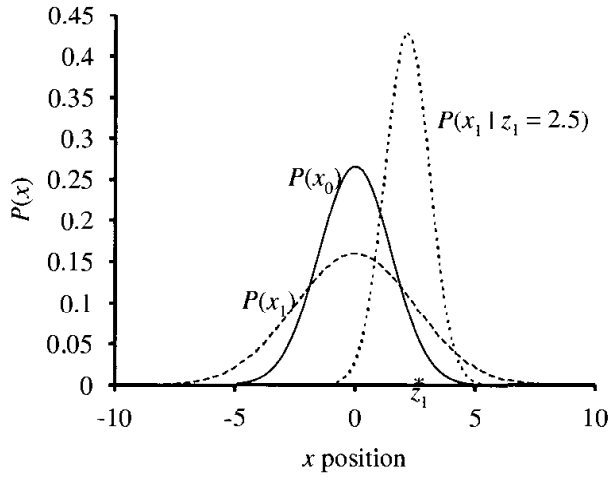


Figure 15.8 Stages in the Kalman filter update cycle for a random walk with a prior given by $\mu_0 = 0.0$ and $\sigma_0 = 1.0$, transition noise given by $\sigma_x = 2.0$, sensor noise given by $\sigma_z = 1.0$, and a first observation $z_1 = 2.5$ (marked on the x-axis). Notice how the prediction $P(x_1)$ 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.

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 (15.16), 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

$$P(x_1|z_1) = \alpha\, e^{-\frac{1}{2}\left(\frac{(x_1-\frac{(\sigma_0^2+\sigma_x^2)z_1+\sigma_z^2\mu_0}{\sigma_0^2+\sigma_x^2+\sigma_z^2})^2}{(\sigma_0^2+\sigma_x^2)\sigma_z^2/(\sigma_0^2+\sigma_x^2+\sigma_z^2)}\right)} . \tag{15.17}$$

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

From the Gaussian formula in Equation (15.17), we see that the new mean and standard deviation can be calculated from the old mean and standard deviation as follows:

$$\begin{aligned} \mu_{t+1} &= \frac{(\sigma_t^2+\sigma_x^2)z_{t+1}+\sigma_z^2\mu_t}{\sigma_t^2+\sigma_x^2+\sigma_z^2} \\ \sigma_{t+1}^2 &= \frac{(\sigma_t^2+\sigma_x^2)\sigma_z^2}{\sigma_t^2+\sigma_x^2+\sigma_z^2} . \end{aligned} \tag{15.18}$$

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 (15.3) or the HMM filtering equation (15.10). 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 converges quickly 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 illustrates the key property of Gaussian distributions that allows Kalman filtering to work: the fact that the exponent is a quadratic form. 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 makes it 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

$$\begin{aligned} P(\mathbf{x}_{t+1}|\mathbf{x}_t) &= N(\mathbf{F}\mathbf{x}_t, \boldsymbol{\Sigma}_x)(\mathbf{x}_{t+1}) \\ P(\mathbf{z}_t|\mathbf{x}_t) &= N(\mathbf{H}\mathbf{x}_t, \boldsymbol{\Sigma}_z)(\mathbf{z}_t) , \end{aligned} \tag{15.19}$$

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

$$\begin{aligned} \boldsymbol{\mu}_{t+1} &= \mathbf{F}\boldsymbol{\mu}_t + \mathbf{K}_{t+1}(\mathbf{z}_{t+1} - \mathbf{H}\mathbf{F}\boldsymbol{\mu}_t) \\ \boldsymbol{\Sigma}_{t+1} &= (\mathbf{I} - \mathbf{K}_{t+1})(\mathbf{F}\boldsymbol{\Sigma}_t\mathbf{F}^\top + \boldsymbol{\Sigma}_x) \end{aligned} \tag{15.20}$$

where $\mathbf{K}_{t+1} = (\mathbf{F}\boldsymbol{\Sigma}_t\mathbf{F}^\top + \boldsymbol{\Sigma}_x)\mathbf{H}^\top(\mathbf{H}(\mathbf{F}\boldsymbol{\Sigma}_t\mathbf{F}^\top + \boldsymbol{\Sigma}_x)\mathbf{H}^\top + \boldsymbol{\Sigma}_z)^{-1}$ 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 $\mathbf{H}\mathbf{F}\boldsymbol{\mu}_t$ is the *predicted* observation. Therefore, the term $\mathbf{z}_{t+1} - \mathbf{H}\mathbf{F}\boldsymbol{\mu}_t$ represents the error in the predicted observation. This is multiplied by $\mathbf{K}_{t+1}$ to correct the predicted state; hence $\mathbf{K}_{t+1}$ is a measure of *how seriously to take the new observation* relative to the prediction. As in Equation (15.18), we also have the property that the variance update is independent of the observations. The sequence of values for $\boldsymbol{\Sigma}_t$ and $\mathbf{K}_t$ can therefore be computed offline, and the actual calculations required during online tracking are quite modest.

KALMAN GAIN MATRIX

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 good job of tracking the actual motion, and, as expected, the variance quickly reaches a fixed point.

We 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.

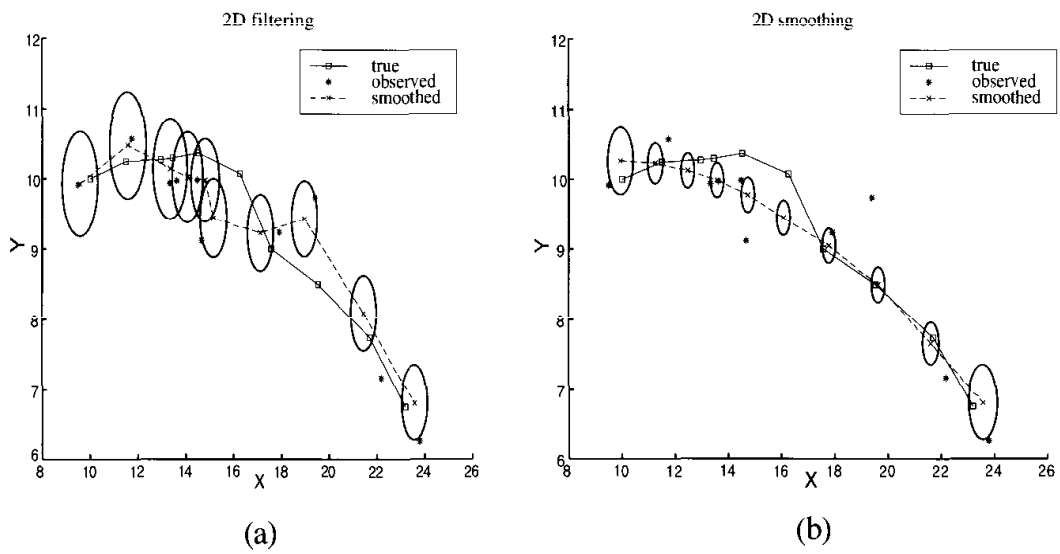


Figure 15.9 (a) Results of Kalman filtering for an object moving on the X–Y plane, showing the true trajectory (left to right), a series of noisy observations, and the trajectory estimated by Kalman filtering. Variance in the position estimate is indicated by the ovals. (b) The results of Kalman smoothing 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—a linear Gaussian transition and sensor models—are very strong. The **extended Kalman filter (EKF)** attempts to overcome nonlin-

EXTENDED KALMAN FILTER (EKF)

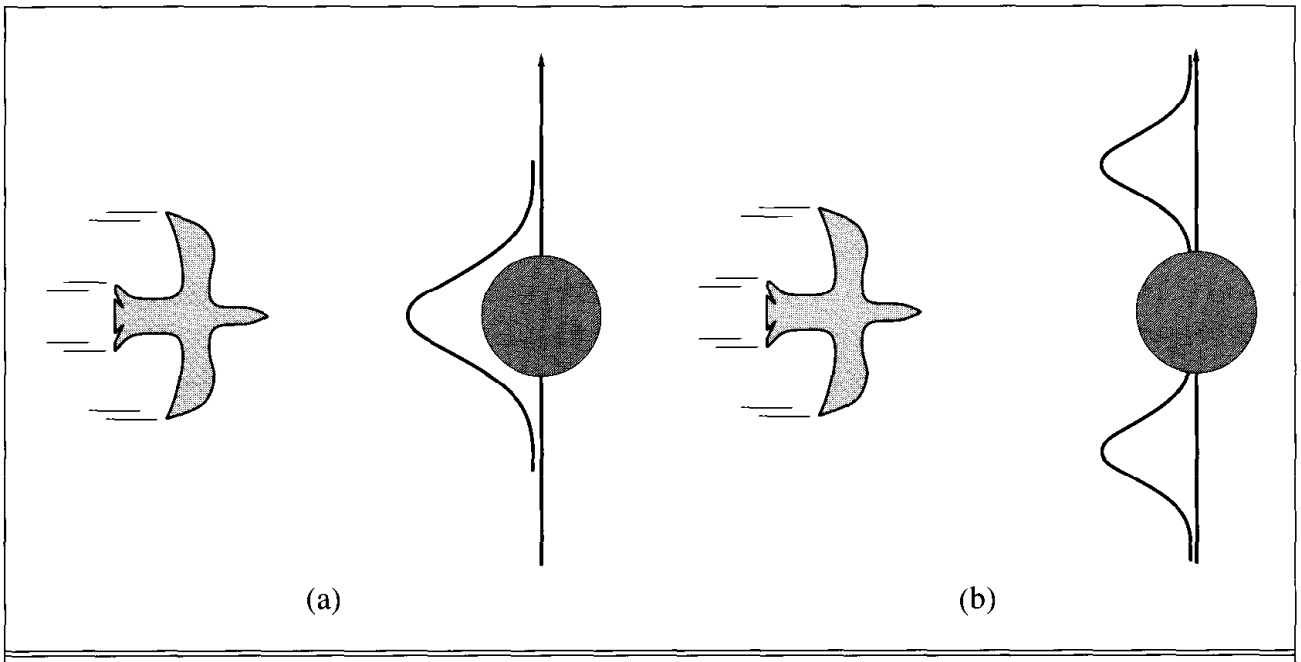


Figure 15.10 A bird flying toward a tree (top views). (a) A Kalman filter will predict the location of the bird using a single Gaussian centered on the obstacle. (b) A more realistic model allows for the bird's evasive action, predicting that it will fly to one side or the other.

earities in the system being modeled. A system is nonlinear if the transition model cannot be described as a matrix multiplication of the state vector, as in Equation (15.19). The EKF works by modeling 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.

What does it mean for a system to be "unsmooth" or "poorly behaved"? Technically, it 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 idea 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 make only 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, as 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.

In order to handle examples like these, we clearly need a more expressive language for representing the behavior of the system being modeled. Within the control theory community, for which 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, and 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

SWITCHING KALMAN FILTER

will see in the next section that this is simply a special case of the general dynamic Bayesian network model, obtained 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 is bad for at least three 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 location. DBNs, on the other hand, can model 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" that require combinations of discrete and continuous variables in order to get reasonable models.

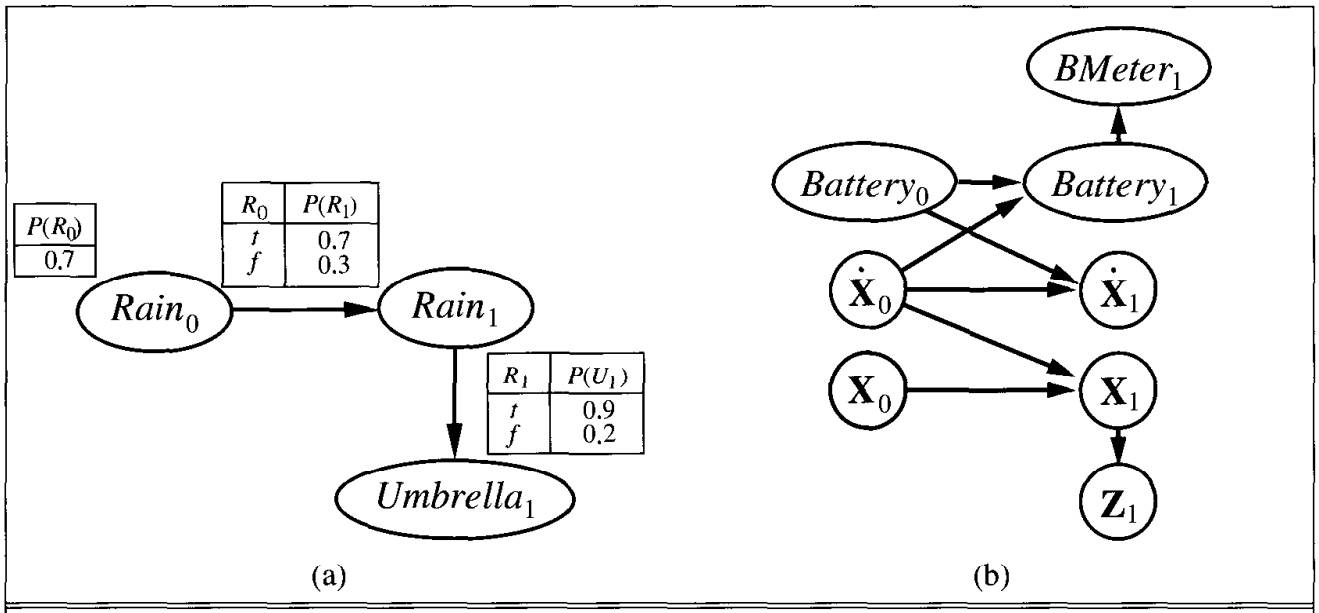


Figure 15.11 (a) Specification of the prior, transition model, and sensor model for the umbrella DBN. All subsequent slices are assumed to be copies of slice 1. (b) A simple DBN for robot motion in the X–Y plane.

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—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 20 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 a high probability to the battery's 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 that 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 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

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

which is presumably much larger than the probability assigned by the simple Gaussian error

[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.

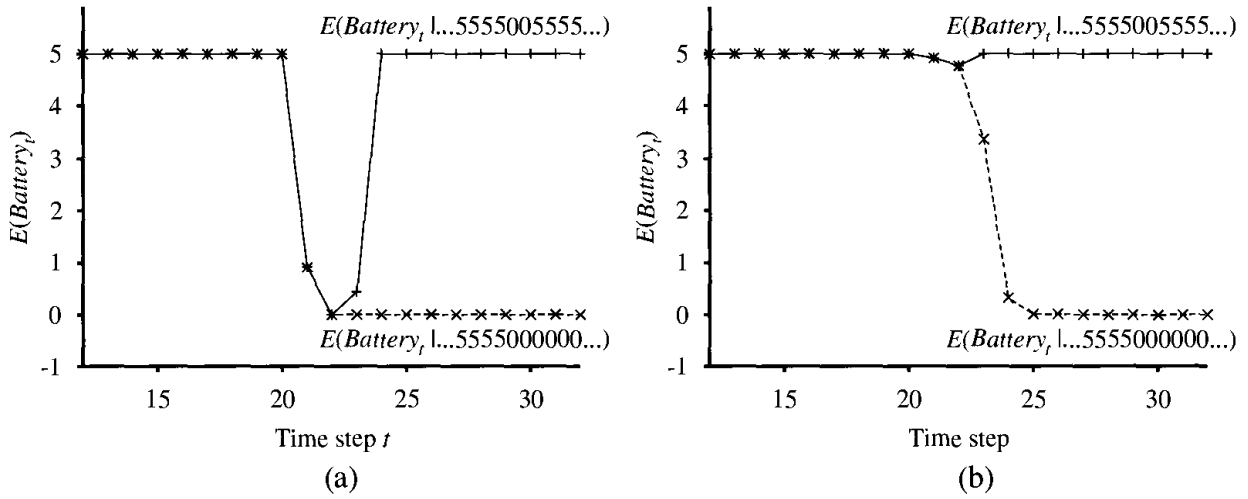


Figure 15.12 (a) Upper curve: trajectory of the expected value of $Battery_t$ for an observation sequence consisting of all 5s except for 0s at $t = 21$ and $t = 22$, using a simple Gaussian error model. Lower curve: trajectory when the observation remains at 0 from $t = 21$ onwards. (b) The same experiment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

TRANSIENT FAILURE MODEL

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

PERSISTENCE ARC

modeled 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.

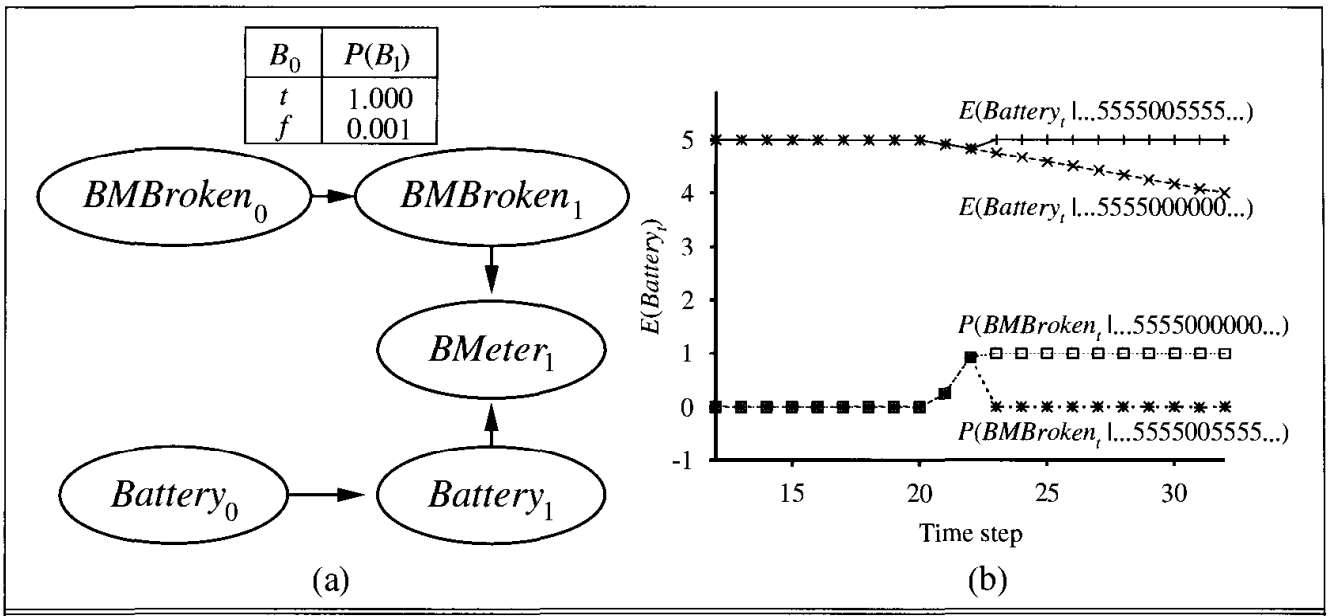


Figure 15.13 (a) A DBN fragment showing the sensor status variable required for modeling persistent failure of the battery sensor. (b) Upper curves: trajectories of the expected value of $Battery_t$ for the "transient failure" and "permanent failure" observations sequences. Lower curves: probability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

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 $E(Battery_t|\ldots)$.

So far, we have merely scratched the surface of the problem of representing complex processes. The variety of transition models is huge, encompassing topics as disparate as modeling the human endocrine system and modeling multiple vehicles driving on a freeway. Sensor modeling 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 technique is called **unrolling**. (Technically, the DBN is equivalent to the semi-infinite network obtained by unrolling forever. Slices added

UNROLLING

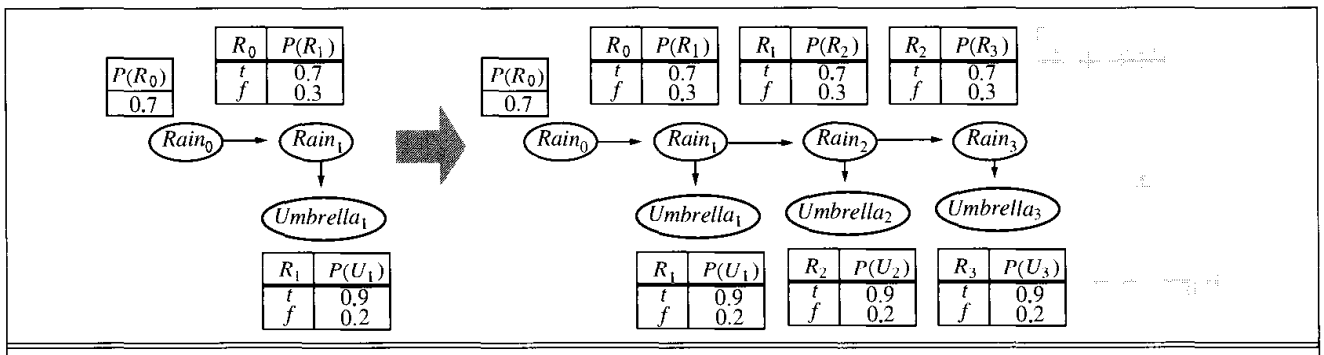


Figure 15.14 Unrolling a dynamic Bayesian network: slices are replicated to accommodate the observation sequence (shaded nodes). Further slices have no effect on inferences within the observation period.

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.

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 would thus grow without bound as more observations were 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 (15.3) 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 (15.3). 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 $O(d^{n+2})$.

Of course, 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 distribu-

tion 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 non-evidence 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 to have in memory only the current slice and the next slice.

In our discussion of likelihood weighting in Chapter 14, we pointed out that the algorithm's accuracy suffers if the evidence variables are "downstream" from 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 could 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. Given that a filtering algorithm that works in real time can only use a fixed number of samples, what happens in practice is that the error blows up after a very small number of update steps.

Clearly, we need a better solution. 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

```
function PARTICLE-FILTERING(e, N, dbn) returns a set of samples for the next time step
  inputs: e, the new incoming evidence
          N, the number of samples to be maintained
          dbn, a DBN with prior P(X_0), transition model P(X_1|X_0), and sensor model
P(E_1|X_1)
  static: S, a vector of samples of size N, initially generated from P(X_0)
  local variables: W, a vector of weights of size N

  for i = 1 to N do
     S[i] ← sample from P(X_1|X_0 = S[i])
     W[i] ← P(e|X_1 = S[i])
  S ← WEIGHTED-SAMPLE-WITH-REPLACEMENT(N, S, W)
  return S
```

Figure 15.15 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.

those that have high weight. In that way, the population of samples will stay reasonably close to reality. If we think of samples as a resource for modeling 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 that. 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, and 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 15.15, and its operation for the umbrella DBN is illustrated in Figure 15.16.

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 that the sample population starts with a correct representation of the forward message $\mathbf{f}_{1:t}$ at time t. 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

$$N(\mathbf{x}_t|\mathbf{e}_{1:t})/N = P(\mathbf{x}_t|\mathbf{e}_{1:t}) \tag{15.21}$$

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

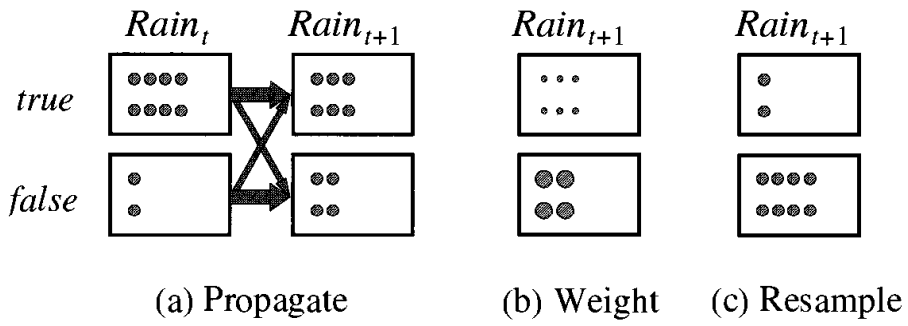


Figure 15.16 The particle filtering update cycle for the umbrella DBN with $N = 10$, showing the sample populations of each state. (a) At time t, 8 samples indicate *Rain* and 2 indicate $\neg$*Rain*. Each is propagated forward by sampling the next state via the transition model. At time $t + 1$, 6 samples indicate *Rain* and 4 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 2 samples that indicate *Rain* and 8 that indicate $\neg$*Rain*.

$\mathbf{x}_t$ 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:

$$\begin{aligned}
N(\mathbf{x}_{t+1}|\mathbf{e}_{1:t+1})/N &= \alpha\, W(\mathbf{x}_{t+1}|\mathbf{e}_{1:t+1}) \\
&= \alpha\, P(\mathbf{e}_{t+1}|\mathbf{x}_{t+1})N(\mathbf{x}_{t+1}|\mathbf{e}_{1:t}) \\
&= \alpha\, P(\mathbf{e}_{t+1}|\mathbf{x}_{t+1}) \sum_{\mathbf{x}_t} P(\mathbf{x}_{t+1}|\mathbf{x}_t)N(\mathbf{x}_t|\mathbf{e}_{1:t}) \\
&= \alpha\, N P(\mathbf{e}_{t+1}|\mathbf{x}_{t+1}) \sum_{\mathbf{x}_t} P(\mathbf{x}_{t+1}|\mathbf{x}_t)P(\mathbf{x}_t|\mathbf{e}_{1:t}) \quad \text{(by 15.21)} \\
&= \alpha' P(\mathbf{e}_{t+1}|\mathbf{x}_{t+1}) \sum_{\mathbf{x}_t} P(\mathbf{x}_{t+1}|\mathbf{x}_t)P(\mathbf{x}_t|\mathbf{e}_{1:t}) \\
&= P(\mathbf{x}_{t+1}|\mathbf{e}_{1:t+1}) \quad \text{(by 15.3).}
\end{aligned}$$

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 that 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 set 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.*, 2000).

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 a 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

$P(signal|words)$ is the **acoustic model**. It describes the sounds of words—that "ceiling" begins with a soft "c" and sounds the same as "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."

HOMOPHONES

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 (e.g., the "a" in *rat* and *rate*. Figure 15.17 lists phones that are used English with an example of each. A **phoneme** is the smallest unit of sound that has a distinct meaning to speakers of a particular language. For example, in English the "t" phone in "stick" is the same phoneme as the "t" phone in "tick,"

PHONOLOGY

PHONES

PHONEME

Vowels		Consonants B–N		Consonants P–Z	
Phone	Example	Phone	Example	Phone	Example
[iy]	beat	[b]	bet	[p]	pet
[ih]	bit	[ch]	Chet	[r]	rat
[eh]	bet	[d]	debt	[s]	set
[æ]	bat	[f]	fat	[sh]	shoe
[ah]	but	[g]	get	[t]	ten
[ao]	bought	[hh]	hat	[th]	thick
[ow]	boat	[hv]	high	[dh]	that
[uh]	book	[jh]	jet	[dx]	butter
[ey]	bait	[k]	kick	[v]	vet
[er]	Bert	[l]	let	[w]	wet
[ay]	buy	[el]	bottle	[wh]	which
[oy]	boy	[m]	met	[y]	yet
[axr]	diner	[em]	bottom	[z]	zoo
[aw]	down	[n]	net	[zh]	measure
[ax]	about	[en]	button		
[ix]	roses	[ng]	sing		
[aa]	cot	[eng]	washing	[-]	*silence*

Figure 15.17 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.

but in Thai they are distinguishable as two separate phonemes.

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.

PRONUNCIATION

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. The 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.

SIGNAL PROCESSING

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.

Speech sounds

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. (High-quality music recordings sample at a rate of 44 kHz or more.) 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 acoustic signal.

SAMPLING RATE

QUANTIZATION FACTOR

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 the 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 15.18 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.

FRAMES

FEATURES

In our example, we have shown frames with just three features. Real systems 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

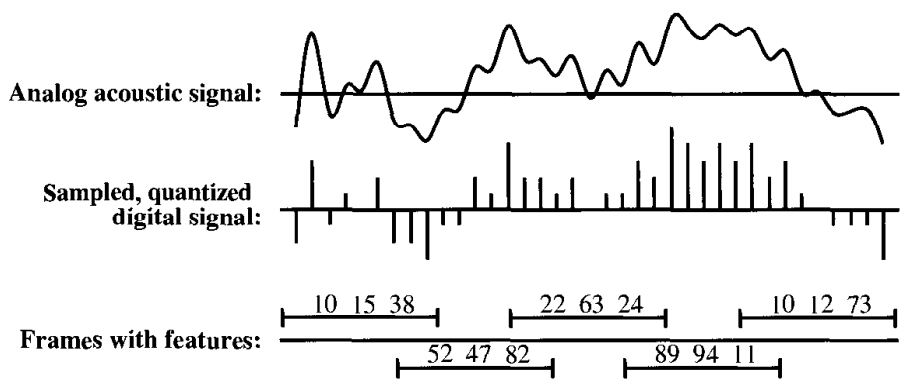


Figure 15.18 Translating the acoustic signal into a sequence of frames; each frame is described by the values of three acoustic features.

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 labeled C1 through C256. Each frame can then be represented with a single label rather than a vector of n numbers. Thus, the tabulated distribution $P(VQ|phone)$ 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.

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 of noise, and by any of millions of different speakers, each with a different accent and vocal tract. Signal processing hopes to eliminate the variations while keeping the commonalities that define the sound.[5]

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 be changed by the surrounding phones.[6] Remember that speech sounds are produced by moving the lips, tongue, and jaw and forcing air through the vocal tract. To coordinate these complex movements at the rate of five or more phones per second, the brain initiates action for a second phone before the first is completed, thereby altering one or both phones. For example, in pronouncing "sweet" the lips are rounded during the production of [s] in anticipation of the following [w]. These **coarticulation effects** are partially captured

COARTICULATION EFFECTS

[5] The complementary problem, **speaker identification**, eliminates the commonalities and keeps the individual differences, and then tries to match the differences to models of individual speakers.

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

TRIPHONE

by the **triphone** model, in which the acoustic model for each phone is allowed to depend on the preceding and succeeding phones. Thus, the [w] in "sweet is written [w(s,iy)], i.e., [w] with left-context [s] and right-context [iy].

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." According to Gershwin (1937), you say [t ow m ey t ow] and I say [t ow m aa t ow]. The top of Figure 15.19 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:

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

The second source of phonetic variation is **coarticulation**. 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 15.19 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 15.20. 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 instance, the $[\text{m}]_{\text{Mid}}$ state persists with probability 0.9, which means that the $[\text{m}]_{\text{Mid}}$ state has an expected duration of 10 frames. In the model we have, the duration of each phone is independent of the duration of other phones; a more sophisticated model could distinguish between 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.

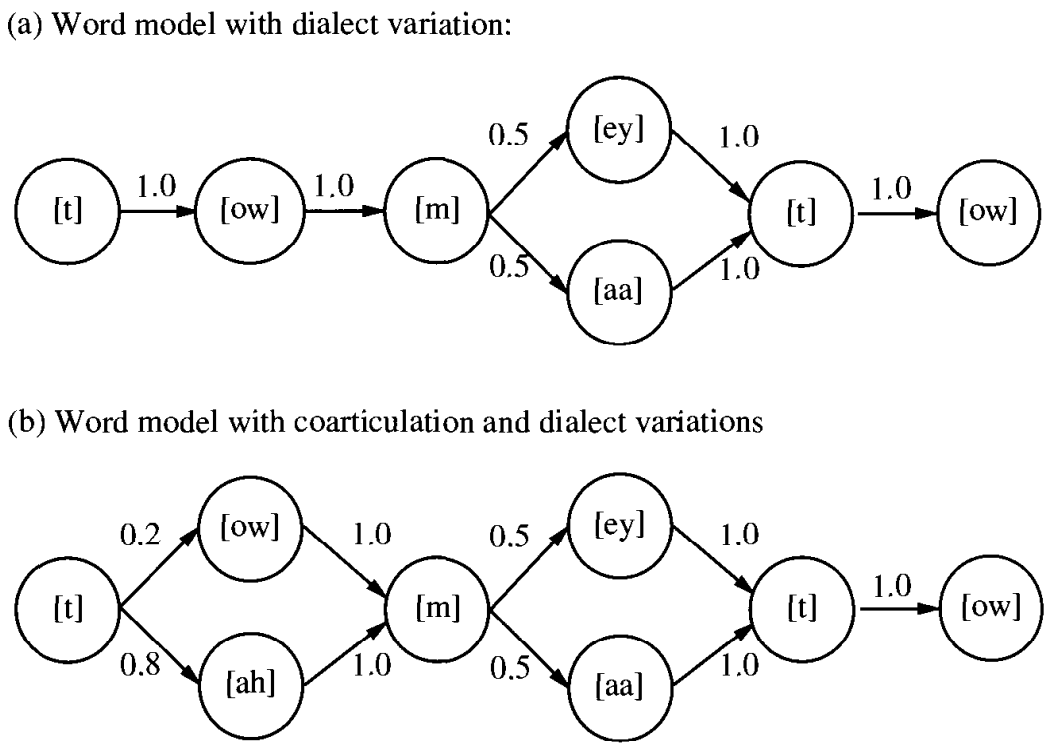
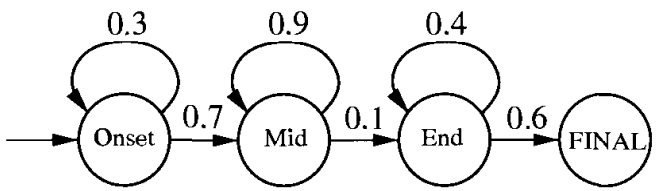


Figure 15.19 Two pronunciation models of the word "tomato." Each model is shown as a transition diagram with states as circles and arrows showing allowed transitions with their associated probabilities. (a) A model allowing for dialect differences. The 0.5 numbers are estimates based on the two authors' preferred pronunciations. (b) A model with a coarticulation effect on the first vowel, allowing either the [ow] or the [ah] phone.

