 19).

The cost and rarity of hand-labelled data, and the fact that the available hand-labelled data sets may not represent the kinds of speakers and acoustic conditions found in a new recognition context, could doom this approach to failure. *Fortunately, the* ***expectation–maximization*** *or EM algorithm learns HMM transition and sensor models without the need for labelled data.* Estimates derived from hand-labelled data can be used to initialize the models; after that, EM takes over and trains the models for the task at hand. The idea is simple: given an HMM and an observation sequence, we can use the smoothing algorithms from Sections 15.2 and 15.3 to compute the probability of each state at each time step, and, by a simple extension, the probability of each state–state pair at consecutive time steps. These probabilities can be viewed as *uncertain labels* in place of the definite labels provided by hand. From the uncertain labels, we can estimate new transition and sensor probabilities, and the EM procedure repeats. The method is guaranteed to increase the fit between model and data on each iteration, and generally converges to a much better set of parameter values than those provided by the initial, hand-labelled estimates.

State-of-the-art speech systems use enormous data sets and massive computational resources to train their models. For isolated word recognition under good acoustic conditions (no background noise or reverberation) with a vocabulary of a few thousand words and a single speaker, accuracy can be over 99%. For unrestricted continuous speech with a variety of speakers, 60–80% accuracy is common, even with good acoustic conditions. With background noise and telephone transmission, accuracy degrades further. Although fielded systems have improved continuously for decades, there is still room for many new ideas.

15.7 Summary

This chapter has addressed the general problem of representing and reasoning about probabilistic temporal processes. The main points are as follows:

- The changing state of the world is handled using a set of random variables to represent the state at each point in time.
- Representations can be designed to satisfy the **Markov property**, so that the future is independent of the past given the present. Combined with the assumption that the process is **stationary**—i..e, the dynamics do not change over time—this greatly simplifies the representation.
- A temporal probability model can be thought of as containing a **transition model** describing the evolution and a **sensor model** describing the observation process.
- The principal inference tasks in temporal models are **filtering**, **prediction**, **smoothing**, and computing the **most likely explanation**. Each of these can be achieved using simple, recursive algorithms whose runtime is linear in the length of the sequence.
- Three families of temporal models were studied in more depth: **hidden Markov models**, **Kalman filters**, and **dynamic Bayesian networks** (which include the other two as special cases).
- **Speech recognition** and **tracking** are two important applications for temporal probability models.

- Unless special assumptions are made, as in Kalman filters, exact inference with many state variables appears to be intractable. In practice, the **particle filtering** algorithm seems to be an effective approximation algorithm.

Bibliographical and Historical Notes

Many of the basic ideas for estimating the state of dynamical systems came from the mathematician C. F. Gauss (1809). Gauss developed a deterministic least-squares algorithm for the problem of estimating orbits from astronomical observations. The Russian mathematician A. A. Markov (1913) developed what was later called the **Markov assumption** in his analysis of stochastic processes; he estimated a first-order Markov chain on letters from the text of *Eugene Onegin*. Significant classified work on filtering was done during World War II by Wiener (1942) for continuous-time processes and by Kolmogorov (1941) for discrete-time processes. Although this work led to important technological developments over the next twenty years, its use of a frequency-domain representation made many calculations quite cumbersome. Direct state-space modelling of the stochastic process turned out to be simpler, as shown by Swerling (1959) and Kalman (Kalman, 1960). The latter paper introduced what is now known as the Kalman filter for forward inference in linear systems with Gaussian noise. Important results on smoothing were derived by Rauch *et al.* (1965), and the impressively named **Rauch-Tung-Striebel smoother** is still a standard technique today. Many early results are gathered in Gelb (1974). Bar-Shalom and Fortmann (1988) give a more modern treatment with a Bayesian flavor, as well as many references to the vast literature on the subject.