Phone HMM for [m]:



Output probabilities for the phone HMM:

Onset:	Mid:	End:
C_1: 0.5	C_3: 0.2	C_4: 0.1
C_2: 0.2	C_4: 0.7	C_6: 0.5
C_3: 0.3	C_5: 0.1	C_7: 0.4

Figure 15.20 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.

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. $P(e_{1:t}|word)$ 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 (15.5) 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 547) 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 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.

SEGMENTATION

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 with the use of the chain rule as follows:[7]

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

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.

BIGRAM

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

[7] 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.

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 15.21 A partial table of unigram and bigram counts for the words in this book. "The" is the most common single word with a count of 33,508 (out of 513,893 total words). The bigram "of the" is the most common, at 3,833. 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."

counts to estimate the probabilities. For example, if "a" appears 10,000 times in the training 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 high estimated probabilities, while "I has" and "an gun" would have low probabilities. Figure 15.21 shows some bigram counts derived from the words in this book.

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 judging that "ate a banana" is more likely than "ate a bandanna." For trigram models and to a lesser extent for bigram and unigram models, there is a problem with counts of zero: We wouldn't want to say that a combination of words is impossible just because they didn't happen to appear in the training corpus. The process of **smoothing** gives a small non-zero probability to such combinations. It is discussed on page 835.

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 be detected only locally: "the man have" gets a low score, but "the man with the yellow hat 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 composed in turn of pronunciation models and phone models) into one large HMM model. A state in a single-word HMM is a frame labeled by the current phone and phone state (for example, $[\mathrm{m}]_{\mathrm{Onset}}$); a state in a continuous-speech HMM is also labeled with a word, as in $[\mathrm{m}]^{\mathrm{tomato}}_{Onset}$. 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 $3pW$ 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 (15.9) can be used to find the most likely state sequence. From this state sequence, we can then 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 the 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.

A* DECODER

In practice, this difficulty is not life-threatening, but it is serious enough that other 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 labeled 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 labeled w_1 to w_2 is given by $-\log P(w_2|w_1)$; in this way, the total path cost of a sequence is

$$Cost(w_1 \cdots w_n) = \sum_{i=1}^{n} -\log P(w_i|w_{i-1}) = -\log \prod_{i=1}^{n} P(w_i|w_{i-1}) \ .$$

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 of 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 can be constructed, and we leave the details of signal processing to other textbooks. We are left with the pronunciation and phone models. The *structure* of the pronunciation models—such as the tomato models in Figure 15.19—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 15.20. That leaves the probabilities themselves. How are these to be obtained, given that the models could 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 20, but we can present 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 it 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 labeling 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 by using statistical fitting methods (for mixture-of-Gaussian models; see Chapter 20).

The cost and rarity of hand-labeled data, and the fact that the available hand-labeled data sets might 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 labeled data.* Estimates derived from hand-labeled 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*. 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 it generally converges to a much better set of parameter values than those provided by the initial, hand-labeled 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 by 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**—that is, 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), who formulated 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 20 years, its use of a frequency-domain representation made many calculations quite cumbersome. Direct state-space modeling of the stochastic process turned out to be simpler, as shown by Swerling (1959) and 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. Chatfield (1989) covers the "classical" approach to time series analysis.

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 (Dempster *et al.*, 1977); see also Chapter 20. Constant-space smoothing appears in Binder *et al.* (1997b), as does the divide-and-conquer algorithm developed in Exercise 15.3.

Dynamic Bayesian 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 Bayesian network generation and compilation. Dynamic Bayesian networks have become popular for modeling 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 (and 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 by Doucet (1997) to reduce the approximation error.

Other 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. Properties of mixing times are discussed by Pak (2001) and by (Luby and Vigoda, 1999).

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 built for recognizing isolated digits (Davis *et al.*, 1952) by means of 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).[8] The final version of HARPY was derived from a system called DRAGON built by CMU graduate student James Baker (1975); DRAGON 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.

[8] 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** that communicated via a common **blackboard** from which they could write and read. Blackboard systems are the foundation of modern user interface architectures.

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 $k = 1 \ldots 20$ 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 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 the input into more than two pieces?

15.4 On page 547, 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).

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 can 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 15.22.

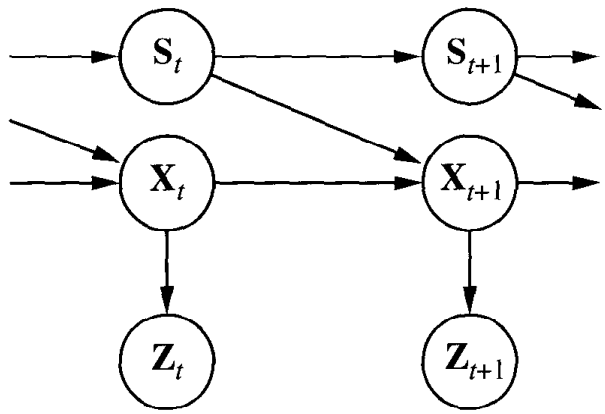


Figure 15.22 A Bayesian network representation of a switching Kalman filter. The switching variable S_t is a discrete state variable whose value determines the transition model for the continuous state variables $\mathbf{X}_t$. For any discrete state i, the transition model $\mathbf{P}(\mathbf{X}_{t+1}|\mathbf{X}_t, S_t = i)$ 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, as in a hidden Markov model.

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 in the general case the updated state estimate $\mathbf{P}(\mathbf{X}_{t+1}|\mathbf{e}_{1:t+1})$ will be a mixture of km Gaussians.

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 (15.17), the first update step for the one-dimensional Kalman filter.

15.7 Let us examine the behavior of the variance update in Equation (15.18).

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 $\sigma_t^2 \rightarrow \sigma^2$ as $t \rightarrow \infty$, and calculate the value of σ^2.

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

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 $t \rightarrow \infty$ 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, regardless of whether the rain variables are eliminated in forward or backward order.

15.11 The model of "tomato" in Figure 15.19 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.

15.12 Calculate the most probable path through the HMM in Figure 15.20 for the output sequence $[C_1, C_2, C_3, C_4, C_4, C_6, C_7]$. Also give its probability.

16 MAKING SIMPLE DECISIONS

In which we see how an agent should make decisions so that it gets what it wants—on average, at least.

In this chapter, we return to the idea of utility theory that was introduced in Chapter 13 and show how it is combined with probability theory to yield a decision-theoretic agent—an agent that can make rational decisions based on what it believes and what it wants. Such an agent can make decisions in contexts where uncertainty and conflicting goals leave a logical agent with no way to decide. In effect, a goal-based agent has a binary distinction between good (goal) and bad (non-goal) states, while a decision-theoretic agent has a continuous measure of state quality.

Section 16.1 introduces the basic principle of decision theory: the maximization of expected utility. Section 16.2 shows that the behavior of any rational agent can be captured by supposing a utility function that is being maximized. Section 16.3 discusses the nature of utility functions in more detail, and in particular their relation to individual quantities such as money. Section 16.4 shows how to handle utility functions that depend on several quantities. In Section 16.5, we describe the implementation of decision-making systems. In particular, we introduce a formalism called **decision networks** (also known as **influence diagrams**) that extends Bayesian networks by incorporating actions and utilities. The remainder of the chapter discusses issues that arise in applications of decision theory to expert systems.

16.1 COMBINING BELIEFS AND DESIRES UNDER UNCERTAINTY

In the *Port-Royal Logic*, written in 1662, the French philosopher Arnauld stated

> To judge what one must do to obtain a good or avoid an evil, it is necessary to consider not only the good and the evil in itself, but also the probability that it happens or does not happen; and to view geometrically the proportion that all these things have together.

Modern texts talk of utility rather than good and evil, but the principle is exactly the same. An agent's preferences between world states are captured by a **utility function**, which assigns

a single number to express the desirability of a state. Utilities are combined with outcome probabilities of actions to give an expected utility for each action.

We will use the notation $U(S)$ to denote the utility of state S according to the agent that is making the decisions. For now, we will consider states as complete snapshots of the world, similar to the **situations** of Chapter 10. This will simplify our initial discussions, but it can become rather cumbersome to specify the utility of each possible state separately. In Section 16.4, we will see how states can be decomposed under some circumstances for the purpose of assigning utilities.

A nondeterministic action A will have possible outcome states $Result_i(A)$, where the index i ranges over the different outcomes. Prior to the execution of A, the agent assigns probability $P(Result_i(A)|Do(A), E)$ to each outcome, where E summarizes the agent's available evidence about the world and $Do(A)$ is the proposition that action A is executed in the current state. Then we can calculate the **expected utility** of the action given the evidence, $EU(A|E)$, using the following formula:

EXPECTED UTILITY

$$EU(A|E) = \sum_i P(Result_i(A)|Do(A), E)\, U(Result_i(A)) \,. \tag{16.1}$$

MAXIMUM EXPECTED UTILITY

The principle of **maximum expected utility** (MEU) says that a rational agent should choose an action that maximizes the agent's expected utility. If we wanted to choose the best sequence of actions using this equation we would have to enumerate all action sequences and choose the best; this is clearly infeasible for long sequences. Therefore, this chapter will focus on simple decisions (usually a single action) and the next chapter will introduce new techniques for efficiently dealing with action sequences.

In a sense, the MEU principle could be seen as defining all of AI. All an intelligent agent has to do is calculate the various quantities, maximize utility over its actions, and away it goes. But this does not mean that the AI problem is *solved* by the definition!

Although the MEU principle defines the right action to take in any decision problem, the computations involved can be prohibitive, and it is sometimes difficult even to formulate the problem completely. Knowing the initial state of the world requires perception, learning, knowledge representation, and inference. Computing $P(Result_i(A)|Do(A), E)$ requires a complete causal model of the world and, as we saw in Chapter 14, NP-hard inference in Bayesian networks. Computing the utility of each state, $U(Result_i(A))$, often requires searching or planning, because an agent does not know how good a state is until it knows where it can get to from that state. So, decision theory is not a panacea that solves the AI problem. On the other hand, it does provide a framework into which we can see where all the components of an AI system fit.

The MEU principle has a clear relation to the idea of performance measures introduced in Chapter 2. The basic idea is very simple. Consider the environments that could lead to an agent having a given percept history, and consider the different agents that we could design. *If an agent maximizes a utility function that correctly reflects the performance measure by which its behavior is being judged, then it will achieve the highest possible performance score if we average over the environments in which the agent could be placed.* This is the central justification for the MEU principle itself. While the claim may seem tautological, it does in fact embody a very important transition from a global, external criterion of rationality—the

performance measure over environment histories—to a local, internal criterion involving the maximization of a utility function applied to the next state.

ONE-SHOT DECISIONS

In this chapter, we will be concerned only with single or **one-shot decisions**, whereas Chapter 2 defined performance measures over environment histories, which usually involve many decisions. In the next chapter, which covers the case of **sequential decisions**, we will show how these two views can be reconciled.

16.2 The Basis of Utility Theory

Intuitively, the principle of Maximum Expected Utility (MEU) seems like a reasonable way to make decisions, but it is by no means obvious that it is the *only* rational way. After all, why should maximizing the *average* utility be so special? Why not try to maximize the sum of the cubes of the possible utilities, or try to minimize the worst possible loss? Also, couldn't an agent act rationally just by expressing preferences between states, without giving them numeric values? Finally, why should a utility function with the required properties exist at all? Perhaps a rational agent can have a preference structure that is too complex to be captured by something as simple as a single real number for each state.

Constraints on rational preferences

These questions can be answered by writing down some constraints on the preferences that a rational agent should have and then showing that the MEU principle can be derived from the constraints. We use the following notation to describe an agent's preferences:

$A \succ B$ A is preferred to B.

$A \sim B$ the agent is indifferent between A and B.

$A \succsim B$ the agent prefers A to B or is indifferent between them.

Now the obvious question is, what sorts of things are A and B? If the agent's actions are deterministic, then A and B will typically be the concrete, fully specified outcome states of those actions. In the more general, nondeterministic case, A and B will be **lotteries**. A lottery is essentially a probability distribution over a set of actual outcomes (the "prizes" of the lottery). A lottery L with possible outcomes $C_1, \ldots, C_n$ that can occur with probabilities $p_1, \ldots, p_n$ is written

LOTTERIES

$$L = [p_1, C_1;\ p_2, C_2;\ \ldots\ p_n, C_n]\ .$$

(A lottery with only one outcome can be written either as A or as $[1, A]$.) In general, each outcome of a lottery can be either an atomic state or another lottery. The primary issue for utility theory is to understand how preferences between complex lotteries are related to preferences between the underlying states in those lotteries.

To do this, we impose reasonable constraints on the preference relation, much as we imposed rationality constraints on degrees of belief in order to obtain the axioms of probability in Chapter 13. One reasonable constraint is that preference should be **transitive**: that is, if $A \succ B$ and $B \succ C$, then we would expect that $A \succ C$. We argue for transitivity by showing

that an agent whose preferences do not respect transitivity would behave irrationally. Suppose, for example, that an agent has the nontransitive preferences $A \succ B \succ C \succ A$, where A, B, and C are goods that can be freely exchanged. If the agent currently has A, then we could offer to trade C for A and some cash. The agent prefers C, and so would be willing to give up some amount of cash to make this trade. We could then offer to trade B for C, extracting more cash, and finally trade A for B. This brings us back where we started from, except that the agent has less money (Figure 16.1(a)). We can keep going around the cycle until the agent has no money at all. Clearly, this case the agent has not acted rationally.

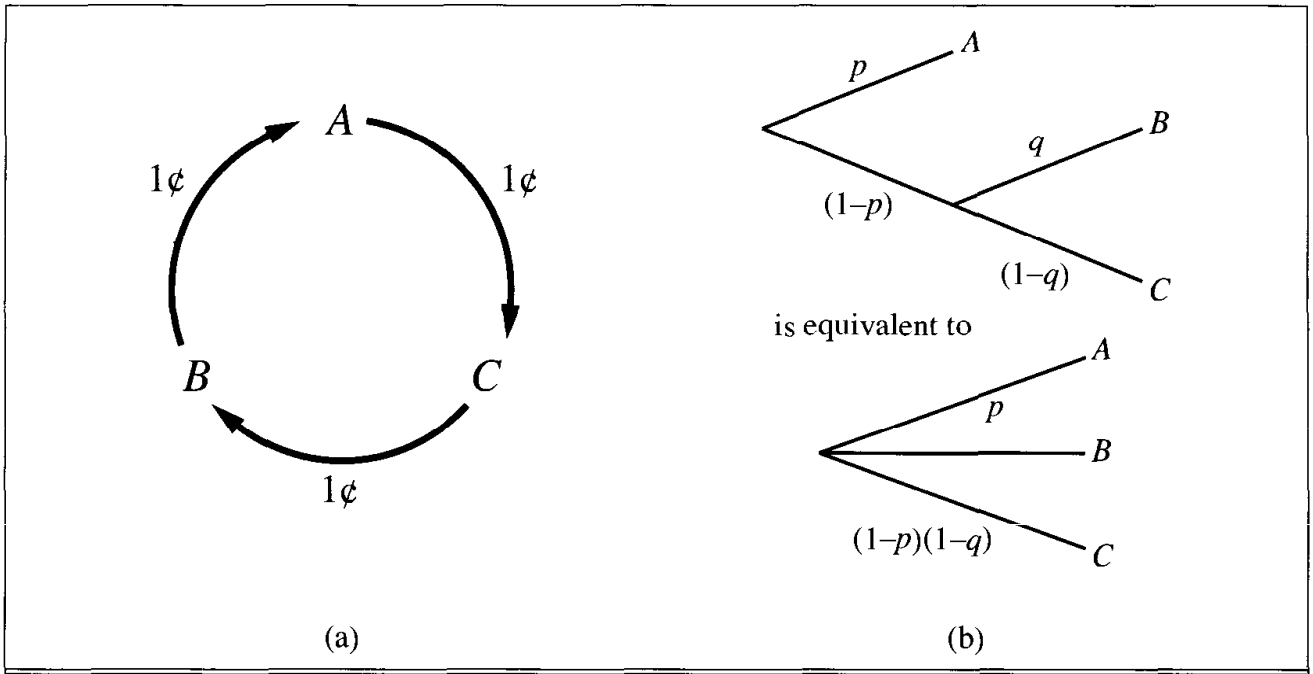


Figure 16.1 (a) A cycle of exchanges showing that the nontransitive preferences $A \succ B \succ C \succ A$ result in irrational behavior. (b) The decomposability axiom.

The following six constraints are known as the axioms of utility theory. They specify the most obvious semantic constraints on preferences and lotteries.

ORDERABILITY

◇ **Orderability**: Given any two states, a rational agent must either prefer one to the other or else rate the two as equally preferable. That is, the agent cannot avoid deciding. As we said on page 474, refusing to bet is like refusing to allow time to pass.

$$(A \succ B) \vee (B \succ A) \vee (A \sim B) .$$

TRANSITIVITY

◇ **Transitivity**: Given any three states, if an agent prefers A to B and prefers B to C, then the agent must prefer A to C.

$$(A \succ B) \wedge (B \succ C) \Rightarrow (A \succ C) .$$

CONTINUITY

◇ **Continuity**: If some state B is between A and C in preference, then there is some probability p for which the rational agent will be indifferent between getting B for sure and the lottery that yields A with probability p and C with probability $1 - p$.

$$A \succ B \succ C \Rightarrow \exists p \ [p, A;\ 1 - p, C] \sim B .$$

SUBSTITUTABILITY

◇ **Substitutability**: If an agent is indifferent between two lotteries, A and B, then the agent is indifferent between two more complex lotteries that are the same except that B is substituted for A in one of them. This holds regardless of the probabilities and the other outcome(s) in the lotteries.

$$A \sim B \;\Rightarrow\; [p, A;\; 1-p, C] \sim [p, B; 1-p, C]\,.$$

MONOTONICITY

◇ **Monotonicity**: Suppose there are two lotteries that have the same two outcomes, A and B. If an agent prefers A to B, then the agent must prefer the lottery that has a higher probability for A (and vice versa).

$$A \succ B \;\Rightarrow\; (p \geq q \;\Leftrightarrow\; [p, A;\; 1-p, B] \succsim [q, A;\; 1-q, B])\,.$$

DECOMPOSABILITY

◇ **Decomposability**: Compound lotteries can be reduced to simpler ones using the laws of probability. This has been called the "no fun in gambling" rule because it says that two consecutive lotteries can be compressed into a single equivalent lottery, as shown in Figure 16.1(b).[1]

$$[p, A;\; 1-p, [q, B;\; 1-q, C]] \sim [p, A;\; (1-p)q, B;\; (1-p)(1-q), C]\,.$$

And then there was Utility

Notice that the axioms of utility theory do not say anything about utility: They talk only about preferences. Preference is assumed to be a basic property of rational agents. The existence of a utility function *follows* from the axioms of utility:

1. **Utility principle**
   If an agent's preferences obey the axioms of utility, then there exists a real-valued function U that operates on states such that $U(A) > U(B)$ if and only if A is preferred to B, and $U(A) = U(B)$ if and only if the agent is indifferent between A and B.

   $$U(A) > U(B) \;\Leftrightarrow\; A \succ B\,;$$
   $$U(A) = U(B) \;\Leftrightarrow\; A \sim B\,.$$

2. **Maximum Expected Utility principle**
   The utility of a lottery is the sum of the probability of each outcome times the utility of that outcome.

   $$U([p_1, S_1; \ldots; p_n, S_n]) = \sum_i p_i U(S_i)\,.$$

In other words, once the probabilities and utilities of the possible outcome states are specified, the utility of a compound lottery involving those states is completely determined. Because the outcome of a nondeterministic action is a lottery, this gives us the MEU decision rule from Equation (16.1).

It is important to remember that the existence of a utility function that describes an agent's preference behavior does not necessarily mean that the agent is *explicitly* maximizing that utility function in its own deliberations. As we showed in Chapter 2, rational behavior can be generated in any number of ways, some of which are more efficient than explicit

[1] We can account for the enjoyment of gambling by encoding gambling events into the state description; for example, "Have \$10 and gambled" could be preferred to "Have \$10 and didn't gamble."

utility maximization. By observing a rational agent's preferences, however, it is possible to construct the utility function that represents what it is that the agent's actions are actually trying to achieve.

16.3 Utility Functions

Utility is a function that maps from states to real numbers. Is that all we can say about utility functions? Strictly speaking, that is it. Beyond the constraints listed earlier, an agent can have any preferences it likes. For example, an agent might prefer to have a prime number of dollars in its bank account; in which case, if it had \$16 it would give away \$3. It might prefer a dented 1973 Ford Pinto to a shiny new Mercedes. Preferences can also interact: for example, it might only prefer prime numbers of dollars when it owns the Pinto, but when it owns the Mercedes, it might prefer more dollars to less.

If all utility functions were as arbitrary as this, however, then utility theory would not be of much help because we would have to observe the agent's preferences in every possible combination of circumstances before being able to make any predictions about its behavior. Fortunately, the preferences of real agents are usually more systematic. Conversely, there are systematic ways of designing utility functions that, when installed in an artificial agent, cause it to generate the kinds of behavior we want.

The utility of money

Utility theory has its roots in economics, and economics provides one obvious candidate for a utility measure: money (or more specifically, an agent's total net assets). The almost universal exchangeability of money for all kinds of goods and services suggests that money plays a significant role in human utility functions. (In fact, most people think of economics as the study of money, when actually the root of the word *economy* refers to management, and its current emphasis is on the management of choice.)

If we restrict our attention to actions that only affect the amount of money that an agent has, then it will usually be the case that the agent prefers more money to less, all other things being equal. We say that the agent exhibits a **monotonic preference** for definite amounts of money. This is not, however, sufficient to guarantee that money behaves as a utility function, because it says nothing about preferences between *lotteries* involving money.

MONOTONIC PREFERENCE

Suppose you have triumphed over the other competitors in a television game show. The host now offers you a choice: either you can take the \$1,000,000 prize or you can gamble it on the flip of a coin. If the coin comes up heads, you end up with nothing, but if it comes up tails, you get \$3,000,000. If you're like most people, you would decline the gamble and pocket the million. Are you being irrational?

EXPECTED MONETARY VALUE

Assuming you believe that the coin is fair, the **expected monetary value** (EMV) of the gamble is $\frac{1}{2}(\$0) + \frac{1}{2}(\$3{,}000{,}000) = \$1{,}500{,}000$, and the EMV of taking the original prize is of course \$1,000,000, which is less. But that does not necessarily mean that accepting the gamble is a better decision. Suppose we use S_n to denote the state of possessing total

wealth $\$n$, and that your current wealth is $\$k$. Then the expected utilities of the two actions of accepting and declining the gamble are

$$\begin{aligned} EU(Accept) &= \tfrac{1}{2}U(S_k) + \tfrac{1}{2}U(S_{k+3,000,000}) \,, \\ EU(Decline) &= U(S_{k+1,000,000}) \,. \end{aligned}$$

To determine what to do, we need to assign utilities to the outcome states. Utility is not directly proportional to monetary value, because the utility—the positive change in lifestyle—for your first million is very high (or so we are told), whereas the utility for an additional million is much smaller. Suppose you assign a utility of 5 to your current financial status (S_k), a 10 to the state $S_{k+3,000,000}$, and an 8 to the state $S_{k+1,000,000}$. Then the rational action would be to decline, because the expected utility of accepting is only 7.5 (less than the 8 for declining). On the other hand, suppose that you happen to have \$500,000,000 in the bank already (and appear on game shows just for fun, one assumes). In this case, the gamble is probably acceptable, because the additional benefit of the 503rd million is probably about the same as that of the 501st million.

In a pioneering study of actual utility functions, Grayson (1960) found that the utility of money was almost exactly proportional to the *logarithm* of the amount. (This idea was first suggested by Bernoulli (1738); see Exercise 16.3.) One particular curve, for a certain Mr. Beard, is shown in Figure 16.2(a). The data obtained for Mr. Beard's preferences are consistent with a utility function

$$U(S_{k+n}) = -263.31 + 22.09 \log(n + 150,000)$$

for the range between $n = -\$150,000$ and $n = \$800,000$.

We should not assume that this is the definitive utility function for monetary value, but it is likely that most people have a utility function that is concave for positive wealth. Going into debt is usually considered disastrous, but preferences between different levels of debt can display a reversal of the concavity associated with positive wealth. For example,

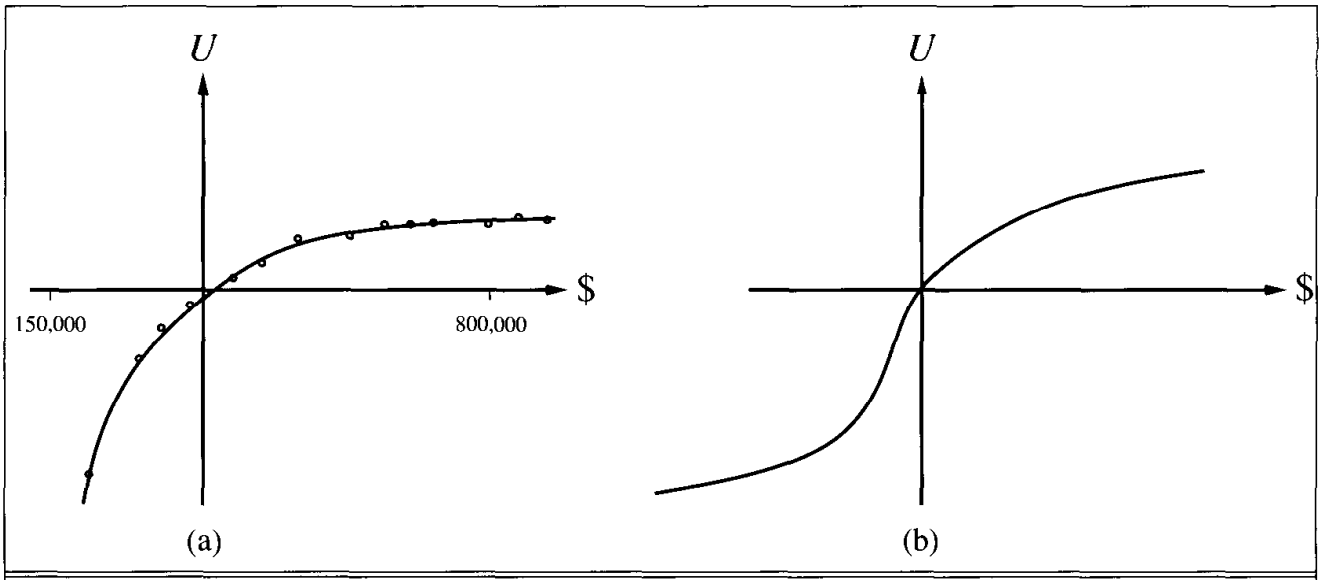


Figure 16.2 The utility of money. (a) Empirical data for Mr. Beard over a limited range. (b) A typical curve for the full range.

someone already \$10,000,000 in debt might well accept a gamble on a fair coin with a gain of \$10,000,000 for heads and a loss of \$20,000,000 for tails.[2] This yields the S-shaped curve shown in Figure 16.2(b).

If we restrict our attention to the positive part of the curves, where the slope is decreasing, then for any lottery L, the utility of being faced with that lottery is less than the utility of being handed the expected monetary value of the lottery as a sure thing:

$$U(L) < U(S_{EMV(L)}) \,.$$

RISK-AVERSE

RISK-SEEKING

CERTAINTY EQUIVALENT

INSURANCE PREMIUM

That is, agents with curves of this shape are **risk-averse**: they prefer a sure thing with a payoff that is less than the expected monetary value of a gamble. On the other hand, in the "desperate" region at large negative wealth in Figure 16.2(b), the behavior is **risk-seeking**. The value an agent will accept in lieu of a lottery is called the **certainty equivalent** of the lottery. Studies have shown that most people will accept about \$400 in lieu of a gamble that gives \$1000 half the time and \$0 the other half—that is, the certainty equivalent of the lottery is \$400. The difference between the expected monetary value of a lottery and its certainty equivalent is called the **insurance premium**. Risk aversion is the basis for the insurance industry, because it means that insurance premiums are positive. People would rather pay a small insurance premium than gamble the price of their house against the chance of a fire. From the insurance company's point of view, the price of the house is very small compared with the firm's total reserves. This means that the insurer's utility curve is approximately linear over such a small region, and the gamble costs the company almost nothing.

RISK-NEUTRAL

Notice that for *small* changes in wealth relative to the current wealth, almost any curve will be approximately linear. An agent that has a linear curve is said to be **risk-neutral**. For gambles with small sums, therefore, we expect risk neutrality. In a sense, this justifies the simplified procedure that proposed small gambles to assess probabilities and to justify the axioms of probability in Chapter 13.

Utility scales and utility assessment

The axioms of utility do not specify a unique utility function for an agent, given its preference behavior. For example, we can transform a utility function $U(S)$ into

$$U'(S) = k_1 + k_2 U(S) \,,$$

where k_1 is a constant and k_2 is any *positive* constant. Clearly, this linear transformation leaves the agent's behavior unchanged.

VALUE FUNCTION

ORDINAL UTILITY FUNCTION

In *deterministic* contexts, where there are states but no lotteries, behavior is unchanged by any *monotonic* transformation. For example, we can take the cube root of all the utilities without affecting the preference ordering on actions. An agent in a deterministic environment is said to have a **value function** or **ordinal utility function**; the function really provides just rankings of states rather than meaningful numerical values. We saw this distinction in Chapter 6 for games: evaluation functions in deterministic games such as chess are value functions, whereas evaluation functions in nondeterministic games like backgammon are true utility functions.

[2] Such behavior might be called desperate, but it is rational if one is already in a desperate situation.

HUMAN JUDGMENT AND FALLIBILITY

Decision theory is a **normative** theory: it describes how a rational agent *should* act. The application of economic theory would be greatly enhanced if it were also a **descriptive** theory of actual human decision making. However, there is experimental evidence indicating that people systematically violate the axioms of utility theory. An example is given by the psychologists Tversky and Kahneman (1982), based on an example by the economist Allais (1953). Subjects in this experiment are given a choice between lotteries A and B and then between C and D:

A : 80% chance of \$4000 C : 20% chance of \$4000
B : 100% chance of \$3000 D : 25% chance of \$3000 .

The majority of subjects choose B over A and C over D. But if we assign $U(\$0) = 0$, then the first of these choices implies that $0.8U(\$4000) < U(\$3000)$, whereas the second choice implies exactly the reverse. In other words, there seems to be no utility function that is consistent with these choices. One possible conclusion is that humans are simply irrational by the standards of our utility axioms. An alternative view is that the analysis does not take into account **regret**—the feeling that humans know they would experience if they gave up a certain reward (B) for an 80% chance at a higher reward and then lost. In other words, if A is chosen, there is a 20% chance of getting no money *and feeling like a complete idiot.*

Kahneman and Tversky go on to develop a descriptive theory that explains how people are risk-averse with high-probability events, but are willing to take more risks with unlikely payoffs. The connection between this finding and AI is that the choices our agents can make are only as good as the preferences they are based on. If our human informants insist on contradictory preference judgments, there is nothing our agent can do to be consistent with them.

Fortunately, preference judgments made by humans are often open to revision in the light of further consideration. In early work at Harvard Business School on assessing the utility of money, Keeney and Raiffa (1976, p. 210) found the following:

> A great deal of empirical investigation has shown that there is a serious deficiency in the assessment protocol. Subjects tend to be too risk-averse in the small and therefore ... the fitted utility functions exhibit unacceptably large risk premiums for lotteries with a large spread. ... Most of the subjects, however, can reconcile their inconsistencies and feel that they have learned an important lesson about how they want to behave. As a consequence, some subjects cancel their automobile collision insurance and take out more term insurance on their lives.

Even today, human (ir)rationality is the subject of intensive investigation.

NORMALIZED UTILITIES

STANDARD LOTTERY

One procedure for assessing utilities is to establish a scale with a "best possible prize" at $U(S) = u_\top$ and a "worst possible catastrophe" at $U(S) = u_\bot$. **Normalized utilities** use a scale with $u_\bot = 0$ and $u_\top = 1$. Utilities of intermediate outcomes are assessed by asking the agent to indicate a preference between the given outcome state S and a **standard lottery** $[p, u_\top;\ (1-p), u_\bot]$. The probability p is adjusted until the agent is indifferent between S and the standard lottery. Assuming normalized utilities, the utility of S is given by p.

In medical, transportation, and environmental decision problems, among others, people's lives are at stake. In such cases, $u_\bot$ is the value assigned to immediate death (or perhaps many deaths). *Although nobody feels comfortable with putting a value on human life, it is a fact that tradeoffs are made all the time.* Aircraft are given a complete overhaul at intervals determined by trips and miles flown, rather than after every trip. Car bodies are made with relatively thin sheet metal to reduce costs, despite the decrease in accident survival rates. Leaded fuel is still widely used even though it has known health hazards. Paradoxically, a refusal to "put a monetary value on life" means that life is often *undervalued.* Ross Shachter relates an experience with a government agency that commissioned a study on removing asbestos from schools. The study assumed a particular dollar value for the life of a school-age child, and argued that the rational choice under that assumption was to remove the asbestos. The government agency, morally outraged, rejected the report out of hand. It then decided against asbestos removal.

MICROMORT

QALY

Some attempts have been made to find out the value that people place on their own lives. Two common "currencies" used in medical and safety analysis are the **micromort** (a one in a million chance of death) and the **QALY**, or quality-adjusted life year (equivalent to a year in good health with no infirmities). A number of studies across a wide range of individuals have shown that a micromort is worth about $20 (1980 dollars). We have already seen that utility functions need not be linear, so this does not imply that a decision maker would kill himself for $20 million. Again, the local linearity of any utility curve means that micromort and QALY values are most appropriate for small incremental risks and rewards.

16.4 MULTIATTRIBUTE UTILITY FUNCTIONS

Decision making in the field of public policy involves both millions of dollars and life and death. For example, in deciding what levels of a carcinogenic substance to allow into the environment, policy makers must weigh the prevention of deaths against the economic hardship that might result from the elimination of certain products and processes. Siting a new airport requires consideration of the disruption caused by construction; the cost of land; the distance from centers of population; the noise of flight operations; safety issues arising from local topography and weather conditions; and so on. Problems like these, in which outcomes are characterized by two or more attributes, are handled by **multiattribute utility theory**.

MULTIATTRIBUTE UTILITY THEORY

We will call the attributes $\mathbf{X} = X_1, \ldots, X_n$; a complete vector of assignments will be $\mathbf{x} = \langle x_1, \ldots, x_n \rangle$. Each attribute is generally assumed to have discrete or continuous scalar values. For simplicity, we will assume that each attribute is defined in such a way that,

all other things being equal, higher values of the attribute correspond to higher utilities. For example, if we choose $AbsenceOfNoise$ as an attribute in the airport problem then the greater its value, the better the solution. In some cases, it may be necessary to subdivide the range of values so that utility varies monotonically within each range.

We begin by examining cases in which decisions can be made *without* combining the attribute values into a single utility value. Then we look at cases where the utilities of attribute combinations can be specified very concisely.

Dominance

STRICT DOMINANCE

Suppose that airport site S_1 costs less, generates less noise pollution, and is safer than site S_2. One would not hesitate to reject S_2. We then say that there is **strict dominance** of S_1 over S_2. In general, if an option is of lower value on all attributes than some other option, it need not be considered further. Strict dominance is often very useful in narrowing down the field of choices to the real contenders, although it seldom yields a unique choice. Figure 16.3(a) shows a schematic diagram for the two-attribute case.

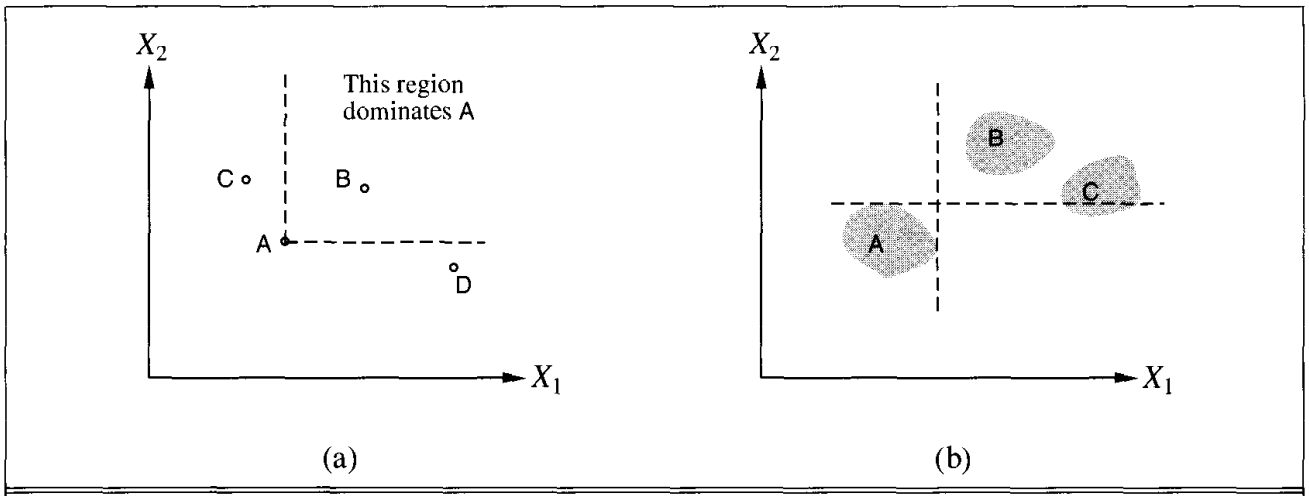


Figure 16.3 Strict dominance. (a) Deterministic: Option A is strictly dominated by B but not by C or D. (b) Uncertain: A is strictly dominated by B but not by C.

That is fine for the deterministic case, in which the attribute values are known for sure. What about the general case, where the action outcomes are uncertain? A direct analog of strict dominance can be constructed, where, despite the uncertainty, all possible concrete outcomes for S_1 strictly dominate all possible outcomes for S_2. (See Figure 16.3(b).) Of course, this will probably occur even less often than in the deterministic case.

STOCHASTIC DOMINANCE

Fortunately, there is a more useful generalization called **stochastic dominance**, which occurs very frequently in real problems. Stochastic dominance is easiest to understand in the context of a single attribute. Suppose we believe that the cost of siting the airport at S_1 is uniformly distributed between \$2.8 billion and \$4.8 billion and that the cost at S_2 is uniformly distributed between \$3 billion and \$5.2 billion. Figure 16.4(a) shows these distributions, with cost plotted as a negative value. Then, given only the information that utility decreases with cost, we can say that S_1 stochastically dominates S_2 (i.e., S_2 can be discarded). It is important

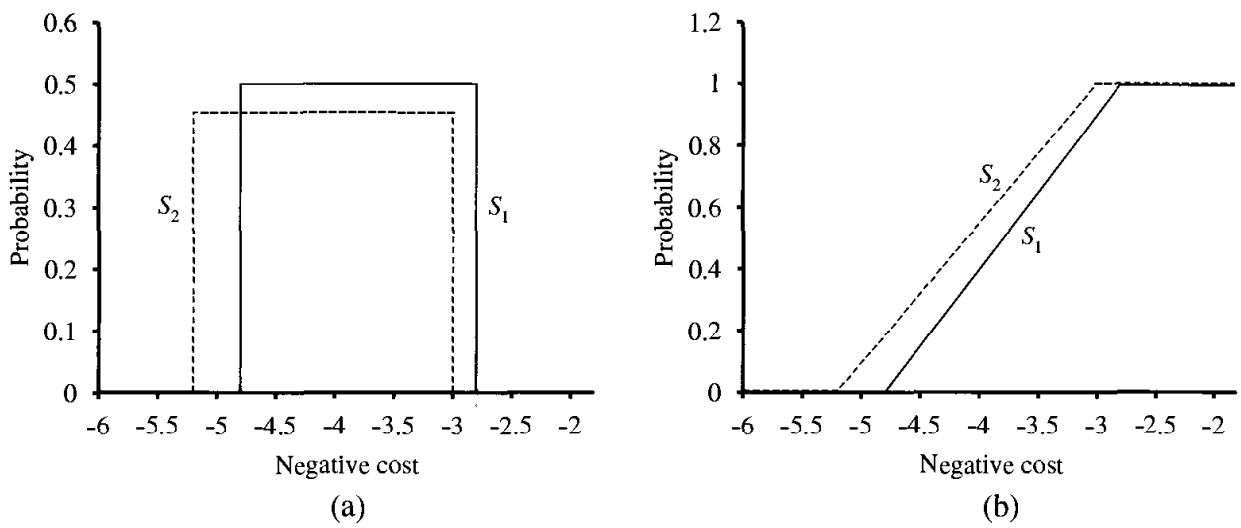


Figure 16.4 Stochastic dominance. (a) S_1 stochastically dominates S_2 on cost. (b) Cumulative distributions for the negative cost of S_1 and S_2.

to note that this does *not* follow from comparing the expected costs. For example, if we knew the cost of S_1 to be *exactly* \$3.8 billion, then we would be *unable* to make a decision without additional information on the utility of money.[3]

The exact relationship between the attribute distributions needed to establish stochastic dominance is best seen by examining the *cumulative* distributions, shown in Figure 16.4(b). The cumulative distribution measures the probability that the cost is less than or equal to any given amount—that is, it integrates the original distribution. If the cumulative distribution for S_1 is always to the right of the cumulative distribution for S_2, then, stochastically speaking, S_1 is cheaper than S_2. Formally, if two actions A_1 and A_2 lead to probability distributions $p_1(x)$ and $p_2(x)$ on attribute X, then A_1 stochastically dominates A_2 on X if

$$\forall x \int_{-\infty}^{x} p_1(x')\, dx' \leq \int_{-\infty}^{x} p_2(x')\, dx' .$$

The relevance of this definition to the selection of optimal decisions comes from the following property: *if A_1 stochastically dominates A_2, then for any monotonically nondecreasing utility function $U(x)$, the expected utility of A_1 is at least as high as the expected utility of A_2.* Hence, if an action is stochastically dominated by another action on all attributes, then it can be discarded.

The stochastic dominance condition might seem rather technical and perhaps not so easy to evaluate without extensive probability calculations. In fact, it can be decided very easily in many cases. Suppose, for example, that the construction cost depends on the distance to centers of population. The cost itself is uncertain, but the greater the distance, the greater the cost. If S_1 is less remote than S_2, then S_1 will dominate S_2 on cost. Although we will not

[3] It might seem odd that *more* information on the cost of S_1 could make the agent *less* able to decide. The paradox is resolved by noting that the decision reached in the absence of exact cost information is less likely to have the highest utility payoff.

QUALITATIVE PROBABILISTIC NETWORKS

present them here, there exist algorithms for propagating this kind of qualitative information among uncertain variables in **qualitative probabilistic networks**, enabling a system to make rational decisions based on stochastic dominance, without using any numeric values.

Preference structure and multiattribute utility

Suppose we have n attributes, each of which has d distinct possible values. To specify the complete utility function $U(x_1, \ldots, x_n)$, we need d^n values in the worst case. Now, the worst case corresponds to a situation in which the agent's preferences have no regularity at all. Multiattribute utility theory is based on the supposition that the preferences of typical agents have much more structure than that. The basic approach is to identify regularities in the preference behavior we would expect to see and to use what are called **representation theorems** to show that an agent with a certain kind of preference structure has a utility function

REPRESENTATION THEOREMS

$$U(x_1, \ldots, x_n) = f[f_1(x_1), \ldots, f_n(x_n)\ ,]$$

where f is, we hope, a simple function such as addition. Notice the similarity to the use of Bayesian networks to decompose the joint probability of several random variables.

Preferences without uncertainty

Let us begin with the deterministic case. Remember that for deterministic environments the agent has a value function $V(x_1, \ldots, x_n)$; the aim is to represent this function concisely. The basic regularity that arises in deterministic preference structures is called **preference independence**. Two attributes X_1 and X_2 are preferentially independent of a third attribute X_3 if the preference between outcomes $\langle x_1, x_2, x_3 \rangle$ and $\langle x'_1, x'_2, x_3 \rangle$ does not depend on the particular value x_3 for attribute X_3.

PREFERENCE INDEPENDENCE

Going back to the airport example, where we have (among other attributes) *Noise*, *Cost*, and *Deaths* to consider, one may propose that *Noise* and *Cost* are preferentially independent of *Deaths*. For example, if we prefer a state with 20,000 people residing in the flight path and a construction cost of \$4 billion to a state with 70,000 people residing in the flight path and a cost of \$3.7 billion when the safety level is 0.06 deaths per million passenger miles in both cases, then we would have the same preference when the safety level is 0.13 or 0.01; and the same independence would hold for preferences between any other pair of values for *Noise* and *Cost*. It is also apparent that *Cost* and *Deaths* are preferentially independent of *Noise* and that *Noise* and *Deaths* are preferentially independent of *Cost*. We say that the set of attributes $\{Noise, Cost, Deaths\}$ exhibits **mutual preferential independence** (MPI). MPI says that, whereas each attribute may be important, it does not affect the way in which one trades off the other attributes against each other.

MUTUAL PREFERENTIAL INDEPENDENCE

Mutual preferential independence is something of a mouthful, but thanks to a remarkable theorem due to the economist Debreu (1960), we can derive from it a very simple form for the agent's value function: *If attributes $X_1, \ldots, X_n$ are mutually preferentially independent, then the agent's preference behavior can be described as maximizing the function*

$$V(x_1, \ldots, x_n) = \sum_i V_i(x_i)\ ,$$

where each V_i is a value function referring only to the attribute X_i. For example, it might well be the case that the airport decision can be made using a value function

$$V(noise, cost, deaths) = -noise \times 10^4 - cost - deaths \times 10^{12} .$$

ADDITIVE VALUE FUNCTION

A value function of this type is called an **additive value function**. Additive functions are an extremely natural way to describe an agent's value function and are valid in many real-world situations. Even when MPI does not strictly hold, as might be the case at extreme values of the attributes, an additive value function might still provide a good approximation to the agent's preferences. This is especially true when the violations of MPI occur in portions of the attribute ranges that are unlikely to occur in practice.

Preferences with uncertainty

When uncertainty is present in the domain, we will also need to consider the structure of preferences between lotteries and to understand the resulting properties of utility functions, rather than just value functions. The mathematics of this problem can become quite complicated, so we will present just one of the main results to give a flavor of what can be done. The reader is referred to Keeney and Raiffa (1976) for a thorough survey of the field.

UTILITY INDEPENDENCE

The basic notion of **utility independence** extends preference independence to cover lotteries: a set of attributes **X** is utility-independent of a set of attributes **Y** if preferences between lotteries on the attributes in **X** are independent of the particular values of the attributes in **Y**. A set of attributes is **mutually utility-independent** (MUI) if each of its subsets is utility-independent of the remaining attributes. Again, it seems reasonable to propose that the airport attributes are MUI.

MUTUALLY UTILITY-INDEPENDENT

MULTIPLICATIVE UTILITY FUNCTION

MUI implies that the agent's behavior can be described using a **multiplicative utility function** (Keeney, 1974). The general form of a multiplicative utility function is best seen by looking at the case for three attributes. For conciseness, we will use U_i to mean $U_i(x_i)$:

$$U = k_1U_1 + k_2U_2 + k_3U_3 + k_1k_2U_1U_2 + k_2k_3U_2U_3 + k_3k_1U_3U_1 + k_1k_2k_3U_1U_2U_3 .$$

Although this does not look very simple, it contains just three single-attribute utility functions and three constants. In general, an n-attribute problem exhibiting MUI can be modeled using n single-attribute utilities and n constants. Each of the single-attribute utility functions can be developed independently of the other attributes, and this combination will be guaranteed to generate the correct overall preferences. Additional assumptions are required to obtain a purely additive utility function.

16.5 Decision Networks

INFLUENCE DIAGRAM

DECISION NETWORK

In this section, we will look at a general mechanism for making rational decisions. The notation is often called an **influence diagram** (Howard and Matheson, 1984), but we will use the more descriptive term **decision network**. Decision networks combine Bayesian networks with additional node types for actions and utilities. We will use airport siting as an example.

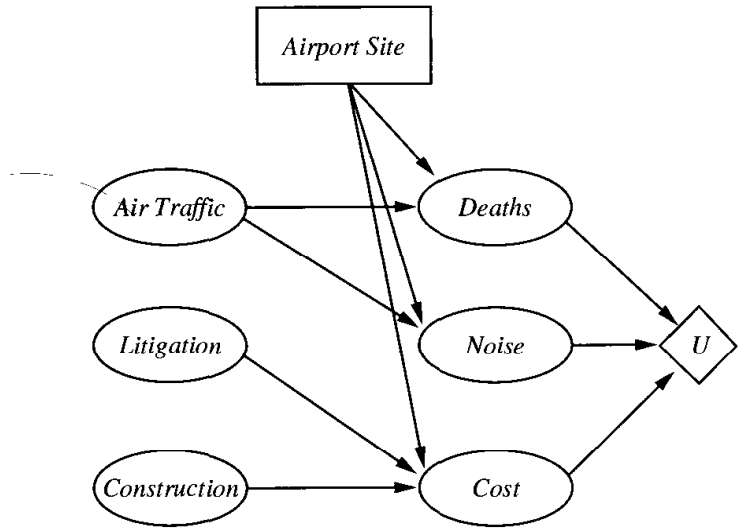


Figure 16.5 A simple decision network for the airport-siting problem.

Representing a decision problem with a decision network

In its most general form, a decision network represents information about the agent's current state, its possible actions, the state that will result from the agent's action, and the utility of that state. It therefore provides a substrate for implementing utility-based agents of the type first introduced in Section 2.4. Figure 16.5 shows a decision network for the airport siting problem. It illustrates the three types of nodes used:

CHANCE NODES

◇ **Chance nodes** (ovals) represent random variables, just as they do in Bayes nets. The agent could be uncertain about the construction cost, the level of air traffic and the potential for litigation, and the *Deaths*, *Noise*, and total *Cost* variables, each of which also depends on the site chosen. Each chance node has associated with it a conditional distribution that is indexed by the state of the parent nodes. In decision networks, the parent nodes can include decision nodes as well as chance nodes. Note that each of the current-state chance nodes could be part of a large Bayes net for assessing construction costs, air traffic levels, or litigation potentials.

DECISION NODES

◇ **Decision nodes** (rectangles) represent points where the decision-maker has a choice of actions. In this case, the *AirportSite* action can take on a different value for each site under consideration. The choice influences the cost, safety, and noise that will result. In this chapter, we will assume that we are dealing with a single decision node. Chapter 17 deals with cases where more than one decision must be made.

UTILITY NODES

◇ **Utility nodes** (diamonds) represent the agent's utility function.[4] The utility node has as parents all variables describing the outcome that directly affect utility. Associated with the utility node is a description of the agent's utility as a function of the parent attributes. The description could be just a tabulation of the function, or it might be a parameterized additive or multilinear function.

[4] These nodes are often called **value nodes** in the literature. We prefer to maintain the distinction between utility and value functions, as discussed earlier, because the outcome state may represent a lottery.

A simplified form is also used in many cases. The notation remains identical, but the chance nodes describing the outcome state are omitted. Instead, the utility node is connected directly to the current-state nodes and the decision node. In this case, rather than representing a utility function on states, the utility node represents the *expected* utility associated with each action, as defined in Equation (16.1). We therefore call such tables **action-utility tables**. Figure 16.6 shows the action-utility representation of the airport problem.

ACTION-UTILITY TABLES

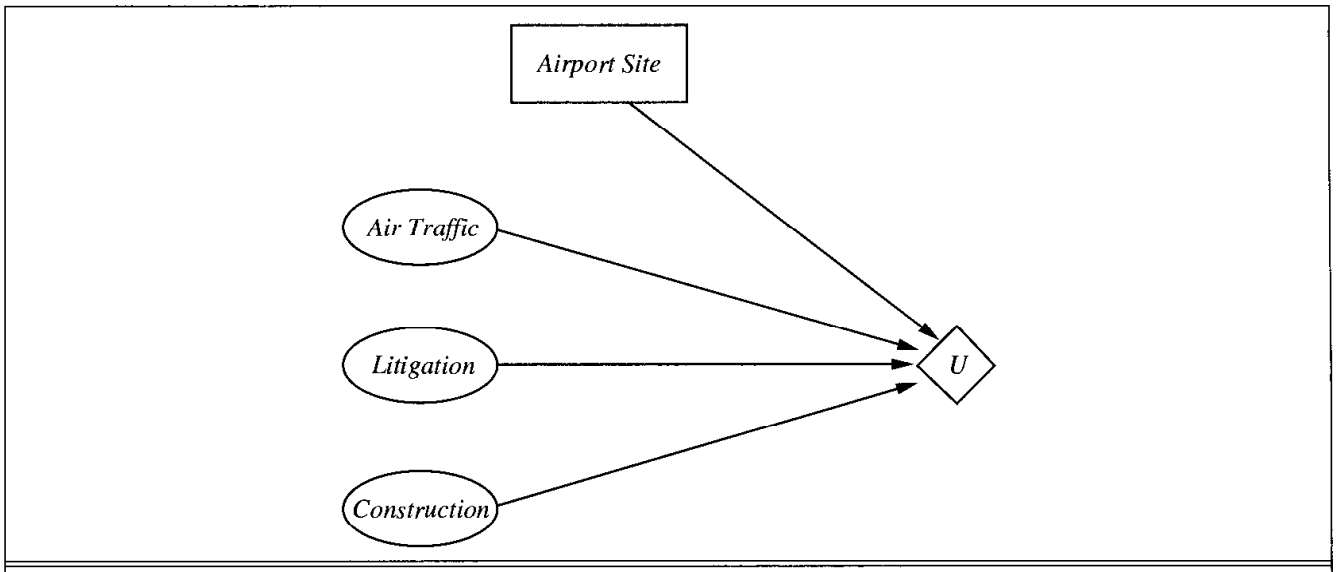


Figure 16.6 A simplified representation of the airport-siting problem. Chance nodes corresponding to outcome states have been factored out.

Notice that, because the *Noise*, *Deaths*, and *Cost* chance nodes in Figure 16.5 refer to future states, they can never have their values set as evidence variables. Thus, the simplified version that omits these nodes can be used whenever the more general form can be used. Although the simplified form contains fewer nodes, the omission of an explicit description of the outcome of the siting decision means that it is less flexible with respect to changes in circumstances. For example, in Figure 16.5, a change in aircraft noise levels can be reflected by a change in the conditional probability table associated with the *Noise* node, whereas a change in the weight accorded to noise pollution in the utility function can be reflected by a change in the utility table. In the action-utility diagram, Figure 16.6, on the other hand, all such changes have to be reflected by changes to the action-utility table. Essentially, the action-utility formulation is a *compiled* version of the original formulation.

Evaluating decision networks

Actions are selected by evaluating the decision network for each possible setting of the decision node. Once the decision node is set, it behaves exactly like a chance node that has been set as an evidence variable. The algorithm for evaluating decision networks is the following:

1. Set the evidence variables for the current state.
2. For each possible value of the decision node;
   (a) Set the decision node to that value.

(b) Calculate the posterior probabilities for the parent nodes of the utility node, using a standard probabilistic inference algorithm.

(c) Calculate the resulting utility for the action.

3. Return the action with the highest utility.

This is a straightforward extension of the Bayes net algorithm and can be incorporated directly into the agent design given in Figure 13.1. We will see in Chapter 17 that the possibility of executing several actions in sequence makes the problem much more interesting.

16.6 THE VALUE OF INFORMATION

In the preceding analysis, we have assumed that all relevant information, or at least all available information, is provided to the agent before it makes its decision. In practice, this is hardly ever the case. *One of the most important parts of decision making is knowing what questions to ask.* For example, a doctor cannot expect to be provided with the results of *all possible* diagnostic tests and questions at the time a patient first enters the consulting room.[5] Tests are often expensive and sometimes hazardous (both directly and because of associated delays). Their importance depends on two factors: whether the test results would lead to a significantly better treatment plan, and how likely the various test results are.

INFORMATION VALUE THEORY

This section describes **information value theory**, which enables an agent to choose what information to acquire. The acquisition of information is achieved by **sensing actions**, as described in Chapter 12. Because the agent's utility function seldom refers to the contents of the agent's internal state, whereas the whole purpose of sensing actions is to affect the internal state, we must evaluate sensing actions by their effect on the agent's subsequent "real" actions. Thus, information value theory involves a form of sequential decision making.

A simple example

Suppose an oil company is hoping to buy one of n indistinguishable blocks of ocean drilling rights. Let us assume further that exactly one of the blocks contains oil worth C dollars and that the price of each block is C/n dollars. If the company is risk-neutral, then it will be indifferent between buying a block and not buying one.

Now suppose that a seismologist offers the company the results of a survey of block number 3, which indicates definitively whether the block contains oil. How much should the company be willing to pay for the information? The way to answer this question is to examine what the company would do if it had the information:

- With probability $1/n$, the survey will indicate oil in block 3. In this case, the company will buy block 3 for C/n dollars and make a profit of $C - C/n = (n-1)C/n$ dollars.
- With probability $(n-1)/n$, the survey will show that the block contains no oil, in which case the company will buy a different block. Now the probability of finding oil in one

[5] In the United States, the only question that is always asked beforehand is whether the patient has insurance.

of the other blocks changes from $1/n$ to $1/(n-1)$, so the company makes an expected profit of $C/(n-1) - C/n = C/n(n-1)$ dollars.

Now we can calculate the expected profit, given the survey information:

$$\frac{1}{n} \times \frac{(n-1)C}{n} + \frac{n-1}{n} \times \frac{C}{n(n-1)} = C/n \ .$$

Therefore, the company should be willing to pay the seismologist up to C/n dollars for the information: the information is worth as much as the block itself.

The value of information derives from the fact that *with* the information, one's course of action can be changed to suit the *actual* situation. One can discriminate according to the situation, whereas without the information, one has to do what's best on average over the possible situations. In general, the value of a given piece of information is defined to be the difference in expected value between best actions before and after information is obtained.

A general formula

It is simple to derive a general mathematical formula for the value of information. Usually, we assume that exact evidence is obtained about the value of some random variable E_j, so the phrase **value of perfect information** (VPI) is used.[6] Let the agent's current knowledge be E. Then the value of the current best action α is defined by

VALUE OF PERFECT INFORMATION

$$EU(\alpha|E) = \max_A \sum_i U(Result_i(A))\ P(Result_i(A)|Do(A), E)$$

and the value of the new best action (after the new evidence E_j is obtained) will be

$$EU(\alpha_{E_j}|E, E_j) = \max_A \sum_i U(Result_i(A))\ P(Result_i(A)|Do(A), E, E_j) \ .$$

But E_j is a random variable whose value is *currently* unknown, so we must average over all possible values e_{jk} that we might discover for E_j, using our *current* beliefs about its value. The value of discovering E_j, given current information E, is then defined as

$$VPI_E(E_j) = \left(\sum_k P(E_j = e_{jk}|E) EU(\alpha_{e_{jk}}|E, E_j = e_{jk}) \right) - EU(\alpha|E) \ .$$

In order to get some intuition for this formula, consider the simple case where there are only two actions, A_1 and A_2, from which to choose. Their current expected utilities are U_1 and U_2. The information E_j will yield some new expected utilities U_1' and U_2' for the actions, but before we obtain E_j, we will have some probability distributions over the possible values of U_1' and U_2' (which we will assume are independent).

Suppose that A_1 and A_2 represent two different routes through a mountain range in winter. A_1 is a nice, straight highway through a low pass, and A_2 is a winding dirt road over the top. Just given this information, A_1 is clearly preferable, because it is quite likely that the second route is blocked by avalanches, whereas it is quite unlikely that the first route is blocked by traffic. U_1 is therefore clearly higher than U_2. It is possible to obtain satellite

[6] Imperfect information about a variable X can be modeled as perfect information about a variable Y that is probabilistically related to X. See Exercise 16.11 for an example of this.

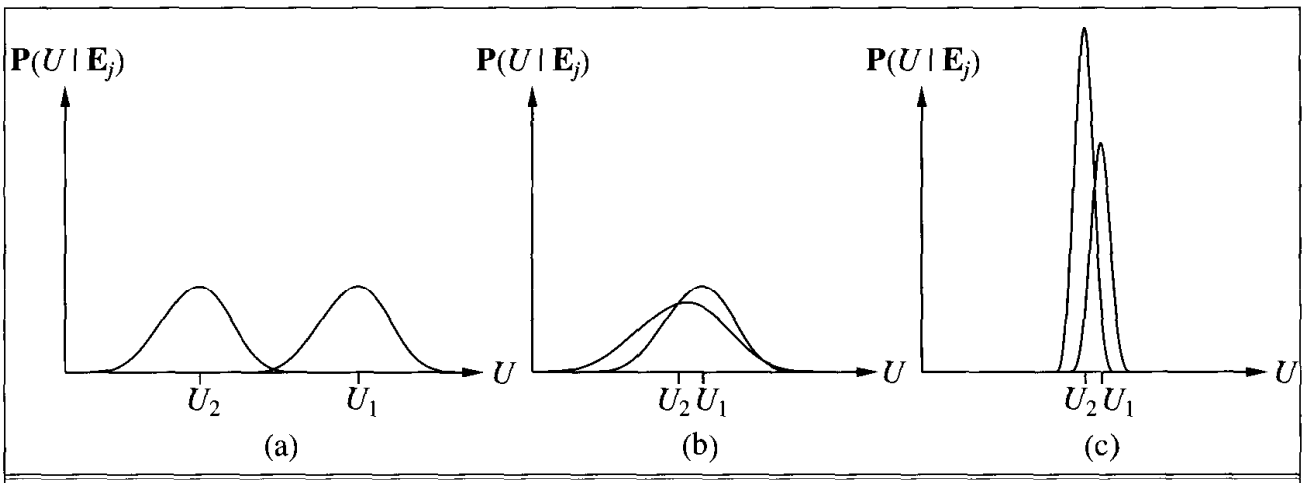


Figure 16.7 Three generic cases for the value of information. In (a), A_1 will almost certainly remain superior to A_2, so the information is not needed. In (b), the choice is unclear and the information is crucial. In (c), the choice is unclear but because it makes little difference, the information is less valuable.

reports E_j on the actual state of each road that would give new expectations, U_1' and U_2', for the two crossings. The distributions for these expectations are shown in Figure 16.7(a). Obviously, in this case, it is not worth the expense of obtaining satellite reports, because it is unlikely that the information derived from them will change the plan. With no change, information has no value.

Now suppose that we are choosing between two different winding dirt roads of slightly different lengths and we are carrying a seriously injured passenger. Then, even when U_1 and U_2 are quite close, the distributions of U_1' and U_2' are very broad. There is a significant possibility that the second route will turn out to be clear while the first is blocked, and in this case the difference in utilities will be very high. The VPI formula indicates that it might be worthwhile getting the satellite reports. Such a situation is shown in Figure 16.7(b).

Now suppose that we are choosing between the two dirt roads in summertime, when blockage by avalanches is unlikely. In this case, satellite reports might show one route to be more scenic than the other because of flowering alpine meadows, or perhaps wetter because of errant streams. It is therefore quite likely that we would change our plan if we had the information. But in this case, the difference in value between the two routes is still likely to be very small, so we will not bother to obtain the reports. This situation is shown in Figure 16.7(c).

In sum, *information has value to the extent that it is likely to cause a change of plan and to the extent that the new plan will be significantly better than the old plan.*

Properties of the value of information

One might ask whether it is possible for information to be deleterious: can it actually have negative expected value? Intuitively, one should expect this to be impossible. After all, one could in the worst case just ignore the information and pretend that one has never received it. This is confirmed by the following theorem, which applies to any decision-theoretic agent:

The value of information is nonnegative:

$$\forall j, E \quad VPI_E(E_j) \geq 0 \,.$$

The theorem follows directly from the definition of VPI, and we leave the proof as an exercise (Exercise 16.12). It is important to remember that VPI depends on the current state of information, which is why it is subscripted. It can change as more information is acquired. In the extreme case, it will become zero if the variable in question already has a known value. Thus, VPI is not additive. That is,

$$VPI_E(E_j, E_k) \neq VPI_E(E_j) + VPI_E(E_k) \qquad \text{(in general)} \,.$$

VPI is, however, order-independent, which should be intuitively obvious. That is,

$$VPI_E(E_j, E_k) = VPI_E(E_j) + VPI_{E,E_j}(E_k) = VPI_E(E_k) + VPI_{E,E_k}(E_j) \,.$$

Order independence distinguishes sensing actions from ordinary actions and simplifies the problem of calculating the value of a sequence of sensing actions.