In many applications of Kalman filtering, one must deal not only with uncertain sensing and dynamics but also with uncertain *identity*—that is, if there are multiple objects being monitored, the system must determine which observations were generated by which objects before it can update each of the state estimates. This is the problem of **data association** (Bar-Shalom and Fortmann, 1988; Bar-Shalom, 1992). With n observations and n tracks (a fairly benign case), there are $n!$ possible assignments of observations to tracks; a proper probabilistic treatment must take all of them into account, and this can be shown to be NP-hard (Cox, 1993; Cox and Hingorani, 1994). Polynomial-time approximation methods based on MCMC appear to work well in practice (Pasula *et al.*, 1999). It is interesting to note that the data association problem is an instance of probabilistic inference in a *first-order* language—unlike most probabilistic inference problems, which are purely propositional, data association involves *objects* as well as the *identity relation*. It is therefore intimately connected to the first-order probabilistic languages that were mentioned in Chapter 14. Recent work has shown that reasoning about identity in general, and data association in particular, can be carried out within the first-order probabilistic framework (Pasula and Russell, 2001).

DATA ASSOCIATION

The hidden Markov model and associated algorithms for inference and learning, including the forward–backward algorithm, were developed by Baum and Petrie (1966). Similar ideas also appeared independently in the Kalman filtering community (Rauch *et al.*, 1965). The forward–backward algorithm was one of the main precursors of the general formulation of the EM algorithm (; see also Chapter 19 Dempster *et al.*, 1977). Constant-space smoothing appears in Binder *et al.* (1997), as does the divide-and-conquer algorithm developed in Exercise 15.3.

Dynamic belief networks (DBNs) can be viewed as a sparse encoding of a Markov process, and were first used in AI by Dean and Kanazawa (1989b), Nicholson (1992), and Kjaerulff (1992). The last work includes a generic extension to the HUGIN belief net system to provide the necessary facilities for dynamic belief network generation and compilation. Dynamic Bayesian networks have become popular for modelling a variety of complex motion processes in computer vision (Huang *et al.*, 1994; Intille and Bobick, 1999). The link between HMMs and DBNs, and between the forward–backward algorithm and Bayesian network propagation, was made explicitly by Smyth *et al.* (1997). A further unification with Kalman filters (as well as several other statistical models) appears in Roweis and Ghahramani (1999).

The particle filtering algorithm described in Section 15.5 has a particularly interesting history. The first sampling algorithms for filtering were developed in the control theory community by Handschin and Mayne (1969), and the resampling idea that is the core of particle filtering appeared in a Russian control journal (Zaritskii *et al.*, 1975). It was later reinvented in statistics as **sequential importance-sampling resampling** or **SIR** (Rubin, 1988; Liu and Chen, 1998), in control theory as particle filtering (Gordon *et al.*, 1993; Gordon, 1994), in AI as **survival of the fittest** (Kanazawa *et al.*, 1995), and in computer vision as **condensation** (Isard and Blake, 1996). The paper by Kanazawa *et al.* (1995) includes an improvement called **evidence reversal** whereby the state at time $t+1$ is sampled conditional on both the state at time *t and the evidence at time* $t+1$. This allows the evidence to influence sample generation directly, and was proved (independently) by Doucet (1997) to reduce the approximation error.

Alternative methods for approximate filtering include the **decayed MCMC** algorithm (Marthi *et al.*, 2002) and the factored approximation method of Boyen *et al.* (1999). Both of these methods have the important property that the approximation error does not diverge over time. Variational techniques (see Chapter 14) have also been developed for temporal models. Ghahramani and Jordan (1997) discuss an approximation algorithm for the **factorial HMM**, a DBN in which two or more independently evolving Markov chains are linked by a shared observation stream. Jordan *et al.* (1998) cover a number of other applications.