Implementing an information-gathering agent

A sensible agent should ask questions of the user in a reasonable order, should avoid asking questions that are irrelevant, should take into account the importance of each piece of information in relation to its cost, and should stop asking questions when that is appropriate. All of these capabilities can be achieved by using the value of information as a guide.

Figure 16.8 shows the overall design of an agent that can gather information intelligently before acting. For now, we will assume that with each observable evidence variable E_j, there is an associated cost, $Cost(E_j)$, which reflects the cost of obtaining the evidence through tests, consultants, questions, or whatever. The agent requests what appears to be the most valuable piece of information, compared with its cost. We assume that the result of the action $Request(E_j)$ is that the next percept provides the value of E_j. If no observation is worth its cost, the agent selects a "real" action.

The agent algorithm we have described implements a form of information gathering that is called **myopic**. This is because it uses the VPI formula shortsightedly, calculating

MYOPIC

```
function INFORMATION-GATHERING-AGENT(percept) returns an action
  static: D, a decision network

  integrate percept into D
  j ← the value that maximizes VPI(E_j) − Cost(E_j)
  if VPI(E_j) > Cost(E_j)
     then return REQUEST(E_j)
  else return the best action from D
```

Figure 16.8 Design of a simple information-gathering agent. The agent works by repeatedly selecting the observation with the highest information value, until the cost of the next observation is greater than its expected benefit.

the value of information as if only a single evidence variable will be acquired. If there is no single evidence variable that will help a lot, a myopic agent might hastily take an action when it would have been better to request two or more variables first and then take action. Myopic control is based on the same heuristic idea as greedy search and often works well in practice. (For example, it has been shown to outperform expert physicians in selecting diagnostic tests.) However, a perfectly rational information-gathering agent should consider all possible sequences of information requests terminating in an external action and all possible outcomes of those requests. Because the value of the second request depends on the outcome of the first request, the agent needs to explore the space of **conditional plans**, as described in Chapter 12.

16.7 Decision-Theoretic Expert Systems

DECISION ANALYSIS

The field of **decision analysis**, which evolved in the 1950s and 1960s, studies the application of decision theory to actual decision problems. It is used to help make rational decisions in important domains where the stakes are high, such as business, government, law, military strategy, medical diagnosis and public health, engineering design, and resource management. The process involves a careful study of the possible actions and outcomes, as well as the preferences placed on each outcome. It is traditional in decision analysis to talk about two roles: the **decision maker** states preferences between outcomes, and the **decision analyst** enumerates the possible actions and outcomes and elicits preferences from the decision maker to determine the best course of action. Until the early 1980s, the main purpose of decision analysis was to help humans make decisions that actually reflect their own preferences. In the current day, more and more decision processes are automated, and decision analysis is used to make sure that the automated processes are behaving as desired.

DECISION MAKER

DECISION ANALYST

As we discussed in Chapter 14, early expert system research concentrated on answering questions, rather than on making decisions. Those systems that did recommend actions rather than providing opinions on matters of fact generally did so using condition-action rules, rather than with explicit representations of outcomes and preferences. The emergence of Bayesian networks in the late 1980s made it possible to build large-scale systems that generated sound probabilistic inferences from evidence. The addition of decision networks means that expert systems can be developed that recommend optimal decisions, reflecting the preferences of the user as well as the available evidence.

A system that incorporates utilities can avoid one of the most common pitfalls associated with the consultation process: confusing likelihood and importance. A common strategy in early medical expert systems, for example, was to rank possible diagnoses in order of likelihood and report the most likely. Unfortunately, this can be disastrous! For the majority of patients in general practice, the two most *likely* diagnoses are usually "There's nothing wrong with you" and "You have a bad cold," but if the third-most-likely diagnosis for a given patient is lung cancer, that's a serious matter. Obviously, a testing or treatment plan should depend both on probabilities and utilities.

We will now describe the knowledge engineering process for decision-theoretic expert systems. As an example we will consider the problem of selecting a medical treatment for a kind of congenital heart disease in children (see Lucas, 1996).

AORTIC COARCTATION

About 0.8% of children are born with a heart anomaly, the most common being **aortic coarctation** (a constriction of the aorta). It can be treated with surgery, angioplasty (expanding the aorta with a balloon placed inside the artery) or medication. The problem is to decide what treatment to use and when to do it: the younger the infant the greater the risks of certain treatments, but one mustn't wait too long. A decision-theoretic expert system for this problem can be created by a team consisting of at least one domain expert (a pediatric cardiologist) and one knowledge engineer. The process can be broken down into the following steps (which you can compare to the steps in developing a logic-based system in Section 8.4).

Create a causal model. Determine what are the possible symptoms, disorders, treatments, and outcomes. Then draw arcs between them, indicating what disorders cause what symptoms, and what treatments alleviate what disorders. Some of this will be well known to the domain expert, and some will come from the literature. Often the model will match well with the informal graphical descriptions given in medical textbooks.

Simplify to a qualitative decision model. Since we are using the model to make treatment decisions and not for other purposes (such as determining the joint probability of certain symptom/disorder combinations), we can often simplify by removing variables that are not involved in treatment decisions. Sometimes variables will have to be split or joined to match the expert's intuitions. For example, the original aortic coarctation model had a *Treatment* variable with values *surgery*, *angioplasty* and *medication*, and a separate variable for *Timing* of the treatment. But the expert had a hard time thinking of these separately, so they were combined, with *Treatment* taking on values such as *surgery in 1 month*. This gives us the model of Figure 16.9.

Assign probabilities. Probabilities can come from patient databases, literature studies, or the expert's subjective assessments. In cases where the wrong kinds of probabilities are given in the literature, techniques such as Bayes' rule and marginalization can be used to compute the desired probabilities. It has been found that experts are best able to assess the probability of an effect given a cause (e.g. $P(dyspnoea|heartfailure)$) rather than the other way around.

Assign utilities. When there are a small number of possible outcomes, they can be enumerated and evaluated individually. We would create a scale from best to worst outcome and give each a numeric value, for example -1000 for death and 0 for complete recovery. We would then place the other outcomes on this scale. This can be done by the expert, but it is better if the patient (or in the case of infants, the patient's parents) can be involved, because different people have different preferences. If there are exponentially many outcomes, we need some way to combine them using multiattribute utility functions. For example, we may say that the negative utility of various complications is additive.

GOLD STANDARD

Verify and refine the model. To evaluate the system we will need a set of correct (input, output) pairs; a so-called **gold standard** to compare against. For medical expert systems this usually means assembling the best available doctors, presenting them with a few cases, and asking them for their diagnosis and recommended treatment plan. We then see

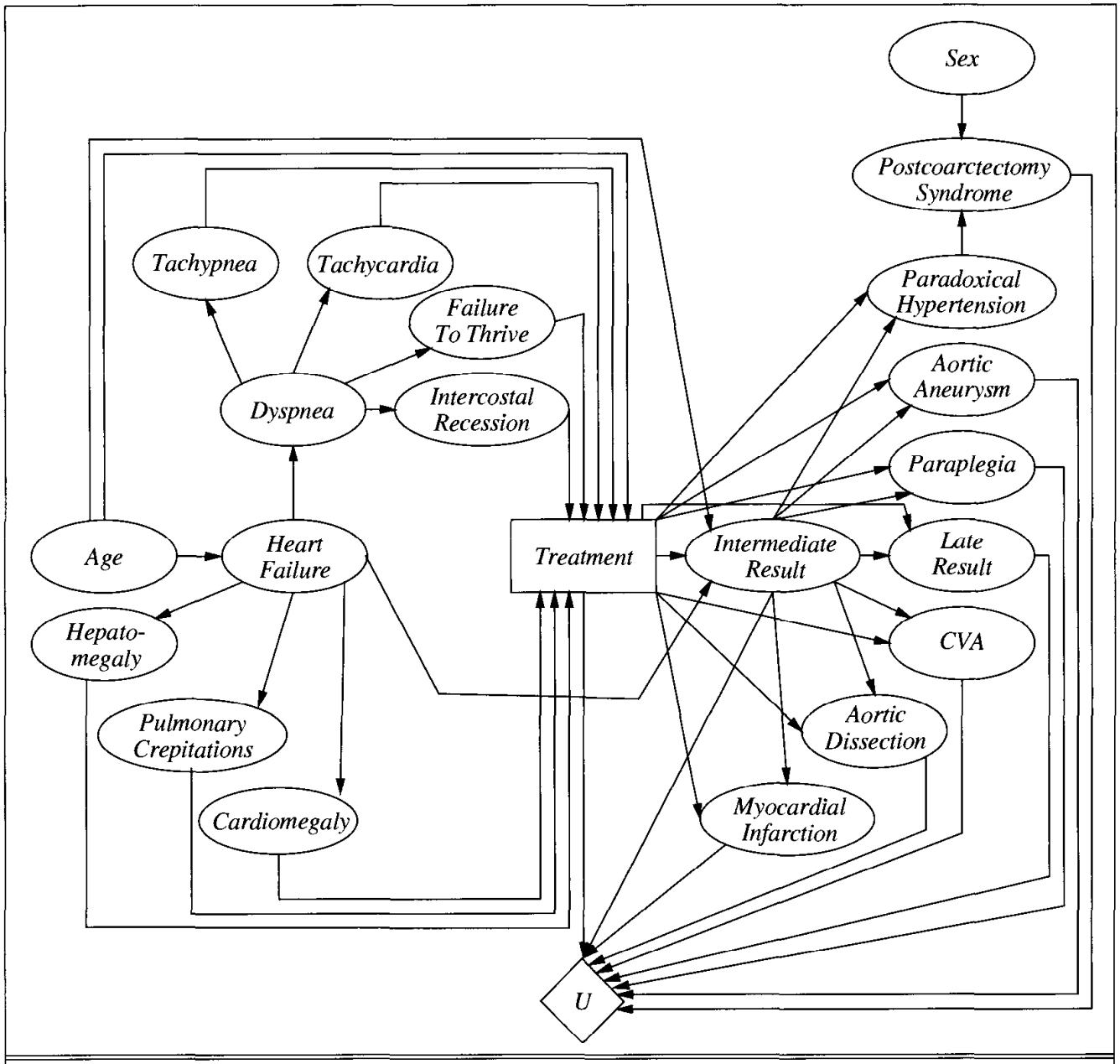


Figure 16.9 Influence diagram for aortic coarctation (courtesy of Peter Lucas).

how well the system matches their recommendations. If it does poorly, we try to isolate the parts that are going wrong and fix them. It can be useful to run the system "backwards." Instead of presenting the system with symptoms and asking for a diagnosis, we can present it with a diagnosis such as "heart failure," examine the predicted probability of symptoms such as tachycardia, and compare to the medical literature.

Perform sensitivity analysis. This important step checks whether the best decision is sensitive to small changes in the assigned probabilities and utilities by systematically varying those parameters and running the evaluation again. If small changes lead to significantly different decisions, then it could be worthwhile to spend more resources to collect better data. If all variations lead to the same decision, then the user will have more confidence that it is the right decision. Sensitivity analysis is particularly important, because one of the main criticisms of probabilistic approaches to expert systems is that it is too difficult to assess the

numerical probabilities required. Sensitivity analysis often reveals that many of the numbers need be specified only very approximately. For example, we might be uncertain about the prior probability $P(tachycardia)$, but if we try many different values for this probability and in each case the recommended action of the influence diagram is the same then we can be less concerned about our ignorance.

16.8 SUMMARY

This chapter shows how to combine utility theory with probability to enable an agent to select actions that will maximize its expected performance.

- **Probability theory** describes what an agent should believe on the basis of evidence, **utility theory** describes what an agent wants, and **decision theory** puts the two together to describe what an agent should do.
- We can use decision theory to build a system that makes decisions by considering all possible actions and choosing the one that leads to the best expected outcome. Such a system is known as a **rational agent**.
- Utility theory shows that an agent whose preferences between lotteries are consistent with a set of simple axioms can be described as possessing a utility function; furthermore, the agent selects actions as if maximizing its expected utility.
- **Multiattribute utility theory** deals with utilities that depend on several distinct attributes of states. **Stochastic dominance** is a particularly useful technique for making unambiguous decisions, even without precise utility values for attributes.
- **Decision networks** provide a simple formalism for expressing and solving decision problems. They are a natural extension of Bayesian networks, containing decision and utility nodes in addition to chance nodes.
- Sometimes, solving a problem involves finding more information before making a decision. The **value of information** is defined as the expected improvement in utility compared with making a decision without the information.
- **Expert systems** that incorporate utility information have additional capabilities compared with pure inference systems. In addition to being able to make decisions, they can use the value of information to decide whether to acquire it and they can calculate the sensitivity of their decisions to small changes in probability and utility assessments.

BIBLIOGRAPHICAL AND HISTORICAL NOTES

One of the earliest applications of the principle of maximum expected utility (although a deviant one involving infinite utilities) was Pascal's wager, first published as part of the *Port-Royal Logic* (Arnauld, 1662). Daniel Bernoulli (1738), investigating the St. Petersburg paradox (see Exercise 16.3), was the first to realize the importance of preference measurement

for lotteries, writing "the *value* of an item must not be based on its *price*, but rather on the *utility* that it yields" (italics his). Jeremy Bentham (1823) proposed the **hedonic calculus** for weighing "pleasures" and "pains," arguing that all decisions (not just monetary ones) could be reduced to utility comparisons.

The derivation of numerical utilities from preferences was first carried out by Ramsey (1931); the axioms for preference in the present text are closer in form to those rediscovered in *Theory of Games and Economic Behavior* (von Neumann and Morgenstern, 1944). A good presentation of these axioms, in the course of a discussion on risk preference, is given by Howard (1977). Ramsey had derived subjective probabilities (not just utilities) from an agent's preferences; Savage (1954) and Jeffrey (1983) carry out more recent constructions of this kind. Von Winterfeldt and Edwards (1986) provide a modern perspective on decision analysis and its relationship to human preference structures. The micromort utility measure is discussed by Howard (1989). A 1994 survey by the *Economist* set the value of a life at between $750,000 and $2.6 million. However, Richard Thaler (1992) found irrational variation in the price one is willing to pay to avoid a risk of death versus the price one is willing to be paid to accept a risk. For a 1/1000 chance, a respondent wouldn't pay more than $200 to remove the risk, but wouldn't accept $50,000 to take on the risk.

QALYs are much more widely used in medical and social policy decision making than are micromorts; see (Russell, 1990) for a typical example of an argument for a major change in public health policy on grounds of increased expected utility measured in QALYs.

The book *Decisions with Multiple Objectives: Preferences and Value Tradeoffs* (Keeney and Raiffa, 1976) gives a thorough introduction to multiattribute utility theory. It describes early computer implementations of methods for eliciting the necessary parameters for a multiattribute utility function and includes extensive accounts of real applications of the theory. In AI, the principal reference for MAUT is Wellman's (1985) paper, which includes a system called URP (Utility Reasoning Package) that can use a collection of statements about preference independence and conditional independence to analyze the structure of decision problems. The use of stochastic dominance together with qualitative probability models was investigated extensively by Wellman (1988, 1990a). Wellman and Doyle (1992) provide a preliminary sketch of how a complex set of utility-independence relationships might be used to provide a structured model of a utility function, in much the same way that Bayesian networks provide a structured model of joint probability distributions. Bacchus and Grove (1995, 1996) and La Mura and Shoham (1999) give further results along these lines.

Decision theory has been a standard tool in economics, finance, and management science since the 1950s. Until the 1980s, decision trees were the main tool used for representing simple decision problems. Smith (1988) gives an overview of the methodology of decision analysis. Decision networks or influence diagrams were introduced by Howard and Matheson (1984), based on earlier work by a group (including Howard and Matheson) at SRI (Miller *et al.*, 1976). Howard and Matheson's method involved the derivation of a decision tree from a decision network, but in general the tree is of exponential size. Shachter (1986) developed a method for making decisions based directly on a decision network, without the creation of an intermediate decision tree. This algorithm was also one of the first to provide complete inference for multiply connected Bayesian networks. Recent work by Nilsson and

Lauritzen (2000) links algorithms for decision networks to ongoing developments in clustering algorithms for Bayesian networks. The collection by Oliver and Smith (1990) has a number of useful articles on decision networks, as does the 1990 special issue of the journal *Networks*. Papers on decision networks and utility modeling also appear regularly in the journal *Management Science*.

Information value theory was first analyzed by Ron Howard (1966). His paper ends with the remark "If information value theory and associated decision theoretic structures do not in the future occupy a large part of the education of engineers, then the engineering profession will find that its traditional role of managing scientific and economic resources for the benefit of man has been forfeited to another profession." To date, the implied revolution in managerial methods has not occurred, although this may change as the use of information value theory in Bayesian expert systems becomes more widespread.

Surprisingly few AI researchers adopted decision-theoretic tools after the early applications in medical decision making described in Chapter 13. One of the few exceptions was Jerry Feldman, who applied decision theory to problems in vision (Feldman and Yakimovsky, 1974) and planning (Feldman and Sproull, 1977). After the resurgence of interest in probabilistic methods in AI in the 1980s, decision-theoretic expert systems gained widespread acceptance (Horvitz *et al.*, 1988). In fact, from 1991 onward, the cover design of the journal *Artificial Intelligence* has depicted a decision network, although some artistic license appears to have been taken with the direction of the arrows.

EXERCISES

16.1 (Adapted from David Heckerman.) This exercise concerns the **Almanac Game**, which is used by decision analysts to calibrate numeric estimations. For each of the questions that follow, give your best guess of the answer, that is, a number that you think is as likely to be too high as it is to be too low. Also give your guess at a 25th percentile estimate, that is, a number that you think has a 25% chance of being too high, and a 75% chance of being too low. Do the same for the 75th percentile. (Thus, you should give three estimates in all—low, median, and high—for each question.)

a. Number of passengers who flew between New York and Los Angeles in 1989.

b. Population of Warsaw in 1992.

c. Year in which Coronado discovered the Mississippi River.

d. Number of votes received by Jimmy Carter in the 1976 presidential election.

e. Age of the oldest living tree, as of 2002.

f. Height of the Hoover Dam in feet.

g. Number of eggs produced in Oregon in 1985.

h. Number of Buddhists in the world in 1992.

i. Number of deaths due to AIDS in the United States in 1981.

j. Number of U.S. patents granted in 1901.

The correct answers appear after the last exercise of this chapter. From the point of view of decision analysis, the interesting thing is not how close your median guesses came to the real answers, but rather how often the real answer came within your 25% and 75% bounds. If it was about half the time, then your bounds are accurate. But if you're like most people, you will be more sure of yourself than you should be, and fewer than half the answers will fall within the bounds. With practice, you can calibrate yourself to give realistic bounds, and thus be more useful in supplying information for decision making. Try this second set of questions and see if there is any improvement:

a. Year of birth of Zsa Zsa Gabor.

b. Maximum distance from Mars to the sun in miles.

c. Value in dollars of exports of wheat from the United States in 1992.

d. Tons handled by the port of Honolulu in 1991.

e. Annual salary in dollars of the governor of California in 1993.

f. Population of San Diego in 1990.

g. Year in which Roger Williams founded Providence, Rhode Island.

h. Height of Mt. Kilimanjaro in feet.

i. Length of the Brooklyn Bridge in feet.

j. Number of deaths due to automobile accidents in the United States in 1992.

16.2 Tickets to a lottery cost \$1. There are two possible prizes: a \$10 payoff with probability 1/50, and a \$1,000,000 payoff with probability 1/2,000,000. What is the expected monetary value of a lottery ticket? When (if ever) is it rational to buy a ticket? Be precise—show an equation involving utilities. You may assume current wealth of $\$k$ and that $U(S_k) = 0$. You may also assume that $U(S_{k+10}) = 10 \times U(S_{k+1}))$, but you may not make any assumptions about $U(S_{k+1,000,000})$. Sociological studies show that people with lower income buy a disproportionate number of lottery tickets. Do you think this is because they are worse decision makers or because they have a different utility function?

16.3 In 1738, J. Bernoulli investigated the St. Petersburg paradox, which works as follows. You have the opportunity to play a game in which a fair coin is tossed repeatedly until it comes up heads. If the first heads appears on the nth toss, you win 2^n dollars.

a. Show that the expected monetary value of this game is infinite.

b. How much would you, personally, pay to play the game?

c. Bernoulli resolved the apparent paradox by suggesting that the utility of money is measured on a logarithmic scale (i.e., $U(S_n) = a \log_2 n + b$, where S_n is the state of having \$n). What is the expected utility of the game under this assumption?

d. What is the maximum amount that it would be rational to pay to play the game, assuming that one's initial wealth is $\$k$?

16.4 Assess your own utility for different incremental amounts of money by running a series of preference tests between some definite amount M_1 and a lottery $[p, M_2; (1-p), 0]$. Choose

different values of M_1 and M_2, and vary p until you are indifferent between the two choices. Plot the resulting utility function.

16.5 Write a computer program to automate the process in Exercise 16.4. Try your program out on several people of different net worth and political outlook. Comment on the consistency of your results, both for an individual and across individuals.

16.6 How much is a micromort worth to you? Devise a protocol to determine this. Ask questions based both on paying to avoid risk and being paid to accept risk.

16.7 Show that if X_1 and X_2 are preferentially independent of X_3, and X_2 and X_3 are preferentially independent of X_1, then X_3 and X_1 are preferentially independent of X_2.

16.8 This exercise completes the analysis of the airport-siting problem in Figure 16.5.

a. Provide reasonable variable domains, probabilities, and utilities for the network, assuming that there are three possible sites.

b. Solve the decision problem.

c. What happens if changes in technology mean that each aircraft generates half as much noise?

d. What if noise avoidance becomes three times more important?

e. Calculate the VPI for *AirTraffic*, *Litigation*, and *Construction* in your model.

16.9 Repeat Exercise 16.8, using the action-utility representation shown in Figure 16.6.

16.10 For either of the airport-siting diagrams from Exercises 16.8 and 16.9, to which conditional probability table entry is the utility most sensitive, given the available evidence?

16.11 (Adapted from Pearl (1988).) A used-car buyer can decide to carry out various tests with various costs (e.g., kick the tires, take the car to a qualified mechanic) and then, depending on the outcome of the tests, decide which car to buy. We will assume that the buyer is deciding whether to buy car c_1, that there is time to carry out at most one test, and that t_1 is the test of c_1 and costs \$50.

A car can be in good shape (quality q^+) or bad shape (quality q^-), and the tests might help to indicate what shape the car is in. Car c_1 costs \$1,500, and its market value is \$2,000 if it is in good shape; if not, \$700 in repairs will be needed to make it in good shape. The buyer's estimate is that c_1 has a 70% chance of being in good shape.

a. Draw the decision network that represents this problem.

b. Calculate the expected net gain from buying c_1, given no test.

c. Tests can be described by the probability that the car will pass or fail the test given that the car is in good or bad shape. We have the following information:
$P(pass(c_1, t_1)|q^+(c_1)) = 0.8$
$P(pass(c_1, t_1)|q^-(c_1)) = 0.35$
Use Bayes' theorem to calculate the probability that the car will pass (or fail) its test and hence the probability that it is in good (or bad) shape given each possible test outcome.

d. Calculate the optimal decisions given either a pass or a fail, and their expected utilities.

e. Calculate the value of information of the test, and derive an optimal conditional plan for the buyer.

16.12 Prove that the value of information is nonnegative and order-independent, as stated in Section 16.6. Explain how it is that one can make a worse decision after receiving information than one would have made before receiving it.

16.13 Modify and extend the Bayesian network code in the code repository to provide for creation and evaluation of decision networks and the calculation of information value.

The answers to Exercise 16.1 (where M stands for million): First set: 3M, 1.6M, 1541, 41M, 4768, 221, 649M, 295M, 132, 25,546. Second set: 1917, 155M, 4,500M, 11M, 120,000, 1.1M, 1,636, 19,340, 1,595, 41,710.

17 MAKING COMPLEX DECISIONS

In which we examine methods for deciding what to do today, given that we may decide again tomorrow.

SEQUENTIAL DECISION PROBLEMS

In this chapter, we address the computational issues involved in making decisions. Whereas Chapter 16 was concerned with one-shot or episodic decision problems, in which the utility of each action's outcome was well known, we will be concerned here with **sequential decision problems**, in which the agent's utility depends on a sequence of decisions. Sequential decision problems, which include utilities, uncertainty, and sensing, generalize the search and planning problems described in Parts II and IV. Section 17.1 explains how sequential decision problems are defined, and Sections 17.2 and 17.3 explain how they can be solved to produce optimal behavior that balances the risks and rewards of acting in an uncertain environment. Section 17.4 extends these ideas to the case of partially observable environments, and Section 17.5 develops a complete design for decision-theoretic agents in partially observable environments, combining dynamic Bayesian networks from Chapter 15 with decision networks from Chapter 16.

The second part of the chapter covers environments with multiple agents. In such environments, the notion of optimal behavior becomes much more complicated by the interactions among the agents. Section 17.6 introduces the main ideas of **game theory**, including the idea that rational agents might need to behave randomly. Section 17.7 looks at how multiagent systems can be designed so that multiple agents can achieve a common goal.

17.1 SEQUENTIAL DECISION PROBLEMS

An example

Suppose that an agent is situated in the 4×3 environment shown in Figure 17.1(a). Beginning in the start state, it must choose an action at each time step. The interaction with the environment terminates when the agent reaches one of the goal states, marked +1 or −1. In each location, the available actions are called *Up*, *Down*, *Left*, and *Right*. We will assume for now that the environment is **fully observable**, so that the agent always knows where it is.

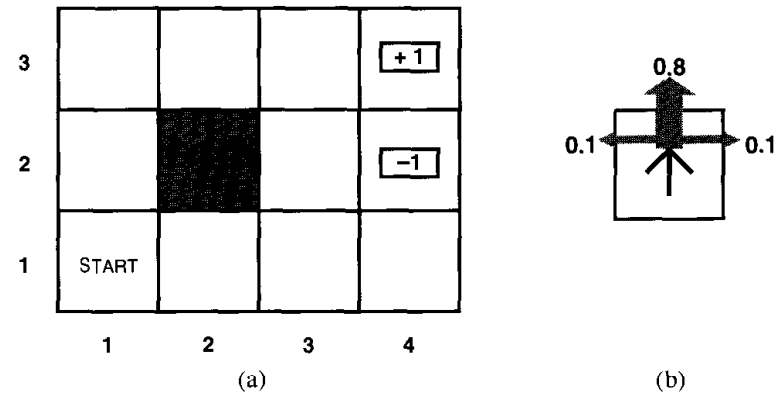


Figure 17.1 (a) A simple 4×3 environment that presents the agent with a sequential decision problem. (b) Illustration of the transition model of the environment: the "intended" outcome occurs with probability 0.8, but with probability 0.2 the agent moves at right angles to the intended direction. A collision with a wall results in no movement. The two terminal states have reward +1 and −1, respectively, and all other states have a reward of −0.04.

If the environment were deterministic, a solution would be easy: [*Up, Up, Right, Right, Right*]. Unfortunately, the environment won't always go along with this solution, because the actions are unreliable. The particular model of stochastic motion that we adopt is illustrated in Figure 17.1(b). Each action achieves the intended effect with probability 0.8, but the rest of the time, the action moves the agent at right angles to the intended direction. Furthermore, if the agent bumps into a wall, it stays in the same square. For example, from the start square (1,1), the action *Up* moves the agent to (1,2) with probability 0.8, but with probability 0.1, it moves right to (2,1), and with probability 0.1, it moves left, bumps into the wall, and stays in (1,1). In such an environment, the sequence $[Up, Up, Right, Right, Right]$ goes up around the barrier and reaches the goal state at (4,3) with probability $0.8^5 = 0.32768$. There is also a small chance of accidentally reaching the goal by going the other way around with probability $0.1^4 \times 0.8$, for a grand total of 0.32776. (See also Exercise 17.1.)

TRANSITION MODEL

A specification of the outcome probabilities for each action in each possible state is called a **transition model** (or just "model," whenever no confusion can arise). We will use $T(s, a, s')$ to denote the probability of reaching state s' if action a is done in state s. We will assume that transitions are **Markovian** in the sense of Chapter 15, that is, the probability of reaching s' from s depends only on s and not on the history of earlier states. For now, you can think of $T(s, a, s')$ as a big three-dimensional table containing probabilities. Later, in Section 17.5, we will see that the transition model can be represented as a **dynamic Bayesian network**, just as in Chapter 15.

REWARD

To complete the definition of the task environment, we must specify the utility function for the agent. Because the decision problem is sequential, the utility function will depend on a sequence of states—an **environment history**—rather than on a single state. Later in this section, we will investigate how such utility functions can be specified in general; for now, we will simply stipulate that in each state s, the agent receives a **reward** $R(s)$, which may be positive or negative, but must be bounded. For our particular example, the reward is -0.04 in all states except the terminal states (which have rewards +1 and −1). The utility of

an environment history is just (for now) the *sum* of the rewards received. For example, if the agent reaches the +1 state after 10 steps, its total utility will be 0.6. The negative reward of –0.04 gives the agent an incentive to reach (4,3) quickly, so our environment is a stochastic generalization of the search problems of Chapter 3. Another way of saying this is that the agent does not enjoy living in this environment and so wants to get out of the game as soon as possible.

MARKOV DECISION PROCESS

The specification of a sequential decision problem for a fully observable environment with a Markovian transition model and additive rewards is called a **Markov decision process**, or **MDP**. An MDP is defined by the following three components:

Initial State: S_0

Transition Model: $T(s, a, s')$

Reward Function:[1] $R(s)$

The next question is, what does a solution to the problem look like? We have seen that any fixed action sequence won't solve the problem, because the agent might end up in a state other than the goal. Therefore, a solution must specify what the agent should do for *any* state that the agent might reach. A solution of this kind is called a **policy**. We usually denote a policy by π, and $\pi(s)$ is the action recommended by the policy π for state s. If the agent has a complete policy, then no matter what the outcome of any action, the agent will always know what to do next.

POLICY

Each time a given policy is executed starting from the initial state, the stochastic nature of the environment will lead to a different environment history. The quality of a policy is therefore measured by the *expected* utility of the possible environment histories generated by that policy. An **optimal policy** is a policy that yields the highest expected utility. We use π^* to denote an optimal policy. Given π^*, the agent decides what to do by consulting its current percept, which tells it the current state s, and then executing the action $\pi^*(s)$. A policy represents the agent function explicitly and is therefore a description of a simple reflex agent, computed from the information used for a utility-based agent:

OPTIMAL POLICY

An optimal policy for the world of Figure 17.1 is shown in Figure 17.2(a). Notice that, because the cost of taking a step is fairly small compared with the penalty for ending up in (4,2) by accident, the optimal policy for the state (3,1) is conservative. The policy recommends taking the long way round, rather than taking the short cut and thereby risking entering (4,2).

The balance of risk and reward changes depends on the value of $R(s)$ for the nonterminal states. Figure 17.2(b) shows optimal policies for four different ranges of $R(s)$. When $R(s) \leq -1.6284$, life is so painful that the agent heads straight for the nearest exit, even if the exit is worth –1. When $-0.4278 \leq R(s) \leq -0.0850$, life is quite unpleasant; the agent takes the shortest route to the +1 state and is willing to risk falling into the –1 state by accident. In particular, the agent takes the shortcut from (3,1). When life is only slightly dreary ($-0.0221 < R(s) < 0$), the optimal policy takes *no risks at all.* In (4,1) and (3,2), the agent

[1] Some definitions of MDPs allow the reward to depend on the action and outcome too, so the reward function is $R(s, a, s')$. This simplifies the description of some environments but does not change the problem in any fundamental way.