The prehistory of speech recognition began in the 1920s with Radio Rex, a voice-activated toy dog. Rex jumped in response to sound frequencies near 500 Hz, which corresponds to the [eh] vowel in “Rex!” Somewhat more serious work began after World War II. At AT&T Bell Labs, a system was build for recognizing isolated digits (Davis *et al.*, 1952) using simple pattern matching of acoustic features. Phone transition probabilities were first used in a system built at University College, London by Fry (1959) and Denes (1959). Starting in 1971, the Defense Advanced Research Projects Agency (DARPA) of the United States Department of Defense funded four competing five-year projects to develop high-performance speech recognition systems. The winner, and the only system to meet the goal of 90% accuracy with a 1000-word vocabulary, was the HARPY system at CMU (Lowerre, 1976; Lowerre and Reddy, 1980).[7] The final version of HARPY was derived from a system called DRAGON built by CMU graduate student James Baker (1975), which was the first to use HMMs for speech. Almost simultaneously, Jelinek (1976) at IBM had developed another HMM-based system. From that point onwards, probabilistic methods in general, and HMMs in particular, came to dominate speech recognition research and development. Recent years have been characterized by incremental progress, larger data sets and models, and more rigorous competitions on more realistic speech tasks. Some researchers have explored the possibility of using DBNs instead of HMMs for speech, with the aim of using the greater expressive power of DBNs to capture more of the complex hidden state of the speech apparatus (Zweig and Russell, 1998; Richardson *et al.*, 2000).

Several good textbooks on speech recognition are available (Rabiner and Juang, 1993; Jelinek, 1997; Gold and Morgan, 2000; Huang *et al.*, 2001). Waibel and Lee (1990) collect important papers in the area, including some tutorial ones. The presentation in this chapter drew on the survey by Kay, Gawron, and Norvig (1994), and on the textbook by Jurafsky and Martin (2000). Speech recognition research is published in *Computer Speech and Language*, *Speech Communications*, and the IEEE *Transactions on Acoustics, Speech, and Signal Processing*, and at the DARPA Workshops on Speech and Natural Language Processing and the Eurospeech, ICSLP, and ASRU conferences.

EXERCISES

15.1 Show that any second-order Markov process can be rewritten as a first-order Markov process with an augmented set of state variables. Can this always be done *parsimoniously*—that is, without increasing the number of parameters needed to specify the transition model?

15.2 In this exercise we examine what happens to the probabilities in the umbrella world in the limit of long time sequences.

a. Suppose we observe an unending sequence of days on which the umbrella appears. Show that, as the days go by, the probability of rain on the current day increases monotonically towards a fixed point. Calculate this fixed point.

b. Now consider *forecasting* further and further into the future, given just the first two umbrella observations. First, compute the probability $P(R_{2+k}|U_1,U_2)$ for $k=1\ldots20$ and plot the results. You should see that the probability converges towards a fixed point. Calculate the exact value of this fixed point.

15.3 This exercise develops a space-efficient variant of the forward–backward algorithm

[7] The second-ranked system in the competition, HEARSAY-II (Erman *et al.*, 1980), had a great deal of influence on other branches of AI research because of its use of the **blackboard architecture**. It was a rule-based expert system with a number of more or less independent, modular **knowledge sources** which communicated via a common **blackboard** from which they could write and read. Blackboard systems are the foundation of modern user interface architectures.

described in Figure 15.4. We wish to compute $\mathbf{P}(\mathbf{X}_k|\mathbf{e}_{1:t})$ for $k=1,\ldots,t$. This will be done with a divide-and-conquer approach.

a. Suppose, for simplicity, that t is odd, and let the halfway point be $h=(t+1)/2$. Show that $\mathbf{P}(\mathbf{X}_k|\mathbf{e}_{1:t})$ can be computed for $k=1,\ldots,h$ given just the initial forward message $\mathbf{f}_{1:0}$, the backward message $\mathbf{b}_{h+1:t}$, and the evidence $\mathbf{e}_{1:h}$.

b. Show a similar result for the second half of the sequence.

c. Given the results of (a) and (b), a recursive, divide-and-conquer algorithm can be constructed by first running forward along the sequence and then backwards from the end, storing just the required messages at the middle and the ends. Then the algorithm is called on each half. Write out the algorithm in detail.

d. Compute the time and space complexity of the algorithm as a function of t, the length of the sequence. How does this change if we divide into more than two pieces?

15.4 On page 552, we outlined a flawed procedure for finding the most likely state sequence, given an observation sequence. The procedure involves finding the most likely state at each time step, using smoothing, and returning the sequence composed of these states. Show that, for some temporal probability models and observation sequences, this procedure returns an impossible state sequence (i.e., the posterior probability of the sequence is zero).