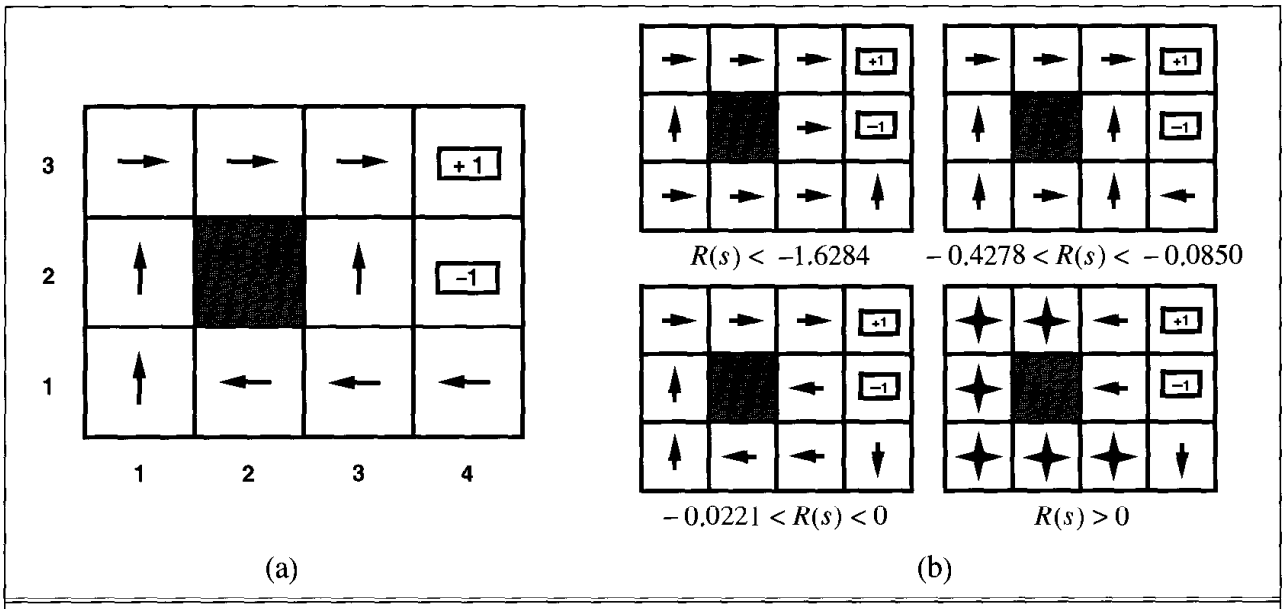


Figure 17.2 (a) An optimal policy for the stochastic environment with $R(s) = -0.04$ in the nonterminal states. (b) Optimal policies for four different ranges of $R(s)$.

heads directly away from the −1 state so that it cannot fall in by accident, even though this means banging its head against the wall quite a few times. Finally, if $R(s) > 0$, then life is positively enjoyable and the agent avoids *both* exits. As long as the actions in (4,1), (3,2), and (3,3) are as shown, every policy is optimal, and the agent obtains infinite total reward because it never enters a terminal state. Surprisingly, it turns out that there are six other optimal policies for various ranges of $R(s)$; Exercise 17.7 asks you to find them.

The careful balancing of risk and reward is a characteristic of MDPs that does not arise in deterministic search problems; moreover, it is a characteristic of many real-world decision problems. For this reason, MDPs have been studied in several fields, including AI, operations research, economics, and control theory. Dozens of algorithms have been proposed for calculating optimal policies. In sections 17.2 and 17.3 we will describe two of the most important algorithm families. First, however, we must complete our investigation of utilities and policies for sequential decision problems.

Optimality in sequential decision problems

In the MDP example in Figure 17.1, the performance of the agent was measured by a sum of rewards for the states visited. This choice of performance measure is not arbitrary, but it is not the only possibility. This section investigates the possible choices for the performance measure—that is, choices for the utility function on environment histories, which we will write as $U_h([s_0, s_1, \ldots, s_n])$. The section draws on ideas from Chapter 16 and is somewhat technical; the main points are summarized at the end.

FINITE HORIZON
INFINITE HORIZON

The first question to answer is whether there is a **finite horizon** or an **infinite horizon** for decision making. A finite horizon means that there is a *fixed* time N after which nothing matters—the game is over, so to speak. Thus, $U_h([s_0, s_1, \ldots, s_{N+k}]) = U_h([s_0, s_1, \ldots, s_N])$ for all $k > 0$. For example, suppose an agent starts at (3,1) in the 4×3 world of Figure 17.1,

and suppose that $N = 3$. Then, to have any chance of reaching the +1 state, the agent must head directly for it, and the optimal action is to go *Up*. On the other hand, if $N = 100$ then there is plenty of time to take the safe route by going *Left*. *So, with a finite horizon, the optimal action in a given state could change over time.* We say that the optimal policy for a finite horizon is **nonstationary**. With no fixed time limit, on the other hand, there is no reason to behave differently in the same state at different times. Hence, the optimal action depends only on the current state, and the optimal policy is **stationary**. Policies for the infinite-horizon case are therefore simpler than those for the finite-horizon case, and we will deal mainly with the infinite-horizon case in this chapter.[2] Note that "infinite horizon" does not necessarily mean that all state sequences are infinite; it just means that there is no fixed deadline. In particular, there can be finite state sequences in an infinite-horizon MDP containing a terminal state.

NONSTATIONARY POLICY

STATIONARY POLICY

The next question we must decide is how to calculate the utility of state sequences. We can view this as a question in **multiattribute utility theory** (see Section 16.4), where each state s_i is viewed as an attribute of the state sequence $[s_0, s_1, s_2 \ldots]$. To obtain a simple expression in terms of the attributes, we will need to make some sort of preference independence assumption. The most natural assumption is that the agent's preferences between state sequences are **stationary**. Stationarity for preferences means the following: if two state sequences $[s_0, s_1, s_2, \ldots]$ and $[s'_0, s'_1, s'_2, \ldots]$ begin with the same state (i.e., $s_0 = s'_0$) then the two sequences should be preference-ordered the same way as the sequences $[s_1, s_2, \ldots]$ and $[s'_1, s'_2, \ldots]$. In English, this means that if you prefer one future to another starting tomorrow, then you should still prefer that future if it were to start today. Stationarity is a fairly innocuous-looking assumption with very strong consequences: it turns out that under stationarity there are just two ways to assign utilities to sequences:

STATIONARY PREFERENCE

ADDITIVE REWARDS

1. **Additive rewards**: The utility of a state sequence is

$$U_h([s_0, s_1, s_2, \ldots]) = R(s_0) + R(s_1) + R(s_2) + \cdots .$$

The 4×3 world in Figure 17.1 uses additive rewards. Notice that additivity was used implicitly in our use of path cost functions in heuristic search algorithms (Chapter 4).

DISCOUNTED REWARDS

2. **Discounted rewards**: The utility of a state sequence is

$$U_h([s_0, s_1, s_2, \ldots]) = R(s_0) + \gamma R(s_1) + \gamma^2 R(s_2) + \cdots ,$$

DISCOUNT FACTOR

where the **discount factor** γ is a number between 0 and 1. The discount factor describes the preference of an agent for current rewards over future rewards. When γ is close to 0, rewards in the distant future are viewed as insignificant. When γ is 1, discounted rewards are exactly equivalent to additive rewards, so additive rewards are a special case of discounted rewards. Discounting appears to be a good model of both animal and human preferences over time. A discount factor of γ is equivalent to an interest rate of $(1/\gamma) - 1$.

For reasons that will shortly become clear, we will assume discounted rewards in the remainder of the chapter, although sometimes we will allow $\gamma = 1$.

[2] This is for completely observable environments. We will see later that for partially observable environments, the infinite-horizon case is not so simple.

Lurking beneath our choice of infinite horizons is a problem: if the environment does not contain a terminal state, or if the agent never reaches one, then all environment histories will be infinitely long, and utilities with additive rewards will generally be infinite. Now, we can agree that $+\infty$ is better than $-\infty$, but comparing two state sequences, both having $+\infty$ utility is more difficult. There are three solutions, two of which we have seen already:

1. With discounted rewards, the utility of an infinite sequence is *finite*. In fact, if rewards are bounded by $R_{\max}$ and $\gamma < 1$, we have
$$U_h([s_0, s_1, s_2, \ldots]) = \sum_{t=0}^{\infty} \gamma^t R(s_t) \leq \sum_{t=0}^{\infty} \gamma^t R_{\max} = R_{\max}/(1-\gamma) \,, \tag{17.1}$$
using the standard formula for the sum of an infinite geometric series.
2. If the environment contains terminal states *and if the agent is guaranteed to get to one eventually*, then we will never need to compare infinite sequences. A policy that is guaranteed to reach a terminal state is called a **proper policy**. With proper policies, we can use $\gamma = 1$ (i.e., additive rewards). The first three policies shown in Figure 17.2(b) are proper, but the fourth is improper. It gains infinite total reward by staying away from the terminal states when the reward for the nonterminal states is positive. The existence of improper policies can cause the standard algorithms for solving MDPs to fail with additive rewards, and so provides a good reason for using discounted rewards.

PROPER POLICY

3. Another possibility is to compare infinite sequences in terms of the **average reward** obtained per time step. Suppose that square (1,1) in the 4×3 world has a reward of 0.1 while the other nonterminal states have a reward of 0.01. Then a policy that does its best to stay in (1,1) will have higher average reward than one that stays elsewhere. Average reward is a useful criterion for some problems, but the analysis of average-reward algorithms is beyond the scope of this book.

AVERAGE REWARD

In sum, the use of discounted rewards presents the fewest difficulties in evaluating state sequences. The final step is to show how to choose between policies, bearing in mind that a given policy π generates not one state sequence, but a whole range of possible state sequences, each with a specific probability determined by the transition model for the environment. Thus, the value of a policy is the *expected* sum of discounted rewards obtained, where the expectation is taken over all possible state sequences that could occur, given that the policy is executed. An optimal policy π^* satisfies
$$\pi^* = \operatorname*{argmax}_{\pi} E\left[\sum_{t=0}^{\infty} \gamma^t R(s_t) \mid \pi\right] . \tag{17.2}$$
The next two sections describe algorithms for finding optimal policies.

17.2 Value Iteration

VALUE ITERATION

In this section, we present an algorithm, called **value iteration**, for calculating an optimal policy. The basic idea is to calculate the utility of each *state* and then use the state utilities to select an optimal action in each state.

Utilities of states

The utility of states is defined in terms of the utility of state sequences. Roughly speaking, the utility of a state is the expected utility of the state sequences that might follow it. Obviously, the state sequences depend on the policy that is executed, so we begin by defining the utility $U^\pi(s)$ with respect to a specific policy π. If we let s_t be the state the agent is in after executing π for t steps (note that s_t is a random variable), then we have

$$U^\pi(s) = E\left[\sum_{t=0}^{\infty} \gamma^t R(s_t) \mid \pi, s_0 = s\right] . \tag{17.3}$$

Given this definition, the true utility of a state, which we write as $U(s)$, is just $U^{\pi^*}(s)$—that is, the expected sum of discounted rewards if the agent executes an optimal policy. Notice that $U(s)$ and $R(s)$ are quite different quantities; $R(s)$ is the "short-term" reward for being *in* s, whereas $U(s)$ is the "long-term" total reward from s onwards. Figure 17.3 shows the utilities for the 4×3 world. Notice that the utilities are higher for states closer to the +1 exit, because fewer steps are required to reach the exit.

3	0.812	0.868	0.918	+1
2	0.762		0.660	−1
1	0.705	0.655	0.611	0.388
	1	2	3	4

Figure 17.3 The utilities of the states in the 4×3 world, calculated with $\gamma = 1$ and $R(s) = -0.04$ for nonterminal states.

The utility function $U(s)$ allows the agent to select actions by using the Maximum Expected Utility principle from Chapter 16—that is, choose the action that maximizes the expected utility of the subsequent state:

$$\pi^*(s) = \operatorname*{argmax}_a \sum_{s'} T(s, a, s') U(s') . \tag{17.4}$$

Now, if the utility of a state is the expected sum of discounted rewards from that point onwards, then there is a direct relationship between the utility of a state and the utility of its neighbors: *the utility of a state is the immediate reward for that state plus the expected discounted utility of the next state, assuming that the agent chooses the optimal action.* That is, the utility of a state is given by

$$U(s) = R(s) + \gamma \max_a \sum_{s'} T(s, a, s') U(s') . \tag{17.5}$$

BELLMAN EQUATION

Equation (17.5) is called the **Bellman equation**, after Richard Bellman (1957). The utilities of the states—defined by Equation (17.3) as the expected utility of subsequent state sequences—are solutions of the set of Bellman equations. In fact, they are the *unique* solutions, as we show in the next two sections.

Let us look at one of the Bellman equations for the 4×3 world. The equation for the state (1,1) is

$$\begin{aligned} U(1,1) = -0.04 + \gamma \max\{ \; & 0.8U(1,2) + 0.1U(2,1) + 0.1U(1,1), && (Up) \\ & 0.9U(1,1) + 0.1U(1,2), && (Left) \\ & 0.9U(1,1) + 0.1U(2,1), && (Down) \\ & 0.8U(2,1) + 0.1U(1,2) + 0.1U(1,1) \; \} && (Right) \end{aligned}$$

When we plug in the numbers from Figure 17.3, we find that *Up* is the best action.

The value iteration algorithm

The Bellman equation is the basis of the value iteration algorithm for solving MDPs. If there are n possible states, then there are n Bellman equations, one for each state. The n equations contain n unknowns—the utilities of the states. So we would like to solve these simultaneous equations to find the utilities. There is one problem: the equations are *nonlinear*, because the "max" operator is not a linear operator. Whereas systems of linear equations can be solved quickly using linear algebra techniques, systems of nonlinear equations are more problematic. One thing to try is an *iterative* approach. We start with arbitrary initial values for the utilities, calculate the right-hand side of the equation, and plug it into the left-hand side—thereby updating the utility of each state from the utilities of its neighbors. We repeat this until we reach an equilibrium. Let $U_i(s)$ be the utility value for state s at the ith iteration. The iteration step, called a **Bellman update**, looks like this:

BELLMAN UPDATE

$$U_{i+1}(s) \leftarrow R(s) + \gamma \max_a \sum_{s'} T(s,a,s')U_i(s') \; . \tag{17.6}$$

If we apply the Bellman update infinitely often, we are guaranteed to reach an equilibrium (see the next subsection), in which case the final utility values must be solutions to the Bellman equations. In fact, they are also the *unique* solutions, and the corresponding policy (obtained using Equation (17.4)) is optimal. The algorithm, called VALUE-ITERATION, is shown in Figure 17.4.

We can apply value iteration to the 4×3 world in Figure 17.1(a). Starting with initial values of zero, the utilities evolve as shown in Figure 17.5(a). Notice how the states at different distances from (4,3) accumulate negative reward until, at some point, a path is found to (4,3) whereupon the utilities start to increase. We can think of the value iteration algorithm as *propagating information* through the state space by means of local updates.

Convergence of value iteration

We said that value iteration eventually converges to a unique set of solutions of the Bellman equations. In this section, we explain why this happens. We introduce some useful mathematical ideas along the way, and we obtain some methods for assessing the error in the utility

```
function VALUE-ITERATION(mdp, ε) returns a utility function
  inputs: mdp, an MDP with states S, transition model T, reward function R, discount γ
          ε, the maximum error allowed in the utility of any state
  local variables: U, U', vectors of utilities for states in S, initially zero
                   δ, the maximum change in the utility of any state in an iteration

  repeat
     U ← U'; δ ← 0
     for each state s in S do
        U'[s] ← R[s] + γ max_a Σ_s' T(s, a, s') U[s']
        if |U'[s] − U[s]| > δ then δ ← |U'[s] − U[s]|
  until δ < ε(1 − γ)/γ
  return U
```

Figure 17.4 The value iteration algorithm for calculating utilities of states. The termination condition is from Equation (17.8).

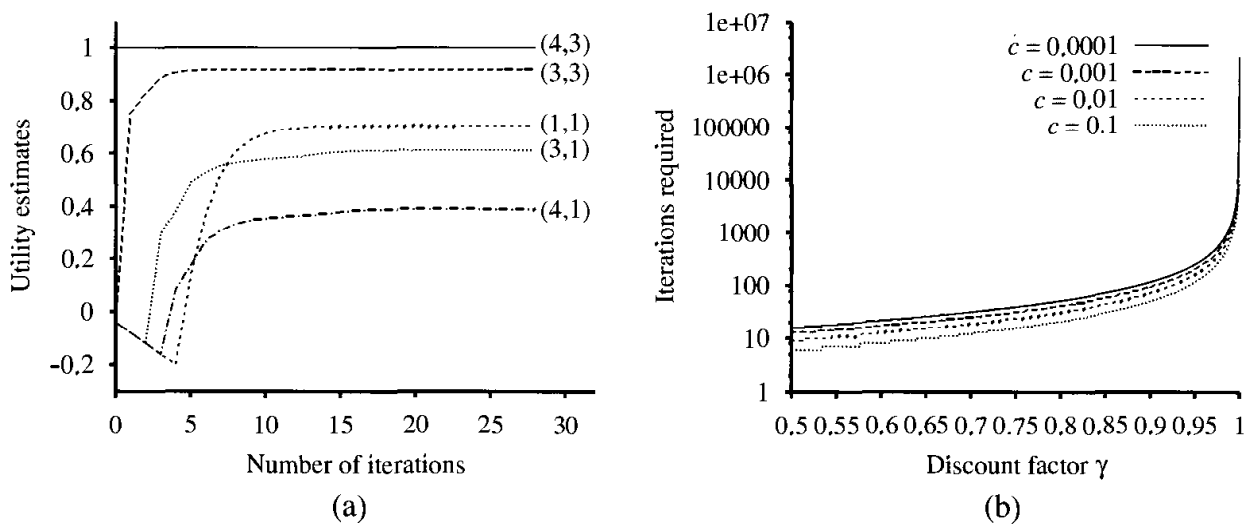


Figure 17.5 (a) Graph showing the evolution of the utilities of selected states using value iteration. (b) The number of value iterations k required to guarantee an error of at most $\epsilon = c \cdot R_{\max}$, for different values of c, as a function of the discount factor γ.

function returned when the algorithm is terminated early; this is useful because it means that we don't have to run forever. The section is quite technical.

CONTRACTION

The basic concept used in showing that value iteration converges is the notion of a **contraction**. Roughly speaking, a contraction is a function of one argument that, when applied to two different inputs in turn, produces two output values that are "closer together," by at least some constant amount, than the original arguments. For example, the function "divide by two" is a contraction, because, after we divide any two numbers by two, their difference is halved. Notice that the "divide by two" function has a fixed point, namely zero, that is un-

changed by the application of the function. From this example, we can discern two important properties of contractions:

- A contraction has only one fixed point; if there were two fixed points they would not get closer together when the function was applied, so it would not be a contraction.
- When the function is applied to any argument, the value must get closer to the fixed point (because the fixed point does not move), so repeated application of a contraction always reaches the fixed point in the limit.

Now, suppose we view the Bellman update (Equation (17.6)) as an operator B that is applied simultaneously to update the utility of every state. Let U_i denote the vector of utilities for all the states at the ith iteration. Then the Bellman update equation can be written as

$$U_{i+1} \leftarrow B\,U_i\,.$$

Next, we need a way to measure distances between utility vectors. We will use the **max norm**, which measures the length of a vector by the length of its biggest component:

MAX NORM

$$||U|| = \max_s |U(s)|\,.$$

With this definition, the "distance" between two vectors, $||U - U'||$, is the maximum difference between any two corresponding elements. The main result of this section is the following: *Let U_i and U_i' be any two utility vectors. Then we have*

$$||B\,U_i - B\,U_i'|| \leq \gamma\,||U_i - U_i'||\,. \tag{17.7}$$

That is, the Bellman update is a contraction by a factor of γ on the space of utility vectors. Hence, value iteration always converges to a unique solution of the Bellman equations.

In particular, we can replace U_i' in Equation (17.7) with the *true* utilities U, for which $B\,U = U$. Then we obtain the inequality

$$||B\,U_i - U|| \leq \gamma\,||U_i - U||\,.$$

So, if we view $||U_i - U||$ as the *error* in the estimate U_i, we see that the error is reduced by a factor of at least γ on each iteration. This means that value iteration converges exponentially fast. We can calculate the number of iterations required to reach a specified error bound ϵ as follows: First, recall from Equation (17.1) that the utilities of all states are bounded by $\pm R_{\max}/(1-\gamma)$. This means that the maximum initial error $||U_0 - U|| \leq 2R_{\max}/(1-\gamma)$. Suppose we run for N iterations to reach an error of at most ϵ. Then, because the error is reduced by at least γ each time, we require $\gamma^N \cdot 2R_{\max}/(1-\gamma) \leq \epsilon$. Taking logs, we find

$$N = \lceil \log(2R_{\max}/\epsilon(1-\gamma))/\log(1/\gamma) \rceil$$

iterations suffice. Figure 17.5(b) shows how N varies with γ, for different values of the ratio $\epsilon/R_{\max}$. The good news is that, because of the exponentially fast convergence, N does not depend much on the ratio $\epsilon/R_{\max}$. The bad news is that N grows rapidly as γ becomes close to 1. We can get fast convergence if we make γ small, but this effectively gives the agent a short horizon and could miss the long-term effects of the agent's actions.

The error bound in the preceding paragraph gives some idea of the factors influencing the runtime of the algorithm, but is sometimes overly conservative as a method of deciding when to stop the iteration. For the latter purpose, we can use a bound relating the error

to the size of the Bellman update on any given iteration. From the contraction property (Equation (17.7)), it can be shown that if the update is small (i.e., no state's utility changes by much), then the error, compared with the true utility function, also is small. More precisely,

$$\text{if} \quad ||U_{i+1} - U_i|| < \epsilon(1-\gamma)/\gamma \quad \text{then} \quad ||U_{i+1} - U|| < \epsilon \,. \tag{17.8}$$

This is the termination condition used in the VALUE-ITERATION algorithm of Figure 17.4.

So far, we have analyzed the error in the utility function returned by the value iteration algorithm. *What the agent really cares about, however, is how well it will do if it makes its decisions on the basis of this utility function.* Suppose that after i iterations of value iteration, the agent has an estimate U_i of the true utility U and obtains the MEU policy π_i based on one-step look-ahead using U_i (as in Equation (17.4)). Will the resulting behavior be nearly as good as the optimal behavior? This is a crucial question for any real agent, and it turns out that the answer is yes. $U^{\pi_i}(s)$ is the utility obtained if π_i is executed starting in s, and the

POLICY LOSS

policy loss $||U^{\pi_i} - U||$ is the most the agent can lose by executing π_i instead of the optimal policy π^*. The policy loss of π_i is connected to the error in U_i by the following inequality:

$$\text{if} \quad ||U_i - U|| < \epsilon \quad \text{then} \quad ||U^{\pi_i} - U|| < 2\epsilon\gamma/(1-\gamma) \,. \tag{17.9}$$

In practice, it often occurs that π_i becomes optimal long before U_i has converged. Figure 17.6 shows how the maximum error in U_i and the policy loss approach zero as the value iteration process proceeds for the 4×3 environment with $\gamma = 0.9$. The policy π_i is optimal when $i = 4$, even though the maximum error in U_i is still 0.46.

Now we have everything we need to use value iteration in practice. We know that it converges to the correct utilities, we can bound the error in the utility estimates if we stop after a finite number of iterations, and we can bound the policy loss that results from executing the corresponding MEU policy. As a final note, all of the results in this section depend on discounting with $\gamma < 1$. If $\gamma = 1$ and the environment contains terminal states, then a similar set of convergence results and error bounds can be derived whenever certain technical conditions are satisfied.

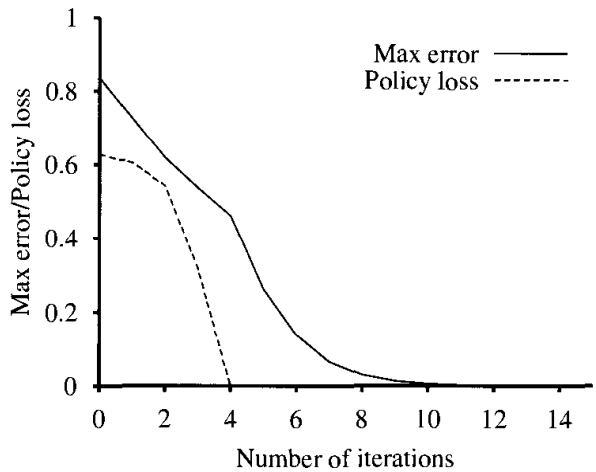


Figure 17.6 The maximum error $||U_i - U||$ of the utility estimates and the policy loss $||U^{\pi_i} - U||$ compared with the optimal policy, as a function of the number of iterations of value iteration.

17.3 POLICY ITERATION

In the previous section, we observed that it is possible to get an optimal policy even when the utility function estimate is inaccurate. If one action is clearly better than all others, then the exact magnitude of the utilities on the states involved need not be precise. This insight suggests an alternative way to find optimal policies. The **policy iteration** algorithm alternates the following two steps, beginning from some initial policy π_0:

POLICY ITERATION

POLICY EVALUATION

- **Policy evaluation**: given a policy π_i, calculate $U_i = U^{\pi_i}$, the utility of each state if π_i were to be executed.

POLICY IMPROVEMENT

- **Policy improvement**: Calculate a new MEU policy π_{i+1}, using one-step look-ahead based on U_i (as in Equation (17.4)).

The algorithm terminates when the policy improvement step yields no change in the utilities. At this point, we know that the utility function U_i is a fixed point of the Bellman update, so it is a solution to the Bellman equations, and π_i must be an optimal policy. Because there are only finitely many policies for a finite state space, and each iteration can be shown to yield a better policy, policy iteration must terminate. The algorithm is shown in Figure 17.7.

The policy improvement step is obviously straightforward, but how do we implement the POLICY-EVALUATION routine? It turns out that doing so is much simpler than solving the standard Bellman equations (which is what value iteration does), because the action in each state is fixed by the policy. At the ith iteration, the policy π_i specifies the action $\pi_i(s)$ in state s. This means that we have a simplified version of the Bellman equation (17.5) relating

```
function POLICY-ITERATION(mdp) returns a policy
  inputs: mdp, an MDP with states S, transition model T
  local variables: U, U', vectors of utilities for states in S, initially zero
                   π, a policy vector indexed by state, initially random

  repeat
     U ← POLICY-EVALUATION(π, U, mdp)
     unchanged? ← true
     for each state s in S do
        if max_a Σ_s' T(s,a,s') U[s'] > Σ_s' T(s,π[s],s') U[s'] then
           π[s] ← argmax_a Σ_s' T(s,a,s') U[s']
           unchanged? ← false
  until unchanged?
  return π
```

Figure 17.7 The policy iteration algorithm for calculating an optimal policy.

the utility of s (under π_i) to the utilities of its neighbors:

$$U_i(s) = R(s) + \gamma \sum_{s'} T(s, \pi_i(s), s') U_i(s') \ . \tag{17.10}$$

For example, suppose π_i is the policy shown in Figure 17.2(a). Then we have $\pi_i(1,1) = Up$, $\pi_i(1,2) = Up$, and so on, and the simplified Bellman equations are

$$\begin{aligned} U_i(1,1) &= -0.04 + 0.8U_i(1,2) + 0.1U_i(1,1) + 0.1U_i(2,1) \ , \\ U_i(1,2) &= -0.04 + 0.8U_i(1,3) + 0.2U_i(1,2) \ , \\ &\vdots \end{aligned}$$

The important point is that these equations are *linear*, because the "max" operator has been removed. For n states, we have n linear equations with n unknowns, which can be solved exactly in time $O(n^3)$ by standard linear algebra methods.

For small state spaces, policy evaluation using exact solution methods is often the most efficient approach. For large state spaces, $O(n^3)$ time might be prohibitive. Fortunately, it is not necessary to do *exact* policy evaluation. Instead, we can perform some number of simplified value iteration steps (simplified because the policy is fixed) to give a reasonably good approximation of the utilities. The simplified Bellman update for this process is

$$U_{i+1}(s) \leftarrow R(s) + \gamma \sum_{s'} T(s, \pi_i(s), s') U_i(s')$$

and this is repeated k times to produce the next utility estimate. The resulting algorithm is called **modified policy iteration**. It is often much more efficient than standard policy iteration or value iteration.

MODIFIED POLICY ITERATION

The algorithms we have described so far require updating the utility or policy for all states at once. It turns out that this is not strictly necessary. In fact, on each iteration, we can pick *any subset* of states and apply *either* kind of updating (policy improvement or simplified value iteration) to that subset. This very general algorithm is called **asynchronous policy iteration**. Given certain conditions on the initial policy and utility function, asynchronous policy iteration is guaranteed to converge to an optimal policy. The freedom to choose any states to work on means that we can design much more efficient heuristic algorithms—for example, algorithms that concentrate on updating the values of states that are likely to be reached by a good policy. This makes a lot of sense in real life: if one has no intention of throwing oneself off a cliff, one should not spend time worrying about the exact value of the resulting states.

ASYNCHRONOUS POLICY ITERATION

17.4 PARTIALLY OBSERVABLE MDPS

The description of Markov decision processes in Section 17.1 assumed that the environment was **fully observable**. With this assumption, the agent always knows which state it is in. This, combined with the Markov assumption for the transition model, means that the optimal policy depends only on the current state. When the environment is only **partially observable**, the situation is, one might say, much less clear. The agent does not necessarily know which

PARTIALLY OBSERVABLE MDP

state it is in, so it cannot execute the action $\pi(s)$ recommended for that state. Furthermore, the utility of a state s and the optimal action in s depend not just on s, but also on *how much the agent knows* when it is in s. For these reasons, **partially observable MDPs** (or POMDPs—pronounced "pom-dee-pees") are usually viewed as much more difficult than ordinary MDPs. We cannot avoid POMDPs, however, because the real world is one.

As an example, consider again the 4×3 world of Figure 17.1, but now let's suppose that the agent has *no sensors whatsoever* and has *no idea where it is*. More precisely, let's suppose the agent's initial state is equally likely to be any of the nine nonterminal states (Figure 17.8(a)). Clearly, if the agent *knew* it was in (3,3), it would move *Right*; if it *knew* it was in (1,1), it would move *Up*; but since it could be anywhere, what should it do? One possible answer is that the agent should first act so as to reduce its uncertainty, and only then should it try heading for the +1 exit. For example, if the agent moves *Left* five times, then it is quite likely to be at the left wall (Figure 17.8(b)). Then, if it moves *Up* five times, it is quite likely to be at the top, probably in the top left corner (Figure 17.8(c)). Finally, if it moves *Right* five times, it has a good chance—about 77.5%—of reaching the +1 exit (Figure 17.8(d)). Continuing to move right thereafter increases its chances to 81.8%. This policy is therefore surprisingly safe, but under it, the agent is rather slow to reach the exit, and has an expected utility of only about 0.08. The optimal policy, which we will describe shortly, does much better.

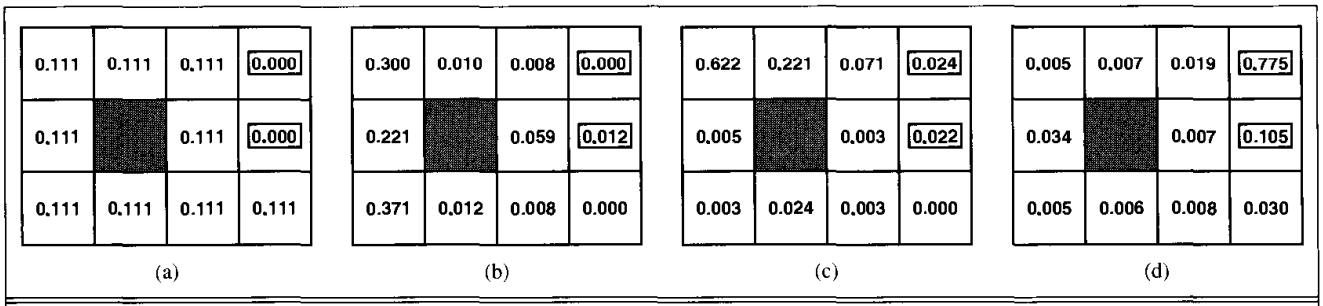


Figure 17.8 (a) The initial probability distribution for the agent's location. (b) After moving *Left* five times. (c) After moving *Up* five times. (d) After moving *Right* five times.

OBSERVATION MODEL

To get a handle on POMDPs, we must first define them properly. A POMDP has the same elements as an MDP—the transition model $T(s, a, s')$ and the reward function $R(s)$—but it also has an **observation model** $O(s, o)$ that specifies the probability of perceiving the observation o in state s.[3] For example, our agent with no sensors has only one possible observation (the empty observation), and this occurs with probability 1 in every state.

In Chapters 3 and 12, we studied nondeterministic and partially observable planning problems and identified the **belief state**—the set of actual states the agent might be in—as a key concept for describing and calculating solutions. In POMDPs, the concept is refined somewhat. A belief state b is now a *probability distribution* over all possible states. For example, the initial belief state in Figure 17.8(a) could be written as $\langle \frac{1}{9}, \frac{1}{9}, \frac{1}{9}, \frac{1}{9}, \frac{1}{9}, \frac{1}{9}, \frac{1}{9}, \frac{1}{9}, \frac{1}{9}, 0, 0 \rangle$.

[3] The observation model is essentially identical to the **sensor model** for temporal processes, as described in Chapter 15. As with the reward function for MDPs, the observation model can also depend on the action and outcome state, but again this change is not fundamental.

We will write $b(s)$ for the probability assigned to the actual state s by belief state b. The agent can calculate its current belief state as the conditional probability distribution over the actual states given the sequence of observations and actions so far. This is essentially the **filtering** task. (See Chapter 15.) The basic recursive filtering equation (15.3 on page 543) shows how to calculate the new belief state from the previous belief state and the new observation. For POMDPs, we also have an action to consider and a slightly different notation, but the result is essentially the same. If $b(s)$ was the previous belief state, and the agent does action a and perceives observation o, then the new belief state is given by

$$b'(s') = \alpha\, O(s', o) \sum_s T(s, a, s') b(s) \tag{17.11}$$

where α is a normalizing constant that makes the belief state sum to 1. We can abbreviate this equation as $b' = \text{FORWARD}(b, a, o)$.

The fundamental insight required to understand POMDPs is this: *the optimal action depends only on the agent's current belief state.* That is, the optimal policy can be described by a mapping $\pi^*(b)$ from belief states to actions. It does *not* depend on the *actual* state the agent is in. This is a good thing, because the agent does not know its actual state; all it knows is the belief state. Hence, the decision cycle of a POMDP agent is this:

1. Given the current belief state b, execute the action $a = \pi^*(b)$.
2. Receive observation o.
3. Set the current belief state to $\text{FORWARD}(b, a, o)$ and repeat.

Now we can think of POMDPs as requiring a search in belief state space, just like the methods for sensorless and contingency problems in Chapter 3. The main difference is that the POMDP belief state space is *continuous*, because a POMDP belief state is a probability distribution. For example, a belief state for the 4×3 world is a point in an 11-dimensional continuous space. An action changes the belief state, not just the physical state, so it is evaluated according to the information the agent acquires as a result. POMDPs therefore include the value of information (Section 16.6) as one component of the decision problem.

Let's look more carefully at the outcome of actions. In particular, let's calculate the probability that an agent in belief state b reaches belief state b' after executing action a. Now, if we knew the action *and the subsequent observation*, then Equation (17.11) would provide a *deterministic* update to the belief state: $b' = \text{FORWARD}(b, a, o)$. Of course, the subsequent observation is not yet known, so the agent might arrive in one of several possible belief states b', depending on the observation that occurs. The probability of perceiving o, given that a was performed starting in belief state b, is given by summing over all the actual states s' that the agent might reach:

$$\begin{aligned} P(o|a, b) &= \sum_{s'} P(o|a, s', b) P(s'|a, b) \\ &= \sum_{s'} O(s', o) P(s'|a, b) \\ &= \sum_{s'} O(s', o) \sum_s T(s, a, s') b(s) \,. \end{aligned}$$

Let us write the probability of reaching b' from b, given action a, as $\tau(b, a, b')$. Then that gives us

$$\begin{aligned}\tau(b, a, b') &= P(b'|a, b) = \sum_o P(b'|o, a, b)P(o|a, b) \\ &= \sum_o P(b'|o, a, b) \sum_{s'} O(s', o) \sum_s T(s, a, s')b(s) \ , \end{aligned} \tag{17.12}$$

where $P(b'|o, a, b)$ is 1 if $b' = \text{FORWARD}(b, a, o)$ and 0 otherwise.

Equation (17.12) can be viewed as defining a transition model for the belief state space. We can also define a reward function for belief states (i.e., the expected reward for the actual states the agent might be in):

$$\rho(b) = \sum_s b(s)R(s) \ .$$

So it seems that $\tau(b, a, b')$ and $\rho(b)$ together define an *observable* MDP on the space of belief states. Furthermore, it can be shown that an optimal policy for this MDP, $\pi^*(b)$, is also an optimal policy for the original POMDP. In other words, *solving a POMDP on a physical state space can be reduced to solving an MDP on the corresponding belief state space.* This fact is perhaps less surprising if we remember that the belief state is always observable to the agent, by definition.

Notice that, although we have reduced POMDPs to MDPs, the MDP we obtain has a continuous (and usually high-dimensional) state space. None of the MDP algorithms described in Sections 17.2 and 17.3 applies directly to such MDPs. It turns out that we *can* develop versions of value and policy iteration that apply to continuous-state MDPs. The basic idea is that a policy $\pi(b)$ can be represented as a set of *regions* of belief state space, each of which is associated with a particular optimal action.[4] The value function associates a distinct *linear* function of b with each region. Each value- or policy-iteration step refines the boundaries of the regions and might introduce new regions.

The details of the algorithms are beyond the scope of this book, but we will report the solution for the sensorless 4×3 world. The optimal policy is the following:

$$[\mathit{Left}, \mathit{Up}, \mathit{Up}, \mathit{Right}, \mathit{Up}, \mathit{Up}, \mathit{Right}, \mathit{Up}, \mathit{Up}, \mathit{Right}, \mathit{Up}, \mathit{Right}, \mathit{Up}, \mathit{Right}, \mathit{Up}, \ldots] \ .$$

The policy is a sequence because this problem is *deterministic* in belief state space—there are no observations. The "trick" it embodies is to have the agent move *Left* once to ensure that it's *not* in (4,1), so that it's then fairly safe to keep moving *Up* and *Right* to reach the +1 exit. The agent reaches the +1 exit 86.6% of the time and does so much faster than the policy given earlier in the section, so its expected utility is 0.38 compared with 0.08.

For more complex POMDPs with observations, finding approximately optimal policies is very difficult (PSPACE-hard, in fact—i.e., very hard indeed). Problems with a few dozen states are often infeasible. The next section describes a different, approximate method for solving POMDPs, one based on look-ahead search.

[4] For some POMDPs, the optimal policy has infinitely many regions, so the simple list-of-regions approach fails and more ingenious methods are needed to find even an approximation.

17.5 DECISION-THEORETIC AGENTS

In this section, we outline a comprehensive approach to agent design for partially observable, stochastic environments. The basic elements of the design are already familiar:

- The transition and observation models are represented by a **dynamic Bayesian network** (as described in Chapter 15).
- The dynamic Bayesian network is extended with decision and utility nodes, as used in **decision networks** in Chapter 16. The resulting model is called a **dynamic decision network** or DDN.

DYNAMIC DECISION NETWORK

- A filtering algorithm is used to incorporate each new percept and action and to update the belief state representation.
- Decisions are made by projecting forward possible action sequences and choosing the best one.

The primary advantage of using a dynamic Bayesian network to represent the transition and sensor models is that it decomposes the state description into a set of random variables, much as planning algorithms use logical representations to decompose the state space used by search algorithms. The agent design is therefore a practical implementation of the **utility-based agent** sketched in Chapter 2.

Because we are using dynamic Bayesian networks, we will revert to the notation of Chapter 15, where $\mathbf{X}_t$ referred to the set of state variables for time t and $\mathbf{E}_t$ referred to the evidence variables. Thus, where we have used s_t (the state at time t) so far in this chapter, we will now use $\mathbf{X}_t$. We will use A_t to refer to the action at time t, so the transition model $T(s, a, s')$ is the same as $\mathbf{P}(\mathbf{X}_{t+1}|\mathbf{X}_t, A_t)$ and the observation model $O(s, o)$ is the same as $\mathbf{P}(\mathbf{E}_t|\mathbf{X}_t)$. We will use R_t to refer to the reward received at time t and U_t to refer to the utility of the state at time t. With this notation, a dynamic decision network looks like the one shown in Figure 17.9.

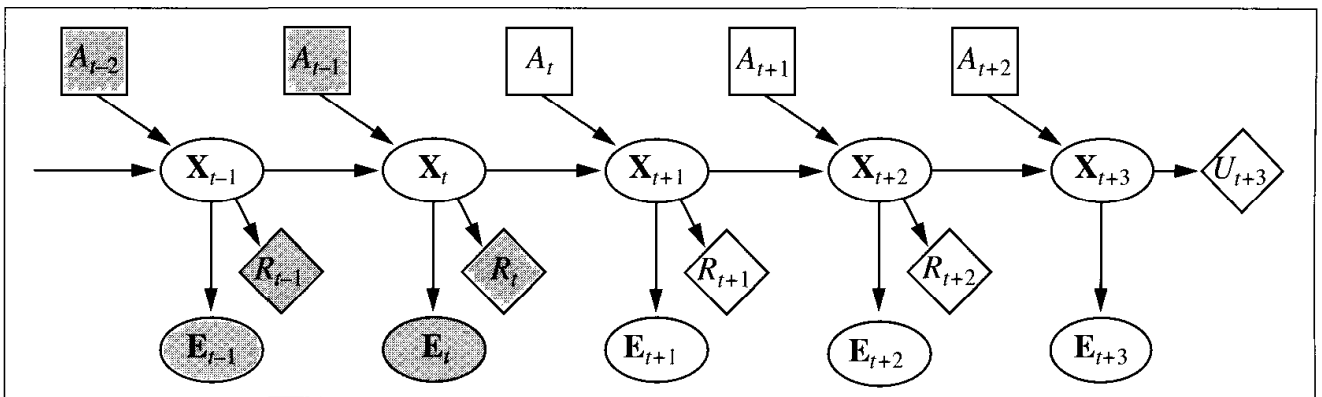


Figure 17.9 The generic structure of a dynamic decision network. Variables with known values are shaded. The current time is t and the agent must decide what to do—that is, choose a value for A_t. The network has been unrolled into the future for three steps and represents future rewards, as well as the utility of the state at the look-ahead horizon.

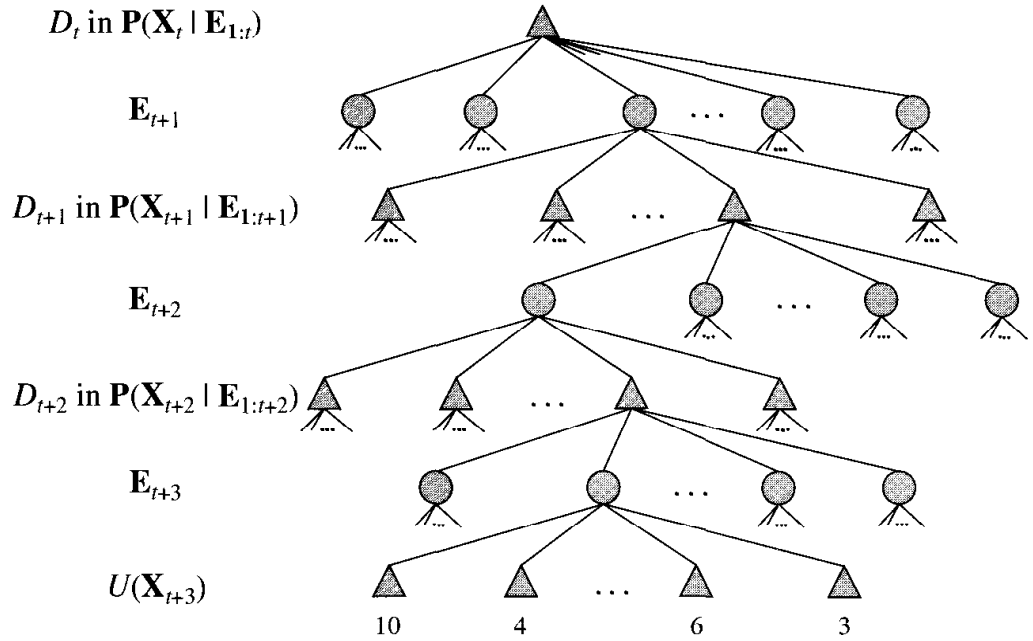


Figure 17.10 Part of the look-ahead solution of the DDN in Figure 17.9.

Dynamic decision networks provide a concise representation for large POMDPs, so they can be used as inputs for any POMDP algorithm including those for value- and policy-iteration methods. In this section, we will focus on look-ahead methods that project action sequences forward from the current belief state in much the same way as do the game-playing algorithms of Chapter 6. The network in Figure 17.9 has been projected three steps into the future; the current and future decisions and the future observations and rewards are all unknown. Notice that the network includes nodes for the *rewards* for $\mathbf{X}_{t+1}$ and $\mathbf{X}_{t+2}$, but the *utility* for $\mathbf{X}_{t+3}$. This is because the agent must maximize the (discounted) sum of all future rewards, and $U(\mathbf{X}_{t+3})$ represents the reward for $\mathbf{X}_{t+3}$ and all subsequent rewards. As in Chapter 6, we assume that U is available only in some approximate form: if exact utility values were available, there would be no need for look-ahead beyond depth 1.

Figure 17.10 shows part of the search tree corresponding to the three-step look-ahead DDN in Figure 17.9. Each of the triangular nodes is a belief state in which the agent makes a decision A_{t+i} for $i = 0, 1, 2, \ldots$. The round nodes correspond to choices by the environment, namely, what observation $\mathbf{E}_{t+i}$ occurs. Notice that there are no chance nodes corresponding to the action outcomes; this is because the belief state update for an action is deterministic regardless of the actual outcome.

The belief state at each triangular node can be computed by applying a filtering algorithm to the sequence of observations and actions leading to it. In this way, the algorithm takes into account the fact that, for decision A_{t+i}, the agent *will* have available percepts $\mathbf{E}_{t+1}, \ldots, \mathbf{E}_{t+i}$, even though at time t it does not know what those percepts will be. In this way, a decision-theoretic agent automatically takes into account the value of information and will execute information-gathering actions where appropriate.

A decision can be extracted from the search tree by backing up the utility values from the leaves, taking an average at the chance nodes and taking the maximum at the decision

nodes. This is similar to the EXPECTIMINIMAX algorithm for game trees with chance nodes, except that (1) there can also be rewards at non-leaf states and (2) the decision nodes correspond to belief states rather than actual states. The time complexity of an exhaustive search to depth d is $O(|D|^d \cdot |\mathbf{E}|^d)$, where $|D|$ is the number of available actions and $\mathbf{E}$ is the number of possible observations. For problems in which the discount factor γ is not too close to 1, a shallow search is often good enough to give near-optimal decisions. It is also possible to approximate the averaging step at the chance nodes, by sampling from the set of possible observations instead of summing over all possible observations. There are various other ways of finding good approximate solutions quickly, but we defer them to Chapter 21.

Decision-theoretic agents based on dynamic decision networks have a number of advantages compared with other, simpler agent designs presented in earlier chapters. In particular, they handle partially observable, uncertainty environments and can easily revise their "plans" to handle unexpected observations. With appropriate sensor models, they can handle sensor failure and can plan to gather information. They exhibit "graceful degradation" under time pressure and in complex environments, using various approximation techniques. So what is missing? The most important defect of our DDN-based algorithm is its reliance on forward search, just like the state-space search algorithms of Part II. In Part IV, we explained how the ability to consider partially ordered, abstract plans via goal-directed search provided a massive increase in problem-solving power, particularly when combined with plan libraries. There have been attempts to extend these methods into the probabilistic domain, but so far they have proven to be inefficient. A second, related problem is the basically propositional nature of the DDN language. We would like to be able to extend some of the ideas for first-order probabilistic languages in Section 14.6 to the problem of decision making. Current research has shown that this extension is possible and has significant benefits, as discussed in the notes at the end of the chapter.

17.6 DECISIONS WITH MULTIPLE AGENTS: GAME THEORY

This chapter has concentrated on making decisions in uncertain environments. But what if the uncertainty is due to other agents and the decisions they make? And what if the decisions of those agents are in turn influenced by our decisions? We addressed this question once before, when we studied games in Chapter 6. There, however, we were concerned with turn-taking games with perfect information, for which minimax search can be used to find optimal moves. In this section we study the aspects of **game theory** that can be used to analyze games with simultaneous moves. To simplify matters, we will look first at games that are only one move long. The word "game" and the simplification to single moves might make this seem trivial, but in fact, game theory is used in very serious decision making situations including bankruptcy proceedings, the auctioning of wireless frequency spectrums, product development and pricing decisions, and national defense, situations involving billions of dollars and hundreds of thousands of lives. Game theory can be used in at least two ways:

GAME THEORY

1. **Agent design**: Game theory can analyze the agent's decisions, and compute the expected utility for each decision (under the assumption that other agents are acting optimally according to game theory). For example, in the game **two-finger Morra**, two players, O and E, simultaneously display one or two fingers. Let the total number of fingers be f. If f is odd, O collects f dollars from E, and if f is even, E collects f dollars from O. Game theory can determine the best strategy against a rational player and the expected return for each player.[5]
2. **Mechanism Design**: When an environment is inhabited by many agents, it might be possible to define the rules of the environment (i.e., the game that the agents must play) so that the collective good of all agents is maximized when each agent adopts the game-theoretic solution that maximizes its own utility. For example, game theory can help design the protocols for a collection of Internet traffic routers so that each router has an incentive to act in such a way that global throughput is maximized. Mechanism design can also be used to construct intelligent **multiagent systems** that solve complex problems in a distributed fashion without the need for each agent to know about the whole problem being solved.

A game in game theory is defined by the following components:

PLAYERS
- **Players** or agents who will be making decisions. Two-player games have received the most attention, although n-player games for $n > 2$ are also common. We will give players capitalized names, like *Alice* and *Bob* or O and E.

ACTIONS
- **Actions** that the players can choose. We will give actions lowercase names, like *one* or *testify*. The players may or may not have the same set of actions available.

PAYOFF MATRIX
- A **payoff matrix** that gives the utility to each player for each combination of actions by all the players. The payoff matrix for two-finger Morra is as follows:

	O: one	*O: two*
E: one	$E = 2, O = -2$	$E = -3, O = 3$
E: two	$E = -3, O = 3$	$E = 4, O = -4$

For example, the lower-right corner shows that when O chooses action *two* and E also chooses *two*, the payoff is 4 for E and -4 for O.

Each player in a game must adopt and then execute a **strategy** (which is the name used in game theory for a policy). A **pure strategy** is a deterministic policy specifying a particular action to take in each situation; for a one-move game, a pure strategy is just a single action. The analysis of games leads to the idea of a **mixed strategy**, which is a randomized policy that selects particular actions according to a specific probability distribution over actions. The mixed strategy that chooses action a with probability p and action b otherwise is written $[p\colon a; (1-p)\colon b]$. For example, a mixed strategy for two-finger Morra might be $[0.5\colon \mathit{one}; 0.5\colon \mathit{two}]$. A **strategy profile** is an assignment of a strategy to each player; given the strategy profile, the game's **outcome** is a numeric value for each player.

PURE STRATEGY
MIXED STRATEGY
STRATEGY PROFILE
OUTCOME

[5] Morra is a recreational version of an **inspection game**. In such games, an inspector chooses a day to inspect a facility (such as a restaurant or a biological weapons plant), and the facility operator chooses a day to hide all the nasty stuff. The inspector wins if the days are different and the facility operator wins if they are the same.

SOLUTION

A **solution** to a game is a strategy profile in which each player adopts a rational strategy. We will see that the most important issue in game theory is to define what "rational" means when each agent chooses only part of the strategy profile that determines the outcome. It is important to realize that outcomes are actual results of playing a game, while solutions are theoretical constructs used to analyze a game. We will see that some games only have a solution in mixed strategies. But that does not mean that a player must literally be adopting a mixed strategy to be rational.

Consider the following story: Two alleged burglars, Alice and Bob, are caught red-handed near the scene of a burglary and are interrogated separately by the police. Both know that if they both confess to the crime, they will each serve 5 years in prison for burglary, but if both refuse to confess, they will serve only 1 year each for the lesser charge of possessing stolen property. However, the police separately offer each a deal: if you testify against your partner as the leader of a burglary ring, you'll go free, while the partner will serve 10 years.

PRISONER'S DILEMMA

Now Alice and Bob face the so-called **prisoner's dilemma**: should they testify or refuse? Being rational agents, Alice and Bob each want to maximize their own expected utility. Let's assume that Alice is callously unconcerned about her partner's fate, so her utility decreases in proportion to the number of years she will spend in prison, regardless of what happens to Bob. Bob feels exactly the same way. To help reach a rational decision, they both construct the following payoff matrix:

	Alice:testify	*Alice:refuse*
Bob:testify	$A = -5, B = -5$	$A = -10, B = 0$
Bob:refuse	$A = 0, B = -10$	$A = -1, B = -1$

Alice analyzes the payoff matrix as follows: Suppose Bob testifies. Then I get 5 years if I testify and 10 years if I don't, so in that case testifying is better. On the other hand, if Bob refuses, then I get 0 years if I testify and 1 year if I refuse, so in that case as well testifying is better. So in either case, it's better for me to testify, so that's what I must do.

DOMINANT STRATEGY

STRONGLY DOMINATES

WEAKLY DOMINATES

Alice has discovered that *testify* is a **dominant strategy** for the game. We say that a strategy s for player p **strongly dominates** strategy s' if the outcome for s is better for p than the outcome for s', for every choice of strategies by the other players. Strategy s **weakly dominates** s' if s is better than s' on at least one strategy profile and no worse on any other. A dominant strategy is a strategy that dominates all others. It is irrational to play a strongly dominated strategy, and irrational not to play a dominant strategy if one exists. Being rational, Alice chooses the dominant strategy. We need just a bit more terminology before we go on:

PARETO OPTIMAL

PARETO DOMINATED

we say that an outcome is **Pareto optimal**[6] if there is no other outcome that all players would prefer. An outcome is **Pareto dominated** by another outcome if all players would prefer the other outcome.

If Alice is clever as well as rational, she will continue to reason as follows: Bob's dominant strategy is also to testify. Therefore, he will testify and we will both get five years. When each player has a dominant strategy, the combination of those strategies is called a

DOMINANT STRATEGY EQUILIBRIUM

EQUILIBRIUM

dominant strategy equilibrium. In general, a strategy profile forms an **equilibrium** if no

[6] Pareto optimality is named after the economist Vilfredo Pareto (1848–1923).

player can benefit by switching strategies, given that every other player sticks with the same strategy. An equilibrium is essentially a **local optimum** in the space of policies; it is the top of a peak that slopes downward along every dimension, where a dimension corresponds to a player's strategy choices.

The *dilemma* in the prisoner's dilemma is that the outcome of the equilibrium point is worse for both players than the outcome they would get if they both refused to testify. In other words, the outcome for the equilibrium solution is Pareto dominated by the (-1, -1) outcome of $(refuse, refuse)$.

Is there any way for Alice and Bob to arrive at the (-1, -1) outcome? It is certainly an *allowable* option for both of them to refuse to testify, but is is an *unlikely* option. Either player contemplating playing $refuse$ will realize that he or she would do better by playing $testify$. That is the attractive power of an equilibrium point.

NASH EQUILIBRIUM

The mathematician John Nash (1928–) proved that *every game has an equilibrium of the type defined here.* It is now called a **Nash equilibrium** in his honor. Clearly, a dominant strategy equilibrium is a Nash equilibrium (Exercise 17.9), but not all games have dominant strategies. Nash's theorem means that there are equilibrium strategies even when there is no dominant strategy.

For the prisoner's dilemma, only the strategy profile $(testify, testify)$ is a Nash equilibrium. It is hard to see how rational players can avoid this outcome, because in any proposed non-equilibrium solution at least one of the players will be tempted to change strategies. Game theorists agree that being a Nash equilibrium is a necessary condition for being a solution—although they disagree whether it is a sufficient condition.

It is easy enough to avoid the $(testify, testify)$ solution if we change the game (or the players) in some way. For example, we could change to an iterated game in which the players know that they will meet again (but crucially, they must be uncertain about how many times they will meet again). Or if the agents have moral beliefs that encourage cooperation and fairness, we could change the payoff matrix to reflect the utility to each agent of cooperating with the other. We will see later that changing the agents to have limited computational powers, rather than the ability to reason absolutely rationally, can also affect the outcome, as can telling one agent that the other has limited rationality.

Now, let's look at a game that has no dominant strategy. Acme, a video game hardware manufacturer, has to decide whether its next game machine will use DVDs or CDs. Meanwhile, the video game software producer Best needs to decide whether to produce its next game on DVD or CD. The profits for both will be positive if they agree and negative if they disagree, as shown in the following payoff matrix:

	$Acme{:}dvd$	$Acme{:}cd$
$Best{:}dvd$	$A = 9, B = 9$	$A = -4, B = -1$
$Best{:}cd$	$A = -3, B = -1$	$A = 5, B = 5$

There is no dominant strategy equilibrium for this game, but there are *two* Nash equilibria: (*dvd, dvd*) and (*cd, cd*). We know these are Nash equilibria because, if either player unilaterally moves to a different strategy, that player will be worse off. Now the agents have a problem: *there are multiple acceptable solutions, but if each agent chooses a different*

solution then the resulting strategy profile won't be a solution at all and both agents will suffer. How can they agree on a solution? One answer is that both should choose the Pareto-optimal solution (*dvd, dvd*); that is, we can restrict the definition of "solution" to the unique Pareto-optimal Nash equilibrium *provided that one exists*. Every game has at least one Pareto-optimal solution, but a game might have several, or they might not be equilibrium points. For example, we could set the payoffs for (*dvd, dvd*) to 5 instead of 9. In that case, there are two equal Pareto-optimal equilibrium points. To choose between them the agents can either guess or *communicate*, which can be done either by establishing a convention that orders the solutions before the game begins or by negotiating to reach a mutually beneficial solution during the game (which would mean including communicative actions as part of a multimove game). Communication thus arises in game theory for exactly the same reasons that it arose in multiagent planning in Chapter 12. Games in which players need to communicate like this are called **coordination games**.

COORDINATION GAME

We have seen that a game can have more than one Nash equilibrium; how do we know that every game must have at least one? It can be that a game has no *pure-strategy* Nash equilibria. Consider, for example, any pure strategy profile for two-finger Morra (page 632). If the total number of fingers is even then O will want to switch; if the total is odd then E will want to switch. Therefore no pure strategy profile can be an equilibrium and we must look to mixed strategies.

But *which* mixed strategy? In 1928, von Neumann developed a method for finding the *optimal* mixed strategy for two-player, **zero-sum games**. A zero-sum game is a game in which the payoffs in each cell of the payoff matrix sum to zero.[7] Clearly, Morra is such a game. For two-player, zero-sum games, we know that the payoffs are equal and opposite, so we need consider the payoffs of only one player, who will be the maximizer (just as in Chapter 6). For Morra, we pick the even player E to be the maximizer, so we can define the payoff matrix by the values $U_E(e, o)$—the payoff to E if E does e and O does o.

ZERO-SUM GAME

MAXIMIN

Von Neumann's method is called the the **maximin** technique, and it works as follows:

- Suppose we change the rules to force E to reveal his or her strategy first, followed by O. Then we have a turn-taking game to which we can apply the standard **minimax** algorithm from Chapter 6. Let's suppose this gives an outcome $U_{E,O}$. Clearly, this game favors O, so the true utility U of the game (from E's point of view) is *at least* $U_{E,O}$. For example, if we just look at pure strategies, the minimax game tree has a root value of -3 (see Figure 17.11(a)), so we know that $U \geq -3$.
- Now suppose we change the rules to force O to reveal his or her strategy first, followed by E. Then the minimax value of this game is $U_{O,E}$, and because this game favors E we know that U is *at most* $U_{O,E}$. With pure strategies, the value is $+2$ (see Figure 17.11(b)), so we know $U \leq +2$.

[7] More general is the concept of **constant-sum games**, in which the sum of every cell in the game adds up to a constant, c. An n-person constant-sum game can be turned into a zero-sum game by subtracting c/n from every payoff. Thus chess, with traditional payoff of 1 for a win, 1/2 for a draw, and 0 for a loss, is technically a constant-sum game with $c = 1$, but can easily be transformed into a zero-sum game by subtracting 1/2 from every payoff.

Combining these two arguments, we see that the true utility U of the solution must satisfy

$$U_{E,O} \leq U \leq U_{O,E} \qquad \text{or in this case,} \qquad -3 \leq U \leq 2\,.$$

To pinpoint the value of U, we need to turn our analysis to mixed strategies. First, observe the following: *once the first player has revealed his or her strategy, the second player cannot lose by playing a pure strategy.* The reason is simple: if the second player plays a mixed strategy, $[p\text{: } one; (1-p)\text{: } two]$, its expected utility is a linear combination $(p \cdot u_{one} + (1-p) \cdot u_{two})$ of the utilities of the pure strategies, u_{one} and u_{two}. This linear combination can never be better than the best of u_{one} and u_{two}, so the second player might as well play a pure strategy.

With this observation in mind, the minimax trees can be thought of as having infinitely many branches at the root, corresponding to the infinitely many mixed strategies the first player can choose. Each of these leads to a node with two branches corresponding to the pure strategies for the second player. We can depict these infinite trees finitely by having one "parameterized" choice at the root:

- If E moves first, the situation is as shown in Figure 17.11(c). E plays $[p\text{: } one; (1-p)\text{: } two]$ at the root, and then O chooses a move given the value of p. If O chooses *one*, the expected payoff (to E) is $2p - 3(1-p) = 5p - 3$; if O chooses *two*, the expected payoff is $-3p + 4(1-p) = 4 - 7p$. We can draw these two payoffs as straight lines on a graph, where p ranges from 0 to 1 on the x-axis, as shown in Figure 17.11(e). O, the minimizer, will always choose the lower of the two lines, as shown by the heavy lines in the figure. Therefore, the best that E can do at the root is to choose p to be at the intersection point, which is where

  $$5p - 3 = 4 - 7p \qquad \Rightarrow \qquad p = 7/12\,.$$

  The utility for E at this point is $U_{E,O} = -1/12$.
- If O moves first, the situation is as shown in Figure 17.11(d). O plays $[q\text{: } one; (1-q)\text{: } two]$ at the root, and then E chooses a move given the value of q. The payoffs are $2q - 3(1-q) = 5q - 3$ and $-3q + 4(1-q) = 4 - 7q$.[8] Again, Figure 17.11(f) shows that the best O can do at the root is to choose the intersection point:

  $$5q - 3 = 4 - 7q \qquad \Rightarrow \qquad q = 7/12\,.$$

  The utility for E at this point is $U_{O,E} = -1/12$.

Now we know that the true utility of the game lies between $-1/12$ and $-1/12$, that is, it is exactly $-1/12$! (The moral is that it is better to be O than E if you are playing this game.) Furthermore, the true utility is attained by the mixed strategy $[7/12\text{: } one; 5/12\text{: } two]$, which should be played by both players. This strategy is called the **maximin equilibrium** of the game, and is a Nash equilibrium. Note that each component strategy in an equilibrium mixed strategy has the same expected utility. In this case, both *one* and *two* have the same expected utility, $-1/12$, as the mixed strategy itself.

MAXIMIN EQUILIBRIUM

Our result for two-finger Morra is an example of the general result by von Neumann: *every two-player zero-sum game has a maximin equilibrium when you allow mixed strategies.*

[8] It is a coincidence that these equations are the same as those for p; the coincidence arises because $U_E(one, two) = U_E(two, one) = -3$. This also explains why the optimal strategy is the same for both players.

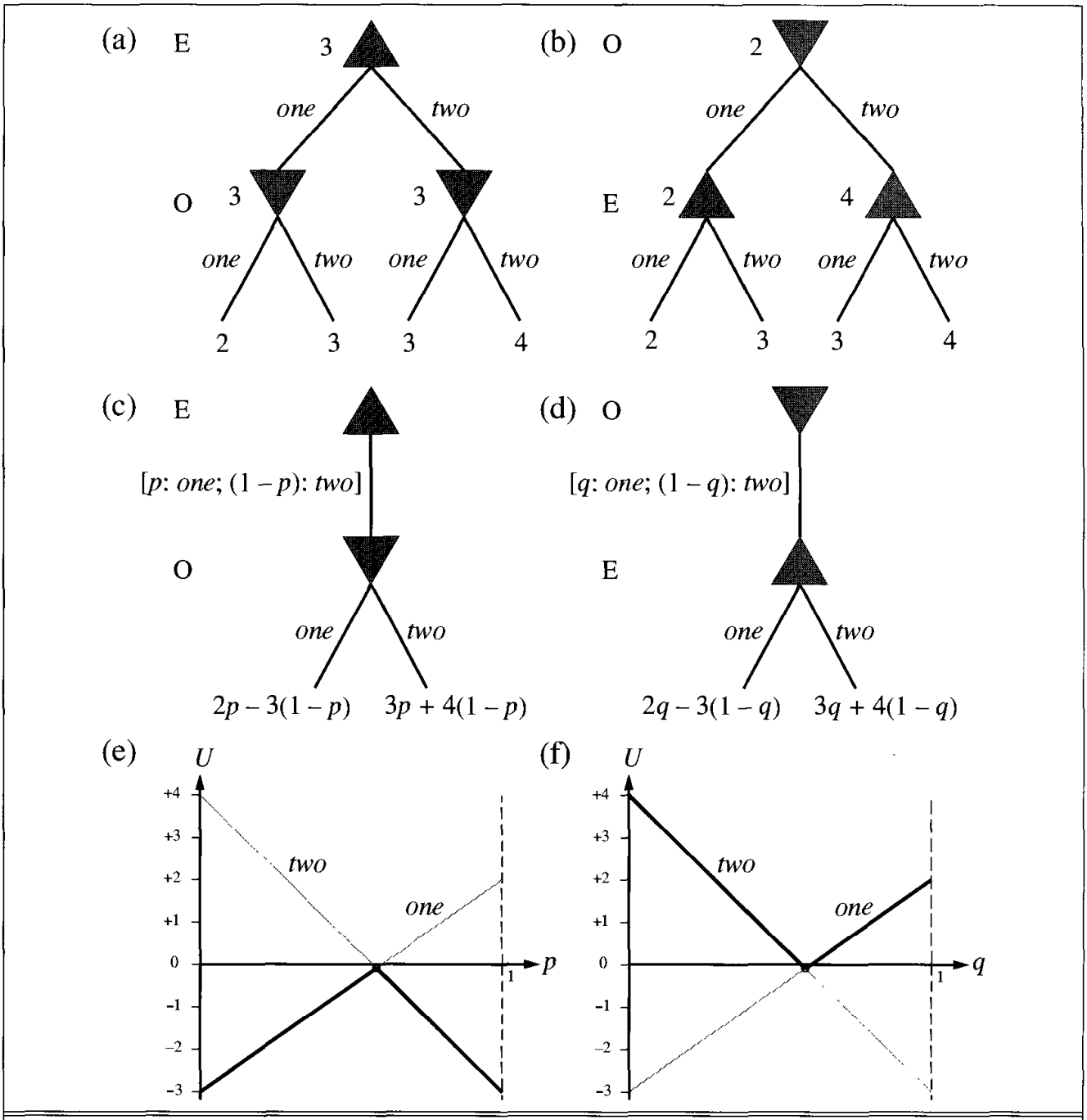


Figure 17.11 (a) and (b): Minimax game trees for two-finger Morra if the players take turns playing pure strategies. (c) and (d): Parameterized game trees where the first player plays a mixed strategy. The payoffs depend on the probability parameter (p or q) in the mixed strategy. (e) and (f): For any particular value of the probability parameter, the second player will choose the "better" of the two actions, so the value of the first player's mixed strategy is given by the heavy lines. The first player will choose the probability parameter for the mixed strategy at the intersection point.