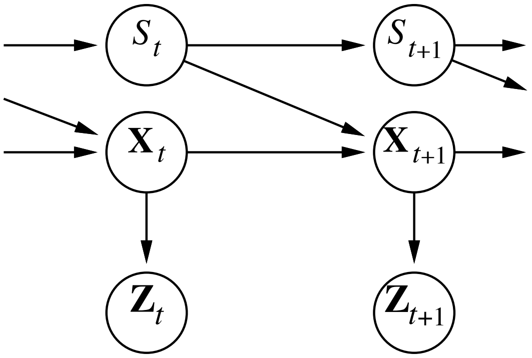


Figure 15.23 A Bayesian network representation of a switching Kalman filter. The switching variable S_t is a discrete state variable whose value determines the transition model for the continuous state variables $\mathbf{X}_t$. For any discrete state i, the transition model $\mathbf{P}(\mathbf{X}_{t+1}|\mathbf{X}_t, S_t=i)$ is a linear Gaussian model, just as in a regular Kalman filter. The transition model for the discrete state, $\mathbf{P}(S_{t+1}|S_t)$, can be thought of as a matrix just as in a hidden Markov model.

15.5 Often we wish to monitor a continuous-state system whose behavior switches unpredictably among a set of k distinct "modes." For example, an aircraft trying to evade a missile may execute a series of distinct maneuvers that the missile may attempt to track. A Bayesian network representation of such a **switching Kalman filter** model is shown in Figure 15.23.

a. Suppose that the discrete state S_t has k possible values and that the prior continuous state estimate $\mathbf{P}(\mathbf{X}_0)$ is a multivariate Gaussian distribution. Show that the prediction

$\mathbf{P}(\mathbf{X}_1)$ is a **mixture of Gaussians**—that is, a weighted sum of Gaussians such that the weights sum to 1.

b. Show that if the current continuous state estimate $\mathbf{P}(\mathbf{X}_t|\mathbf{e}_{1:t})$ is a mixture of m Gaussians, then the updated state estimate $\mathbf{P}(\mathbf{X}_{t+1}|\mathbf{e}_{1:t+1})$ will be a mixture of km Gaussians in the general case.

c. What aspect of the temporal process do the weights in the Gaussian mixture represent?

Together, the results in (a) and (b) show that the representation of the posterior grows without limit even for switching Kalman filters, which are the simplest hybrid dynamic models.

15.6 Complete the missing step in the derivation of Equation (15.18), the first update step for the one-dimensional Kalman filter.

15.7 Let us examine the behavior of the variance update in Equation (15.19).

a. Plot the value of σ_t^2 as a function of t, given various values for σ_x^2 and σ_z^2.

b. Show that the update has a fixed point σ^2 such that $\sigma_t^2 \to \sigma^2$ as $t \to \infty$, and calculate it.

c. Give a qualitative explanation for what happens as $\sigma_x^2 \to 0$ and as $\sigma_z^2 \to 0$.

15.8 Show how to represent an HMM as a recursive relational probabilistic model, as suggested in Section 14.6.

15.9 In this exercise, we analyze in more detail the persistent failure model for the battery sensor in Figure 15.13(a).

a. Figure 15.13(b) stops at $t = 32$. Describe qualitatively what should happen as $t \to \infty$ if the sensor continues to read 0.

b. Suppose that the external temperature affects the battery sensor, in such a way that transient failures become more likely as temperature increases. Show how to augment the DBN structure in Figure 15.13(a) and explain any required changes to the CPTs.

c. Given the new network structure, can battery readings be used by the robot to infer the current temperature?

15.10 Consider applying the variable elimination algorithm to the umbrella DBN unrolled for three slices, where the query is $\mathbf{P}(R_3|U_1, U_2, U_3)$. Show that the complexity of the algorithm—the size of the largest factor—is the same whether the rain variables are eliminated in forward or backward order.

15.11 The model of “tomato” in Figure 15.20 allows for a coarticulation on the first vowel by giving two possible phones. An alternative approach is to use a triphone model in which the [ow(t,m)] phone automatically includes the change in vowel sound. Draw a complete triphone model for “tomato,” including the dialect variation.