Furthermore, every Nash equilibrium in a zero-sum game is a maximin for both players. The general algorithm for finding maximin equilibria in zero-sum games is somewhat more involved than Figures 17.11(e) and (f) might suggest. When there are n possible actions, a mixed strategy is a point in n-dimensional space and the lines become hyperplanes. It's

also possible for some pure strategies for the second player to be dominated by others, so that they are not optimal against *any* strategy for the first player. After removing all such strategies (which might have to be done repeatedly), the optimal choice at the root is the highest (or lowest) intersection point of the remaining hyperplanes. Finding this choice is an example of a **linear programming** problem: maximizing an objective function subject to linear constraints. Such problems can be solved by standard techniques in time polynomial in the number of actions (and in the number of bits used to specify the reward function, if you want to get technical).

The question remains, what should a rational agent actually *do* in playing a single game of Morra? The rational agent will have derived the fact that $[7/12\colon one; 5/12\colon two]$ is the maximin equilibrium strategy, and will assume that this is mutual knowledge with a rational opponent. The agent could use a 12-sided die or a random number generator to pick randomly according to this mixed strategy, in which case the expected payoff would be -1/12 for E. Or the agent could just decide to play *one*, or *two*. In either case, the expected payoff remains -1/12 for E. Curiously, unilaterally choosing a particular action does not harm one's expected payoff, but allowing the other agent to know that one has made such a unilateral decision *does* affect the expected payoff, because then the opponent can adjust his strategy accordingly.

Finding solutions to non-zero-sum finite games (i.e., Nash equilibria) is somewhat more complicated. The general approach has two steps: (1) Enumerate all possible subsets of actions that might form mixed strategies. For example, first try all strategy profiles where each player uses a single action, then those where each player uses either one or two actions, and so on. This is exponential in the number of actions, and so only applies to relatively small games. (2) For each strategy profile enumerated in (1), check to see if it is an equilibrium. This is done by solving a set of equations and inequalities that are similar to the ones used in the zero-sum case. For two players these equations are linear and can be solved with basic linear programming techniques, but for three or more players they are nonlinear and may be very difficult to solve.

REPEATED GAME

So far we have looked only at games that last a single move. The simplest kind of multiple-move game is the **repeated game**, in which players face the same choice repeatedly, but each time with knowledge of the history of all players' previous choices. A strategy profile for a repeated game specifies an action choice for each player at each time step for every possible history of previous choices. As with MDPs, payoffs are additive over time.

Let's consider the repeated version of the prisoner's dilemma. Will Alice and Bob work together and refuse to testify, knowing that they will meet again? The answer depends on the details of the engagement. For example, suppose Alice and Bob know that they must play exactly 100 rounds of prisoner's dilemma. Then they both know that the 100th round will not be a repeated game—that is, its outcome can have no effect on future rounds—and therefore they will both choose the dominant strategy, *testify*, in that round. But once the 100th round is determined, the 99th round can have no effect on subsequent rounds, so it too will have a dominant strategy equilibrium at (*testify*, *testify*). By induction, both players will choose *testify* on every round, earning a total jail sentence of 500 years each.

We can get different solutions by changing the rules of the interaction. For example, suppose that after each round there is a 99% chance that the players will meet again. Then

the expected number of rounds is still 100, but neither player knows for sure which round will be the last. Under these conditions, more cooperative behavior is possible. For example, one equilibrium strategy is for each player to *refuse* unless the other player has ever played *testify*. This strategy could be called **perpetual punishment**. Suppose both players have adopted this strategy, and this is mutual knowledge. Then as long as neither player has played *testify*, then at any point in time the expected future total payoff for each player is

PERPETUAL PUNISHMENT

$$\sum_{t=0}^{\infty} 0.99^t \cdot (-1) = -100 .$$

A player who chooses *testify* will gain a score of 0 rather than -1 on the very next move, but his or her total expected future payoff becomes

$$0 + \sum_{t=1}^{\infty} 0.99^t \cdot (-10) = -99 .$$

Therefore, at every step, there is no incentive to deviate from $(refuse, refuse)$. Perpetual punishment is the "mutually assured destruction" strategy of the prisoner's dilemma: once either player decides to *testify*, it assures that both players suffer a great deal. But it only works as a deterrent if the other player believes you have adopted this strategy—or at least that you might have adopted it.

There are other strategies that are more forgiving. The most famous, called **tit-for-tat**, calls for starting with *refuse* and then echoing the other player's previous move on all subsequent moves. So Alice would refuse as long as Bob refuses and would testify the move after Bob testified, but would go back to refusing if Bob did. Although very simple, this strategy has proven to be highly robust and effective against a wide variety of strategies.

TIT-FOR-TAT

We can also get different solutions by changing the agents, rather than changing the rules of engagement. Suppose the agents are finite-state machines with n states and they are playing a game with $m > n$ total steps. The agents are thus incapable of representing the number of remaining steps, and must treat it as an unknown. Therefore they cannot do the induction, and are free to arrive at the more favorable (*refuse, refuse*) equilibrium. In this case, ignorance *is* bliss—or rather, having your opponent believe that you are ignorant is bliss. Your success in these repeated games depends on the other player's *perception* of you as a bully or a simpleton, and not on your actual characteristics.

Repeated games in full generality are beyond the scope of this book, but they arise in many settings. For example, we can construct a sequential game by putting two agents in the 4×3 world of Figure 17.1. If we specify that no movement occurs when the two agents try to move into the same square simultaneously (a common problem at many traffic intersections), then certain pure strategies can get stuck forever. One solution is for each agent to randomize its choice between moving forward and staying put; the stalemate will be resolved quickly and both agents will be happy. This is exactly what is done to resolve packet collisions in Ethernet networks.

Currently known solution methods for repeated games resemble those for turn-taking games in Chapter 6, in that a game tree can be constructed from the root downwards and solved from the leaves upwards. The main difference is that, instead of simply taking the

maximum or minimum of the child values, the algorithm must solve a game in mixed strategies at each level, assuming that the child nodes have been solved and have well-defined values to work with.

PARTIAL INFORMATION

Repeated games in *partially observable* environments are called games of **partial information**. Examples include card games such as poker and bridge, wherein each player can see only a subset of the cards, and more serious "games" such as abstractions of nuclear war, where neither side knows the location of all its opponent's weapons. Games of partial information are solved by considering a tree of belief states, as in POMDPs. (See Section 17.4.) One important difference is that, while one's own belief state is observable, the opponent's belief state is not. Only recently have practical algorithms been developed for such games. Some simplified versions of poker have been solved, proving that bluffing is indeed a rational choice, as part of a well balanced mixed strategy. One important insight to emerge from such studies is that mixed strategies are useful not just for making one's actions unpredictable, but also for minimizing the amount of information that one's opponent can learn from observing one's actions. It is interesting that, although designers of programs for playing bridge are well aware of the importance of gathering and hiding information, none has yet proposed the use of randomized strategies.

BAYES-NASH EQUILIBRIUM

So far, there have been some barriers that have prevented game theory from being widely used in agent design. First, note that in a Nash equilibrium solution, a player is assuming that the opponents will definitely play the equilibrium strategy. This means that the player is unable to incorporate any beliefs it might have about how the other players are likely to act, and therefore that it might be wasting some of its value defending against threats that will never materialize. The notion of a **Bayes–Nash equilibrium** partially addresses this point: it is an equilibrium with respect to a player's prior probability distribution over the other players' strategies—in other words, it expresses a player's beliefs about the other players' likely strategies. Second, there is currently no good way to combine game theoretic and POMDP control strategies. Because of these and other problems, game theory has been used primarily to *analyze* environments that are at equilibrium, rather than to *control* agents within an environment. We shall soon see how it can help *design* environments.

17.7 Mechanism Design

MECHANISM DESIGN

In the previous section, we looked at the question "Given a game, what is a rational strategy?" In this section, we ask "Given that agents are rational, what game should we design?" More specifically, we would like to design a game whose solutions, consisting of each agent pursuing its own rational strategy, result in the maximization of some global utility function. This problem is called **mechanism design**, or sometimes **inverse game theory**. Mechanism design is a staple of economics and political science. For collections of agents, it holds the possibility of using game-theoretic mechanisms to construct smart systems out of a collection of more limited systems—even noncooperative systems—in much the same way that teams of humans can achieve goals far beyond the reach of any individual.

Examples of mechanism design include auctioning off cheap airline tickets, routing TCP packets between computers, deciding how medical interns will be assigned to hospitals, and deciding how robotic soccer players will cooperate with their teammates. Mechanism design became more than an academic subject in the 1990s when several nations, faced with the problem of auctioning off licenses to broadcast in various frequency bands, lost hundreds of millions of dollars in potential revenue as a result of poor mechanism design. Formally, a

MECHANISM

mechanism consists of (1) a language for describing the (possibly infinite) set of allowable strategies that agents may adopt and (2) an outcome rule G that determines the payoffs to the agents given a strategy profile of allowable strategies.

At first sight, the mechanism design problem can seem trivial. Suppose that the global utility function U is decomposed into any set of individual agent utility functions U_i, such that $U = \sum_i U_i$. Then, one might say, if each agent maximizes its own utility, surely that will lead automatically to the maximization of the global utility. (For example, Capitalism 101 says that if everyone tries to get rich, the total wealth of society will increase.) Unfortunately, this doesn't work. The actions of each agent could affect the well-being of other agents in

TRAGEDY OF THE COMMONS

ways that decrease global utility. One example of this is the **tragedy of the commons**, a situation in which individual farmers all bring their livestock to graze for free on the town commons, with the result being the destruction of the commons and a negative utility for all the farmers. Each farmer individually acted rationally, reasoning that the use of the commons was free and that, although using the commons could lead to its destruction, refraining from using it would not help (because the others would use it anyway). Similar arguments apply to the use of the atmosphere and the oceans for free dumping of pollutants.

A standard approach in mechanism design for dealing with such problems is to charge

EXTERNALITIES

each agent for using the commons. More generally, we need to ensure that all **externalities**—effects on global utility that are not recognized in the individual agents' transactions—are made explicit. Setting the prices correctly is the difficult part. In the limit, this approach amounts to creating a mechanism in which each agent is effectively required to maximize global utility. This is an impossibly difficult task for the agent, who can neither assess the current state of the world nor observe the effects of its actions on all other agents. Mechanism design therefore concentrates on finding mechanisms for which the decision problem for the individual agents is straightforward.

Let's consider auctions first. In the most common form, an auction is a mechanism for selling some goods to members of a pool of bidders. The strategies are the bids and the outcome determines who gets the goods and how much they pay. One example of where auctions can come into play within AI is when a collection of agents are deciding whether to cooperate on a joint plan. Hunsberger and Grosz (2000) show that this can be accomplished efficiently with an auction in which the agents bid for roles in the joint plan.

For now, we'll consider auctions wherein (1) there is a single good, (2) each bidder has a utility value v_i for the good, and (3) these values are known only to the bidder. Bidders make bids b_i, and the highest bid wins the goods, but the mechanism determines how the bids are made and the price paid by the winner (it need not be b_i). The best-known type of

ENGLISH AUCTION

auction is the **English auction**, in which the auctioneer increments the price of the goods, checking whether bidders are still interested, until only one bidder is left. This mechanism

has the property that the bidder with the highest value v_i gets the goods at a price of $b_m + d$, where b_m is the highest bid among all the other players and d is the auctioneer's increment between bids.[9] The English auction also has the property that bidders have a simple dominant strategy: keep bidding as long as the current cost is below your personal value. Recall that "dominant" means that the strategy works against all other strategies, which in turn means that a player can adopt it without regard for the other strategies. Therefore, players don't have to waste time and energy contemplating other players' possible strategies. A mechanism where players have a dominant strategy that involves revealing their true incentives is called a **strategy-proof** mechanism.

STRATEGY-PROOF

One negative property of the English auction is its high communication costs, so either the auction takes place in one room or all bidders have to have high-speed, secure communication lines. An alternative mechanism that requires much less communication is the **sealed bid auction**. Here, each bidder makes a single bid and communicates it to the auctioneer, and the highest bid wins. With this mechanism, the strategy of bidding your true value is no longer dominant. If your value is v_i and you believe that the maximum of all the other players' bids will be b_m, then you should bid the lower of v_i and $b_m + \epsilon$. Two drawbacks of the sealed bid auction are that the player with the highest v_i might not get the goods and that players must spend effort contemplating the other players' strategies.

SEALED BID AUCTION

A small change in the rules for sealed bid auctions produces the **sealed bid second-price auction**, also known as a **Vickrey auction**.[10] In such auctions, the winner pays the price of the *second* highest bid, rather than paying his own bid. This simple modification completely eliminates the complex deliberations required for standard (or **first-price**) sealed bid auctions, because the dominant strategy is now to bid your actual value. To see that, we note that any player can think of the auction as a two-player game, ignoring all players except himself and the highest bidder among the other players. The utility of player i in terms of his bid b_i, his value v_i, and the best bid among the other players, b_m, is

SEALED BID SECOND-PRICE AUCTION

VICKREY AUCTION

$$u_i(b_i, b_m) = \begin{cases} (v_i - b_m) & \text{if } b_i > b_m \\ 0 & \text{otherwise.} \end{cases}$$

To see that $b_i = v_i$ is a dominant strategy, note that when $(v_i - b_m)$ is positive, any bid that wins the auction is optimal, and bidding v_i in particular wins the auction. On the other hand, when $(v_i - b_m)$ is negative, any bid that loses the auction is optimal, and bidding v_i in particular loses the auction. So bidding v_i is optimal for all possible values of b_m, and in fact, v_i is the only bid that has this property. Because of its simplicity and the minimal computation requirements for both seller and bidders, the Vickrey auction is widely used in constructing distributed AI systems.

Now let's consider the Internet routing problem. The players correspond to edges in the graph of network connections. Each player knows the cost c_i of sending a message along its own edge; the cost of not having a message to send is 0. The goal is to find the cheapest path for a message to travel from origin to destination, where the cost of the whole route

[9] There is actually a small chance that the player with highest v_i fails to get the goods, in the case where $b_m < v_i < b_m + d$. The chance of this happening can be made arbitrarily small by decreasing the increment d.

[10] Named after William Vickrey (1914–1996), winner of the 1996 Nobel prize in economics.

is the sum of the individual edge costs. Chapter 4 gives several algorithms for computing the shortest path, given the edge costs, so all we have to do is get each agent to report its true cost, c_i. Unfortunately, if we just ask each agent, it will report costs that are high, to encourage us to send traffic elsewhere. We need to develop a strategy-proof mechanism. One such mechanism is to pay each player a payoff p_i equal to the length of the shortest path that does not contain the ith edge minus the length of the shortest path (as computed by a search algorithm) where the cost of the ith edge is assumed to be 0:

$$p_i = \text{LENGTH}(\text{path with } c_i = \infty) - \text{LENGTH}(\text{path with } c_i = 0)\ .$$

We can show that, under this mechanism, the dominant strategy for each player is to report c_i truthfully and that doing so will result in a cheapest path. Despite this desirable property, the mechanism outlined here is not used in practice, because of the high communication and central computation cost. The mechanism designer must communicate with all n players and then must solve the optimization problem. This might be worth it if the costs could be amortized over many messages, but in a real network the costs c_i would fluctuate constantly, because of traffic congestion and because some machines will crash and others will come online. No completely satisfactory solution has yet been devised.

17.8 SUMMARY

This chapter shows how to use knowledge about the world to make decisions even when the outcomes of an action are uncertain and the rewards for acting might not be reaped until many actions have passed. The main points are as follows:

- Sequential decision problems in uncertain environments, also called **Markov decision processes**, or MDPs, are defined by a **transition model** specifying the probabilistic outcomes of actions and a **reward function** specifying the reward in each state.
- The utility of a state sequence is the sum of all the rewards over the sequence, possibly discounted over time. The solution of an MDP is a **policy** that associates a decision with every state that the agent might reach. An optimal policy maximizes the utility of the state sequences encountered when it is executed.
- The utility of a state is the expected utility of the state sequences encountered when an optimal policy is executed, starting in that state. The **value iteration** algorithm for solving MDPs works by iteratively solving the equations relating the utilities of each state to that of its neighbors.
- **Policy iteration** alternates between calculating the utilities of states under the current policy and improving the current policy with respect to the current utilities.
- Partially observable MDPs, or POMDPs, are much more difficult to solve than are MDPs. They can be solved by conversion to an MDP in the continuous space of belief states. Optimal behavior in POMDPs includes information gathering to reduce uncertainty and therefore make better decisions in the future.

- A decision-theoretic agent can be constructed for POMDP environments. The agent uses a **dynamic decision network** to represent the transition and observation models, to update its belief state, and to project forward possible action sequences.
- **Game theory** describes rational behavior for agents in situations where multiple agents interact simultaneously. Solutions of games are **Nash equilibria**—strategy profiles in which no agent has an incentive to deviate from the specified strategy.
- **Mechanism design** can be used to set the rules by which agents will interact, in order to maximize some global utility through the operation of individually rational agents. Sometimes, mechanisms exist that achieve this goal without requiring each agent to consider the choices made by other agents.

We shall return to the world of MDPs and POMDP in Chapter 21, when we study **reinforcement learning** methods that allow an agent to improve its behavior from experience in sequential, uncertain environments.

BIBLIOGRAPHICAL AND HISTORICAL NOTES

Richard Bellman (1957) initiated the modern approach to sequential decision problems and proposed the dynamic programming approach in general and the value iteration algorithm in particular. Ron Howard's Ph.D. thesis (1960) introduced policy iteration and the idea of average reward for solving infinite-horizon problems. Several additional results were introduced by Bellman and Dreyfus (1962). Modified policy iteration is due to van Nunen (1976) and Puterman and Shin (1978). Asynchronous policy iteration was analyzed by Williams and Baird (1993), who also proved the policy loss bound in Equation (17.9). The analysis of discounting in terms of stationary preferences is due to Koopmans (1972). The texts by Bertsekas (1987), Puterman (1994), and Bertsekas and Tsitsiklis (1996) provide a rigorous introduction to sequential decision problems. Papadimitriou and Tsitsiklis (1987) describe results on the computational complexity of MDPs.

Seminal work by Sutton (1988) and Watkins (1989) on reinforcement learning methods for solving MDPs played a significant role in introducing MDPs into the AI community, as did the later survey by Barto *et al.* (1995). (Earlier work by Werbos (1977) contained many similar ideas, but was not taken up to the same extent.) The connection between MDPs and AI planning problems was made first by Sven Koenig (1991), who showed how probabilistic STRIPS operators provide a compact representation for transition models. (See also Wellman (1990b).) Work by Dean *et al.* (1993) and Tash and Russell (1994) attempted to overcome the combinatorics of large state spaces by using a limited search horizon and abstract states. Heuristics based on the value of information can be used to select areas of the state space where a local expansion of the horizon will yield a significant improvement in decision quality. Agents using this approach can tailor their effort to handle time pressure and generate some interesting behaviors such as using familiar "beaten paths" to find their way around the state space quickly without having to recompute optimal decisions at each point. Recent work by Boutilier and others (Boutilier *et al.*, 2000, 2001) has focused on dynamic program-

ming using *symbolic* representations of both transition models and value functions, based on propositional and first-order formulæ.

The observation that a partially observable MDP can be transformed into a regular MDP by using the belief states is due to Astrom (1965). The first complete algorithm for the exact solution of partially-observable Markov decision processes (POMDPs) was proposed by Edward Sondik (1971) in his Ph.D. thesis. (A later journal paper by Smallwood and Sondik (1973) contains some errors, but is more accessible.) Lovejoy (1991) surveys the state of the art in POMDPs. The first significant contribution within AI was the Witness algorithm (Cassandra *et al.*, 1994; Kaelbling *et al.*, 1998), an improved version of POMDP value iteration. Other algorithms soon followed, including an approach due to Hansen (1998) that constructs a policy incrementally in the form of a finite-state automaton. In this policy representation, the belief state corresponds directly to a particular state in the automaton. Approximately optimal policies for POMDPs can be constructed by forward search combined with sampling of possible observations and action outcomes (Kearns *et al.*, 2000; Ng and Jordan, 2000). Additional work on POMDP algorithms is covered in Chapter 21.

The basic ideas for an agent architecture using dynamic decision networks were proposed by Dean and Kanazawa (1989a). The book *Planning and Control* by Dean and Wellman (1991) goes into much greater depth, making connections between DBN/DDN models and the classical control literature on filtering. Tatman and Shachter (1990) showed how to apply dynamic programming algorithms to DDN models. Russell (1998) explains various ways in which such agents can be scaled up and identifies a number of open research issues.

The early roots of game theory can be traced back to proposals made in the 17th century by Christiaan Huygens and Gottfried Leibniz to study competitive and cooperative human interactions scientifically and mathematically. Throughout the 19th century, several leading economists created simple mathematical examples to analyze particular examples of competitive situations. The first formal results in game theory are due to Zermelo (1913) (who had, the year before, suggested a form of minimax search for games, albeit an incorrect one). Emile Borel (1921) introduced the notion of a mixed strategy. John von Neumann (1928) proved that every two-person, zero-sum game has a maximin equilibrium in mixed strategies and a well-defined value. Von Neumann's collaboration with the economist Oskar Morgenstern led to the publication in 1944 of the *Theory of Games and Economic Behavior*, the defining book for game theory. Publication of the book was delayed by the wartime paper shortage until a member of the Rockefeller family personally subsidized its publication.

In 1950, at the age of 21, John Nash published his ideas concerning equilibria in general games. His definition of an equilibrium solution, although originating in the work of Cournot (1838), became known as Nash equilibrium. After a long delay due to the schizophrenia he suffered from 1959 onwards, Nash was awarded the Nobel prize in Economics (along with Reinhart Selten and John Harsanyi) in 1994.

The Bayes–Nash equilibrium is described by Harsanyi (1967) and discussed by Kadane and Larkey (1982). Some issues in the use of game theory for agent control are covered by Binmore (1982).

The prisoner's dilemma was invented as a classroom exercise by Albert W. Tucker in 1950 and is covered extensively by Axelrod (1985). Repeated games were introduced by

Luce and Raiffa (1957) as were games of partial information by Kuhn (1953). The first practical algorithm for partial information games was developed within AI by Koller *et al.* (1996); the paper by Koller and Pfeffer (1997) provides a readable introduction to the general area and describes a working system for representing and solving sequential games. Game theory and MDPs are combined in the theory of Markov games (Littman, 1994). Shapley (1953) actually described the value iteration algorithm before Bellman, but his results were not widely appreciated, perhaps because they were presented in the context of Markov games. Textbooks on game theory include those by Myerson (1991), Fudenberg and Tirole (1991), and Osborne and Rubinstein (1994).

The tragedy of the commons, a motivating problem for the field of mechanism design, was presented by Hardin (1968). Hurwicz (1973) created a mathematical foundation for mechanism design. Milgrom (1997) writes about the multibillion-dollar spectrum auction mechanism he designed. Auctions can also be used in planning (Hunsberger and Grosz, 2000) and scheduling (Rassenti *et al.*, 1982). Varian (1995) gives a brief overview with connections to the computer science literature, and Rosenschein and Zlotkin (1994) present a book-length treatment with applications to distributed AI. Related work on distributed AI also goes under other names, including Collective Intelligence (Tumer and Wolpert, 2000) and market-based control (Clearwater, 1996). Papers on computational issues in auctions often appear in the ACM Conferences on Electronic Commerce.

EXERCISES

17.1 For the 4×3 world shown in Figure 17.1, calculate which squares can be reached from (1,1) by the action sequence $[Up, Up, Right, Right, Right]$ and with what probabilities. Explain how this computation is related to the task of projecting a hidden Markov model.

17.2 Suppose that we define the utility of a state sequence to be the *maximum* reward obtained in any state in the sequence. Show that this utility function does not result in stationary preferences between state sequences. Is it still possible to define a utility function on states such that MEU decision making gives optimal behavior?

17.3 Can any finite search problem be translated exactly into a Markov decision problem such that an optimal solution of the latter is also an optimal solution of the former? If so, explain *precisely* how to translate the problem and how to translate the solution back; if not, explain *precisely* why not (i.e., give a counterexample).

17.4 Consider an undiscounted MDP having three states, (1, 2, 3), with rewards -1, -2, 0 respectively. State 3 is a terminal state. In states 1 and 2 there are two possible actions: a and b. The transition model is as follows:

- In state 1, action a moves the agent to state 2 with probability 0.8 and makes the agent stay put with probability 0.2.
- In state 2, action a moves the agent to state 1 with probability 0.8 and makes the agent stay put with probability 0.2.

- In either state 1 or state 2, action b moves the agent to state 3 with probability 0.1 and makes the agent stay put with probability 0.9.

Answer the following questions:

a. What can be determined *qualitatively* about the optimal policy in states 1 and 2?

b. Apply policy iteration, showing each step in full, to determine the optimal policy and the values of states 1 and 2. Assume that the initial policy has action b in both states.

c. What happens to policy iteration if the initial policy has action a in both states? Does discounting help? Does the optimal policy depend on the discount factor?

17.5 Sometimes MDPs are formulated with a reward function $R(s, a)$ that depends on the action taken or a reward function $R(s, a, s')$ that also depends on the outcome state.

a. Write the Bellman equations for these formulations.

b. Show how an MDP with reward function $R(s, a, s')$ can be transformed into a different MDP with reward function $R(s, a)$, such that optimal policies in the new MDP correspond exactly to optimal policies in the original MDP.

c. Now do the same to convert MDPs with $R(s, a)$ into MDPs with $R(s)$.

17.6 Consider the 4×3 world shown in Figure 17.1.

a. Implement an environment simulator for this environment, such that the specific geography of the environment is easily altered. Some code for doing this is already in the online code repository.

b. Create an agent that uses policy iteration, and measure its performance in the environment simulator from various starting states. Perform several experiments from each starting state, and compare the average total reward received per run with the utility of the state, as determined by your algorithm.

c. Experiment with increasing the size of the environment. How does the runtime for policy iteration vary with the size of the environment?

17.7 For the environment shown in Figure 17.1, find all the threshold values for $R(s)$ such that the optimal policy changes when the threshold is crossed. You will need a way to calculate the optimal policy and its value for fixed $R(s)$. [*Hint*: Prove that the value of any fixed policy varies linearly with $R(s)$.]

17.8 In this exercise we will consider two-player MDPs that correspond to zero-sum, turn-taking games like those in Chapter 6. Let the players be A and B, and let $R(s)$ be the reward for player A in s. (The reward for B is always equal and opposite.)

a. Let $U_A(s)$ be the utility of state s when it is A's turn to move in s, and let $U_B(s)$ be the utility of state s when it is B's turn to move in s. All rewards and utilities are calculated from A's point of view (just as in a minimax game tree). Write down Bellman equations defining $U_A(s)$ and $U_B(s)$.

b. Explain how to do two-player value iteration with these equations, and define a suitable stopping criterion.

c. Consider the game described in Figure 6.14 on page 190. Draw the state space (rather than the game tree), showing the moves by A as solid lines and moves by B as dashed lines. Mark each state with $R(s)$. You will find it helpful to arrange the states (s_A, s_B) on a two-dimensional grid, using s_A and s_B as "coordinates."

d. Now apply two-player value iteration to solve this game, and derive the optimal policy.

17.9 Show that a dominant strategy equilibrium is a Nash equilibrium, but not *vice versa*.

17.10 In the children's game of rock–paper–scissors each player reveals at the same time a choice of rock, paper, or scissors. Paper wraps rock, rock blunts scissors, and scissors cut paper. In the extended version rock–paper–scissors–fire–water, fire beats rock, paper and scissors; rock, paper and scissors beat water, and water beats fire. Write out the payoff matrix and find a mixed-strategy solution to this game.

17.11 Solve the game of *three*-finger Morra.

17.12 Prior to 1999, teams in the National Hockey League received 2 points for a win, 1 for a tie, and 0 for a loss. Is this a constant-sum game? In 1999, the rules were amended so that a team receives 1 point for a loss in overtime. The winning team still gets 2 points. How does this modification change the answers to the questions above? If it were legal to do so, when would it be rational for the two teams to secretly agree to end regulation play in a tie and then battle it out in overtime? Assume that the utility to each team is the number of points it receives, and that there is a mutually known prior probability p that the first team will win in overtime. For what values of p would both teams agree to this arrangement?

17.13 The following payoff matrix, from Blinder (1983) by way of Bernstein (1996), shows a game between politicians and the Federal Reserve.

	Fed: contract	Fed: do nothing	Fed: expand
Pol: contract	$F = 7, P = 1$	$F = 9, P = 4$	$F = 6, P = 6$
Pol: do nothing	$F = 8, P = 2$	$F = 5, P = 5$	$F = 4, P = 9$
Pol: expand	$F = 3, P = 3$	$F = 2, P = 7$	$F = 1, P = 8$

Politicians can expand or contract fiscal policy, while the Fed can expand or contract monetary policy. (And of course either side can choose to do nothing.) Each side also has preferences for who should do what—neither side wants to look like the bad guys. The payoffs shown are simply the rank orderings: 9 for first choice through 1 for last choice. Find the Nash equilibrium of the game in pure strategies. Is this a Pareto optimal solution? The reader might wish to analyze the policies of recent administrations in this light.

18 LEARNING FROM OBSERVATIONS

In which we describe agents that can improve their behavior through diligent study of their own experiences.

The idea behind learning is that percepts should be used not only for acting, but also for improving the agent's ability to act in the future. Learning takes place as the agent observes its interactions with the world and its own decision-making processes. Learning can range from trivial memorization of experience, as exhibited by the wumpus-world agent in Chapter 10, to the creation of entire scientific theories, as exhibited by Albert Einstein. This chapter describes **inductive learning** from observations. In particular, we describe how to learn simple theories in propositional logic. We also give a theoretical analysis that explains why inductive learning works.

18.1 FORMS OF LEARNING

In Chapter 2, we saw that a learning agent can be thought of as containing a **performance element** that decides what actions to take and a **learning element** that modifies the performance element so that it makes better decisions. (See Figure 2.15.) Machine learning researchers have come up with a large variety of learning elements. To understand them, it will help to see how their design is affected by the context in which they will operate. The design of a learning element is affected by three major issues:

- Which *components* of the performance element are to be learned.
- What *feedback* is available to learn these components.
- What *representation* is used for the components.

We now analyze each of these issues in turn. We have seen that there are many ways to build the performance element of an agent. Chapter 2 described several agent designs (Figures 2.9, 2.11, 2.13, and 2.14). The components of these agents include the following:

1. A direct mapping from conditions on the current state to actions.
2. A means to infer relevant properties of the world from the percept sequence.

3. Information about the way the world evolves and about the results of possible actions the agent can take.
4. Utility information indicating the desirability of world states.
5. Action-value information indicating the desirability of actions.
6. Goals that describe classes of states whose achievement maximizes the agent's utility.

Each of these components can be learned from appropriate feedback. Consider, for example, an agent training to become a taxi driver. Every time the instructor shouts "Brake!" the agent can learn a condition–action rule for when to brake (component 1). By seeing many camera images that it is told contain buses, it can learn to recognize them (2). By trying actions and observing the results—for example, braking hard on a wet road—it can learn the effects of its actions (3). Then, when it receives no tip from passengers who have been thoroughly shaken up during the trip, it can learn a useful component of its overall utility function (4).

The *type of feedback* available for learning is usually the most important factor in determining the nature of the learning problem that the agent faces. The field of machine learning usually distinguishes three cases: **supervised**, **unsupervised**, and **reinforcement** learning.

SUPERVISED LEARNING

The problem of **supervised learning** involves learning a *function* from examples of its inputs and outputs. Cases (1), (2), and (3) are all instances of supervised learning problems. In (1), the agent learns condition–action rule for braking—this is a function from states to a Boolean output (to brake or not to brake), In (2), the agent learns a function from images to a Boolean output (whether the image contains a bus). In (3), the theory of braking is a function from states and braking actions to, say, stopping distance in feet. Notice that in cases (1) and (2), a teacher provided the correct output value of the examples; in the third, the output value was available directly from the agent's percepts. For fully observable environments, it will always be the case that an agent can observe the effects of its actions and hence can use supervised learning methods to learn to predict them. For partially observable environments, the problem is more difficult, because the immediate effects might be invisible.

UNSUPERVISED LEARNING

The problem of **unsupervised learning** involves learning patterns in the input when no specific output values are supplied. For example, a taxi agent might gradually develop a concept of "good traffic days" and "bad traffic days" without ever being given labelled examples of each. A purely unsupervised learning agent cannot learn what to do, because it has no information as to what constitutes a correct action or a desirable state. We will study unsupervised learning primarily in the context of probabilistic reasoning systems (Chapter 20.

REINFORCEMENT LEARNING

The problem of **reinforcement learning**, which we cover in Chapter 21, is the most general of the three categories. Rather than being told what to do by a teacher, a reinforcement learning agent must learn from **reinforcement**.[1] For example, the lack of a tip at the end of the journey (or a hefty bill for rear-ending the car in front) give the agent some indication that its behavior is undesirable. Reinforcement learning typically includes the subproblem of learning how the environment works.

REINFORCEMENT

The *representation of the learned information* also plays a very important role in determining how the learning algorithm must work. Any of the components of an agent can be represented using any of the representation schemes in this book. We have seen sev-

[1] The term **reward** as used in Chapter 17 is a synonym for **reinforcement**.

eral examples: linear weighted polynomials for utility functions in game-playing programs; propositional and first-order logical sentences for all of the components in a logical agent; and probabilistic descriptions such as Bayesian networks for the inferential components of a decision-theoretic agent. Effective learning algorithms have been devised for all of these. This chapter will cover methods for propositional logic, Chapter 19 describes methods for first-order logic, and Chapter 20 covers methods for Bayesian networks and for neural networks (which include linear polynomials as a special case).

The last major factor in the design of learning systems is the *availability of prior knowledge*. The majority of learning research in AI, computer science, and psychology has studied the case in which the agent begins with no knowledge at all about what it is trying to learn. It has access only to the examples presented by its experience. Although this is an important special case, it is by no means the general case. Most human learning takes place in the context of a good deal of background knowledge. Some psychologists and linguists claim that even newborn babies exhibit knowledge of the world. Whatever the truth of this claim, there is no doubt that prior knowledge can help enormously in learning. A physicist examining a stack of bubble-chamber photographs might be able to induce a theory positing the existence of a new particle of a certain mass and charge; but an art critic examining the same stack might learn nothing more than that the "artist" must be some sort of abstract expressionist. Chapter 19 shows several ways in which learning is helped by the use of existing knowledge; it also shows how knowledge can be *compiled* in order to speed up decision making. Chapter 20 shows how prior knowledge helps in the learning of probabilistic theories.

18.2 Inductive Learning

An algorithm for deterministic supervised learning is given as input the correct value of the unknown function for particular inputs and must try to recover the unknown function or something close to it. More formally, we say that an **example** is a pair $(x, f(x))$, where x is the input and $f(x)$ is the output of the function applied to x. The task of **pure inductive inference** (or **induction**) is this:

EXAMPLE

PURE INDUCTIVE INFERENCE

> Given a collection of examples of f, return a function h that approximates f.

The function h is called a **hypothesis**. The reason that learning is difficult, from a conceptual point of view, is that it is not easy to tell whether any particular h is a good approximation of f. A good hypothesis will **generalize** well—that is, will predict unseen examples correctly. This is the fundamental **problem of induction**. The problem has been studied for centuries; Section 18.5 provides a partial solution.

HYPOTHESIS

GENERALIZATION

PROBLEM OF INDUCTION

Figure 18.1 shows a familiar example: fitting a function of a single variable to some data points. The examples are $(x, f(x))$ pairs, where both x and $f(x)$ are real numbers. We choose the **hypothesis space H**—the set of hypotheses we will consider—to be the set of polynomials of degree at most k, such as $3x^2 + 2$, $x^{17} - 4x^3$, and so on. Figure 18.1(a) shows some data with an exact fit by a straight line (a polynomial of degree 1). The line is called a **consistent** hypothesis because it agrees with all the data. Figure 18.1(b) shows a

HYPOTHESIS SPACE

CONSISTENT

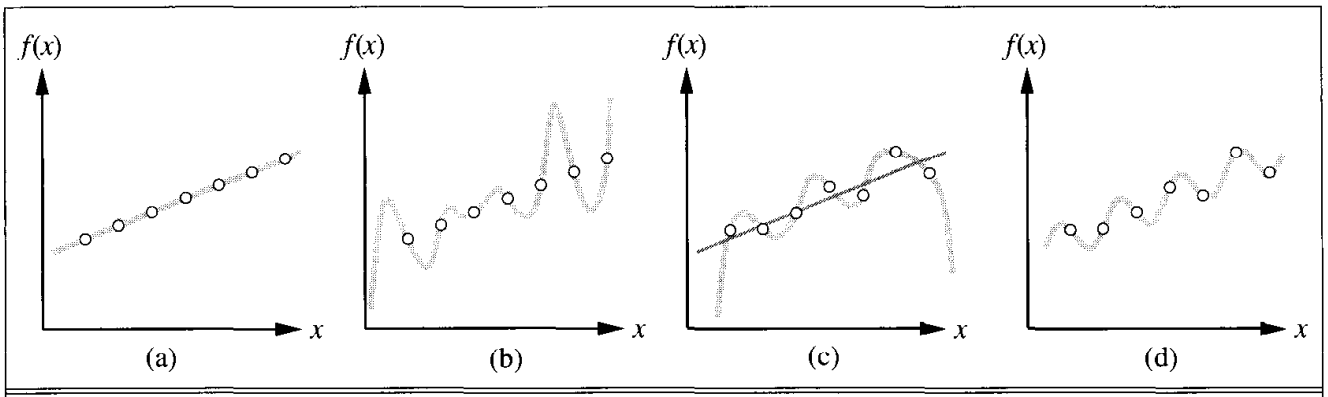


Figure 18.1 (a) Example $(x, f(x))$ pairs and a consistent, linear hypothesis. (b) A consistent, degree-7 polynomial hypothesis for the same data set. (c) A different data set that admits an exact degree-6 polynomial fit or an approximate linear fit. (d) A simple, exact sinusoidal fit to the same data set.

high-degree polynomial that is also consistent with the same data. This illustrates the first issue in inductive learning: *how do we choose from among multiple consistent hypotheses?* One answer is **Ockham's**[2] **razor**: prefer the *simplest* hypothesis consistent with the data. Intuitively, this makes sense, because hypotheses that are no simpler than the data themselves are failing to extract any *pattern* from the data. Defining simplicity is not easy, but it seems reasonable to say that a degree-1 polynomial is simpler than a degree-12 polynomial.

OCKHAM'S RAZOR

Figure 18.1(c) shows a second data set. There is no consistent straight line for this data set; in fact, it requires a degree-6 polynomial (with 7 parameters) for an exact fit. There are just 7 data points, so the polynomial has as many parameters as there are data points; thus, it does not seem to be finding any pattern in the data and we do not expect it to generalize well. It might be better to fit a simple straight line that is not exactly consistent but might make reasonable predictions. This amounts to accepting the possibility that the true function is not deterministic (or, roughly equivalently, that the true inputs are not fully observed). *For nondeterministic functions, there is an inevitable tradeoff between the complexity of the hypothesis and the degree of fit to the data.* Chapter 20 explains how to make this tradeoff using probability theory.

One should keep in mind that the possibility or impossibility of finding a simple, consistent hypothesis depends strongly on the hypothesis space chosen. Figure 18.1(d) shows that the data in (c) can be fit exactly by a simple function of the form $ax + b + c \sin x$. This example shows the importance of the choice of hypothesis space. A hypothesis space consisting of polynomials of finite degree cannot represent sinusoidal functions accurately, so a learner using that hypothesis space will not be able to learn from sinusoidal data. We say that a learning problem is **realizable** if the hypothesis space contains the true function; otherwise, it is **unrealizable**. Unfortunately, we cannot always tell whether a given learning problem is realizable, because the true function is not known. One way to get around this barrier is to use *prior knowledge* to derive a hypothesis space in which we know the true function must lie. This topic is covered in Chapter 19.

REALIZABLE

UNREALIZABLE

[2] Named after the 14th-century English philosopher, William of Ockham. The name is often misspelled as "Occam," perhaps from the French rendering, "Guillaume d'Occam."

Another approach is to use the largest possible hypothesis space. For example, why not let **H** be the class of all Turing machines? After all, every computable function can be represented by some Turing machine, and that is the best we can do. The problem with this idea is that it does not take into account the *computational complexity* of learning. *There is a tradeoff between the expressiveness of a hypothesis space and the complexity of finding simple, consistent hypotheses within that space.* For example, fitting straight lines to data is very easy; fitting high-degree polynomials is harder; and fitting Turing machines is very hard indeed because determining whether a given Turing machine is consistent with the data is not even decidable in general. A second reason to prefer simple hypothesis spaces is that the resulting hypotheses may be simpler to use—that is, it is faster to compute $h(x)$ when h is a linear function than when it is an arbitrary Turing machine program.

For these reasons, most work on learning has focused on relatively simple representations. In this chapter, we concentrate on propositional logic and related languages. Chapter 19 looks at learning theories in first-order logic. We will see that the expressiveness–complexity tradeoff is not as simple as it first seems: it is often the case, as we saw in Chapter 8, that an expressive language makes it possible for a *simple* theory to fit the data, whereas restricting the expressiveness of the language means that any consistent theory must be very complex. For example, the rules of chess can be written in a page or two of first-order logic, but require thousands of pages when written in propositional logic. In such cases, it should be possible to learn much faster by using the more expressive language.

18.3 Learning Decision Trees

Decision tree induction is one of the simplest, and yet most successful forms of learning algorithm. It serves as a good introduction to the area of inductive learning, and is easy to implement. We first describe the performance element, and then show how to learn it. Along the way, we will introduce ideas that appear in all areas of inductive learning.

Decision trees as performance elements

DECISION TREE
ATTRIBUTES
CLASSIFICATION
REGRESSION
POSITIVE
NEGATIVE

A **decision tree** takes as input an object or situation described by a set of **attributes** and returns a "decision"—the predicted output value for the input. The input attributes can be discrete or continuous. For now, we assume discrete inputs. The output value can also be discrete or continuous; learning a discrete-valued function is called **classification** learning; learning a continuous function is called **regression**. We will concentrate on *Boolean* classification, wherein each example is classified as true (**positive**) or false (**negative**).

A decision tree reaches its decision by performing a sequence of tests. Each internal node in the tree corresponds to a test of the value of one of the properties, and the branches from the node are labeled with the possible values of the test. Each leaf node in the tree specifies the value to be returned if that leaf is reached. The decision tree representation seems to be very natural for humans; indeed, many "How To" manuals (e.g., for car repair) are written entirely as a single decision tree stretching over hundreds of pages.

GOAL PREDICATE

A somewhat simpler example is provided by the problem of whether to wait for a table at a restaurant. The aim here is to learn a definition for the **goal predicate** *WillWait*. In setting this up as a learning problem, we first have to state what attributes are available to describe examples in the domain. In Chapter 19, we will see how to automate this task; for now, let's suppose we decide on the following list of attributes:

1. *Alternate*: whether there is a suitable alternative restaurant nearby.
2. *Bar*: whether the restaurant has a comfortable bar area to wait in.
3. *Fri/Sat*: true on Fridays and Saturdays.
4. *Hungry*: whether we are hungry.
5. *Patrons*: how many people are in the restaurant (values are *None*, *Some*, and *Full*).
6. *Price*: the restaurant's price range ($, $$, $$$).
7. *Raining*: whether it is raining outside.
8. *Reservation*: whether we made a reservation.
9. *Type*: the kind of restaurant (French, Italian, Thai, or burger).
10. *WaitEstimate*: the wait estimated by the host (0–10 minutes, 10–30, 30–60, >60).

The decision tree usually used by one of us (SR) for this domain is shown in Figure 18.2. Notice that the tree does not use the *Price* and *Type* attributes, in effect considering them to be irrelevant. Examples are processed by the tree starting at the root and following the appropriate branch until a leaf is reached. For instance, an example with *Patrons* = *Full* and *WaitEstimate* = 0–10 will be classified as positive (i.e., yes, we will wait for a table).

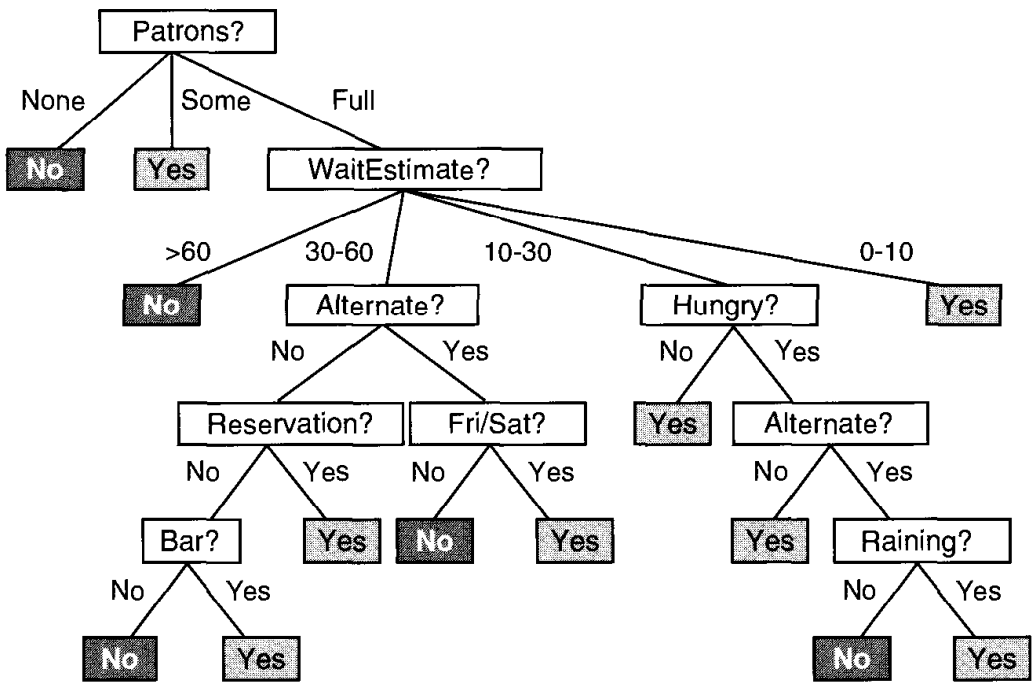


Figure 18.2 A decision tree for deciding whether to wait for a table.

Expressiveness of decision trees

Logically speaking, any particular decision tree hypothesis for the *WillWait* goal predicate can be seen as an assertion of the form

$$\forall s \;\; WillWait(s) \;\Leftrightarrow\; (P_1(s) \vee P_2(s) \vee \cdots \vee P_n(s)) \;,$$

where each condition $P_i(s)$ is a conjunction of tests corresponding to a path from the root of the tree to a leaf with a positive outcome. Although this looks like a first-order sentence, it is, in a sense, propositional, because it contains just one variable and all the predicates are unary. The decision tree is really describing a relationship between *WillWait* and some logical combination of attribute values. We cannot use decision trees to represent tests that refer to two or more different objects—for example,

$$\exists r_2 \;\; Nearby(r_2, r) \wedge Price(r, p) \wedge Price(r_2, p_2) \wedge Cheaper(p_2, p)$$

(is there a cheaper restaurant nearby?). Obviously, we could add another Boolean attribute with the name *CheaperRestaurantNearby*, but it is intractable to add *all* such attributes. Chapter 19 will delve further into the problem of learning in first-order logic proper.

Decision trees *are* fully expressive within the class of propositional languages; that is, any Boolean function can be written as a decision tree. This can be done trivially by having each row in the truth table for the function correspond to a path in the tree. This would yield an exponentially large decision tree representation because the truth table has exponentially many rows. Clearly, decision trees can represent many functions with much smaller trees.

PARITY FUNCTION

MAJORITY FUNCTION

For some kinds of functions, however, this is a real problem. For example, if the function is the **parity function**, which returns 1 if and only if an even number of inputs are 1, then an exponentially large decision tree will be needed. It is also difficult to use a decision tree to represent a **majority function**, which returns 1 if more than half of its inputs are 1.

In other words, decision trees are good for some kinds of functions and bad for others. Is there *any* kind of representation that is efficient for *all* kinds of functions? Unfortunately, the answer is no. We can show this in a very general way. Consider the set of all Boolean functions on n attributes. How many different functions are in this set? This is just the number of different truth tables that we can write down, because the function is defined by its truth table. The truth table has 2^n rows, because each input case is described by n attributes. We can consider the "answer" column of the table as a 2^n-bit number that defines the function. No matter what representation we use for functions, some of the functions (almost all of them, in fact) are going to require at least that many bits to represent.

If it takes 2^n bits to define the function, then there are 2^{2^n} different functions on n attributes. This is a scary number. For example, with just six Boolean attributes, there are $2^{2^6} = 18,446,744,073,709,551,616$ different functions to choose from. We will need some ingenious algorithms to find consistent hypotheses in such a large space.

Inducing decision trees from examples

An example for a Boolean decision tree consists of a vector of input attributes, X, and a single Boolean output value y. A set of examples $(X_1, y_1) \ldots, (X_{12}, y_{12})$ is shown in Figure 18.3.

Example	Attributes										Goal
	Alt	*Bar*	*Fri*	*Hun*	*Pat*	*Price*	*Rain*	*Res*	*Type*	*Est*	*WillWait*
X_1	*Yes*	*No*	*No*	*Yes*	*Some*	*$$$*	*No*	*Yes*	*French*	*0–10*	*Yes*
X_2	*Yes*	*No*	*No*	*Yes*	*Full*	*$*	*No*	*No*	*Thai*	*30–60*	*No*
X_3	*No*	*Yes*	*No*	*No*	*Some*	*$*	*No*	*No*	*Burger*	*0–10*	*Yes*
X_4	*Yes*	*No*	*Yes*	*Yes*	*Full*	*$*	*Yes*	*No*	*Thai*	*10–30*	*Yes*
X_5	*Yes*	*No*	*Yes*	*No*	*Full*	*$$$*	*No*	*Yes*	*French*	*>60*	*No*
X_6	*No*	*Yes*	*No*	*Yes*	*Some*	*$$*	*Yes*	*Yes*	*Italian*	*0–10*	*Yes*
X_7	*No*	*Yes*	*No*	*No*	*None*	*$*	*Yes*	*No*	*Burger*	*0–10*	*No*
X_8	*No*	*No*	*No*	*Yes*	*Some*	*$$*	*Yes*	*Yes*	*Thai*	*0–10*	*Yes*
X_9	*No*	*Yes*	*Yes*	*No*	*Full*	*$*	*Yes*	*No*	*Burger*	*>60*	*No*
X_{10}	*Yes*	*Yes*	*Yes*	*Yes*	*Full*	*$$$*	*No*	*Yes*	*Italian*	*10–30*	*No*
X_{11}	*No*	*No*	*No*	*No*	*None*	*$*	*No*	*No*	*Thai*	*0–10*	*No*
X_{12}	*Yes*	*Yes*	*Yes*	*Yes*	*Full*	*$*	*No*	*No*	*Burger*	*30–60*	*Yes*

Figure 18.3 Examples for the restaurant domain.

The positive examples are the ones in which the goal *WillWait* is true $(X_1, X_3, \ldots)$; the negative examples are the ones in which it is false $(X_2, X_5, \ldots)$. The complete set of examples is called the **training set**.

TRAINING SET

The problem of finding a decision tree that agrees with the training set might seem difficult, but in fact there is a trivial solution. We could simply construct a decision tree that has one path to a leaf for each example, where the path tests each attribute in turn and follows the value for the example and the leaf has the classification of the example. When given the same example again,[3] the decision tree will come up with the right classification. Unfortunately, it will not have much to say about any other cases!

The problem with this trivial tree is that it just memorizes the observations. It does not extract any pattern from the examples, so we cannot expect it to be able to extrapolate to examples it has not seen. Applying Ockham's razor, we should find instead the *smallest* decision tree that is consistent with the examples. Unfortunately, for any reasonable definition of "smallest," finding the smallest tree is an intractable problem. With some simple heuristics, however, we can do a good job of finding a "smallish" one. The basic idea behind the DECISION-TREE-LEARNING algorithm is to test the most important attribute first. By "most important," we mean the one that makes the most difference to the classification of an example. That way, we hope to get to the correct classification with a small number of tests, meaning that all paths in the tree will be short and the tree as a whole will be small.

Figure 18.4 shows how the algorithm gets started. We are given 12 training examples, which we classify into positive and negative sets. We then decide which attribute to use as the first test in the tree. Figure 18.4(a) shows that *Type* is a poor attribute, because it leaves us with four possible outcomes, each of which has the same number of positive and negative examples. On the other hand, in Figure 18.4(b) we see that *Patrons* is a fairly important

[3] The same example *or an example with the same description*—this distinction is very important, and we will return to it in Chapter 19.

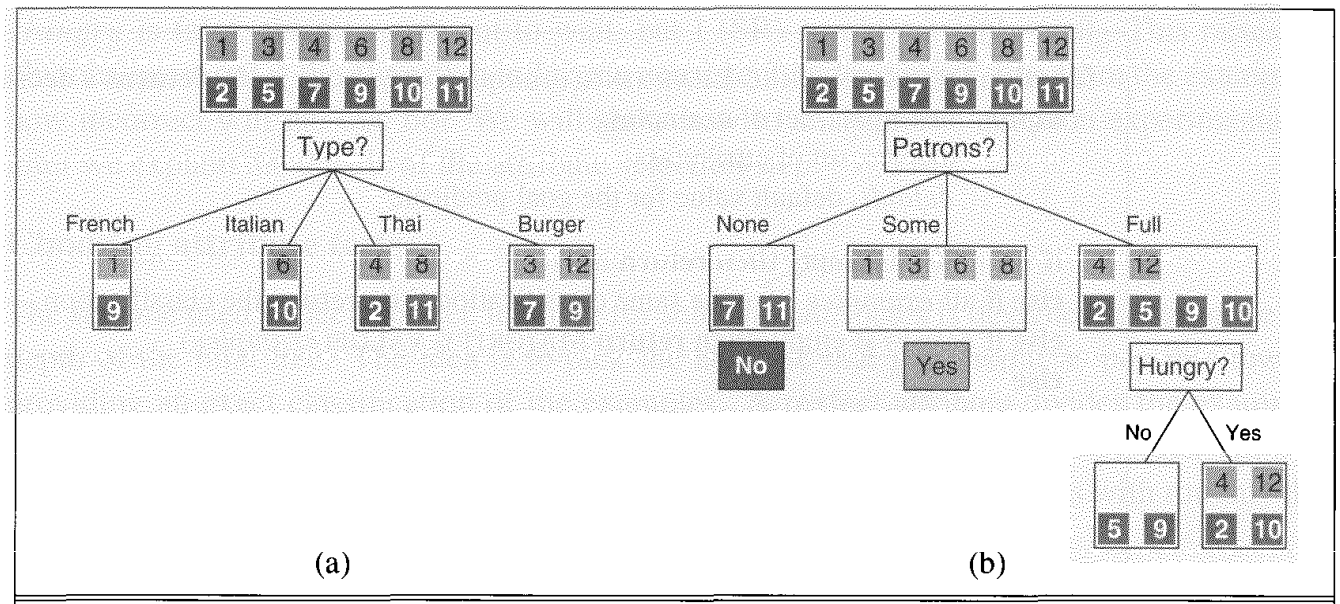


Figure 18.4 Splitting the examples by testing on attributes. (a) Splitting on *Type* brings us no nearer to distinguishing between positive and negative examples. (b) Splitting on *Patrons* does a good job of separating positive and negative examples. After splitting on *Patrons*, *Hungry* is a fairly good second test.

attribute, because if the value is *None* or *Some*, then we are left with example sets for which we can answer definitively (*No* and *Yes*, respectively). If the value is *Full*, we are left with a mixed set of examples. In general, after the first attribute test splits up the examples, each outcome is a new decision tree learning problem in itself, with fewer examples and one fewer attribute. There are four cases to consider for these recursive problems:

1. If there are some positive and some negative examples, then choose the best attribute to split them. Figure 18.4(b) shows *Hungry* being used to split the remaining examples.
2. If all the remaining examples are positive (or all negative), then we are done: we can answer *Yes* or *No*. Figure 18.4(b) shows examples of this in the *None* and *Some* cases.
3. If there are no examples left, it means that no such example has been observed, and we return a default value calculated from the majority classification at the node's parent.
4. If there are no attributes left, but both positive and negative examples, we have a problem. It means that these examples have exactly the same description, but different classifications. This happens when some of the data are incorrect; we say there is **noise** in the data. It also happens either when the attributes do not give enough information to describe the situation fully, or when the domain is truly nondeterministic. One simple way out of the problem is to use a majority vote.

NOISE

The DECISION-TREE-LEARNING algorithm is shown in Figure 18.5. The details of the method for CHOOSE-ATTRIBUTE are given in the next subsection.

The final tree produced by the algorithm applied to the 12-example data set is shown in Figure 18.6. The tree is clearly different from the original tree shown in Figure 18.2, despite the fact that the data were actually generated from an agent using the original tree. One might conclude that the learning algorithm is not doing a very good job of learning the correct

```
function DECISION-TREE-LEARNING(examples, attribs, default) returns a decision tree
  inputs: examples, set of examples
          attribs, set of attributes
          default, default value for the goal predicate

  if examples is empty then return default
  else if all examples have the same classification then return the classification
  else if attribs is empty then return MAJORITY-VALUE(examples)
  else
     best ← CHOOSE-ATTRIBUTE(attribs, examples)
     tree ← a new decision tree with root test best
     m ← MAJORITY-VALUE(examples)
     for each value v_i of best do
        examples_i ← {elements of examples with best = v_i}
        subtree ← DECISION-TREE-LEARNING(examples_i, attribs − best, m)
        add a branch to tree with label v_i and subtree subtree
     return tree
```

Figure 18.5 The decision tree learning algorithm.

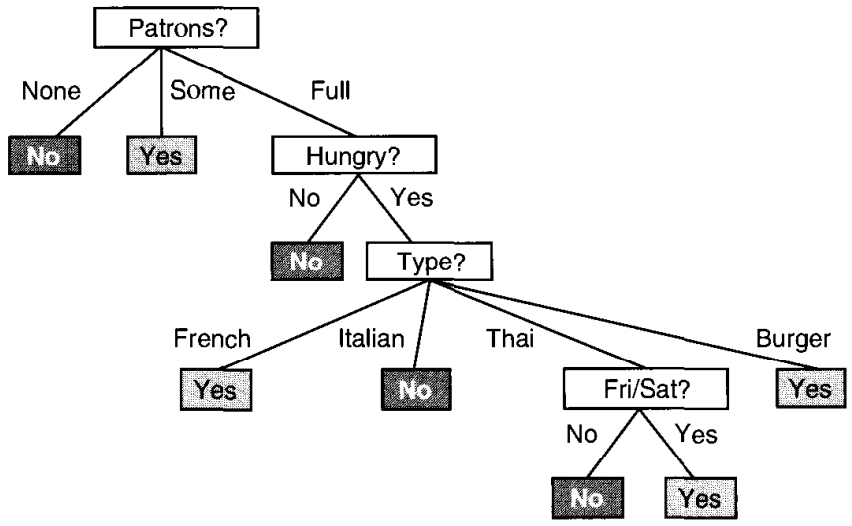


Figure 18.6 The decision tree induced from the 12-example training set.

function. This would be the wrong conclusion to draw, however. The learning algorithm looks at the *examples*, not at the correct function, and in fact, its hypothesis (see Figure 18.6) not only agrees with all the examples, but is considerably simpler than the original tree. The learning algorithm has no reason to include tests for *Raining* and *Reservation*, because it can classify all the examples without them. It has also detected an interesting and previously unsuspected pattern: the first author will wait for Thai food on weekends.

Of course, if we were to gather more examples, we might induce a tree more similar to the original. The tree in Figure 18.6 is bound to make a mistake; for example, it has never seen a case where the wait is 0–10 minutes but the restaurant is full. For a case where

Hungry is false, the tree says not to wait, but I (SR) would certainly wait. This raises an obvious question: if the algorithm induces a consistent, but incorrect, tree from the examples, how incorrect will the tree be? We will show how to analyze this question experimentally, after we explain the details of the attribute selection step.

Choosing attribute tests

The scheme used in decision tree learning for selecting attributes is designed to minimize the depth of the final tree. The idea is to pick the attribute that goes as far as possible toward providing an exact classification of the examples. A perfect attribute divides the examples into sets that are all positive or all negative. The *Patrons* attribute is not perfect, but it is fairly good. A really useless attribute, such as *Type*, leaves the example sets with roughly the same proportion of positive and negative examples as the original set.

All we need, then, is a formal measure of "fairly good" and "really useless" and we can implement the CHOOSE-ATTRIBUTE function of Figure 18.5. The measure should have its maximum value when the attribute is perfect and its minimum value when the attribute is of no use at all. One suitable measure is the expected amount of **information** provided by the attribute, where we use the term in the mathematical sense first defined in Shannon and Weaver (1949). To understand the notion of information, think about it as providing the answer to a question—for example, whether a coin will come up heads. The amount of information contained in the answer depends on one's prior knowledge. The less you know, the more information is provided. Information theory measures information content in **bits**. One bit of information is enough to answer a yes/no question about which one has no idea, such as the flip of a fair coin. In general, if the possible answers v_i have probabilities $P(v_i)$, then the information content I of the actual answer is given by

INFORMATION

$$I(P(v_1), \ldots, P(v_n)) = \sum_{i=1}^{n} -P(v_i) \log_2 P(v_i) \; .$$

To check this equation, for the tossing of a fair coin, we get

$$I\left(\tfrac{1}{2}, \tfrac{1}{2}\right) = -\tfrac{1}{2}\log_2 \tfrac{1}{2} - \tfrac{1}{2}\log_2 \tfrac{1}{2} = 1 \text{ bit.}$$

If the coin is loaded to give 99% heads, we get $I\,(1/100, 99/100) = 0.08$ bits, and as the probability of heads goes to 1, the information of the actual answer goes to 0.

For decision tree learning, the question that needs answering is; for a given example, what is the correct classification? A correct decision tree will answer this question. An estimate of the probabilities of the possible answers before any of the attributes have been tested is given by the proportions of positive and negative examples in the training set. Suppose the training set contains p positive examples and n negative examples. Then an estimate of the information contained in a correct answer is

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

The restaurant training set in Figure 18.3 has $p = n = 6$, so we need 1 bit of information.

Now a test on a single attribute A will not usually tell us this much information, but it will give us some of it. We can measure exactly how much by looking at how much

information we still need *after* the attribute test. Any attribute A divides the training set E into subsets $E_1, \ldots, E_v$ according to their values for A, where A can have v distinct values. Each subset E_i has p_i positive examples and n_i negative examples, so if we go along that branch, we will need an additional $I(p_i/(p_i+n_i), n_i/(p_i+n_i))$ bits of information to answer the question. A randomly chosen example from the training set has the ith value for the attribute with probability $(p_i+n_i)/(p+n)$, so on average, after testing attribute A, we will need

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

INFORMATION GAIN

bits of information to classify the example. The **information gain** from the attribute test is the difference between the original information requirement and the new requirement:

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

The heuristic used in the CHOOSE-ATTRIBUTE function is just to choose the attribute with the largest gain. Returning to the attributes considered in Figure 18.4, we have

$$Gain(Patrons) = 1 - \left[\tfrac{2}{12}I(0,1) + \tfrac{4}{12}I(1,0) + \tfrac{6}{12}I\left(\tfrac{2}{6}, \tfrac{4}{6}\right)\right] \approx 0.541 \text{ bits.}$$

$$Gain(Type) = 1 - \left[\tfrac{2}{12}I\left(\tfrac{1}{2}, \tfrac{1}{2}\right) + \tfrac{2}{12}I\left(\tfrac{1}{2}, \tfrac{1}{2}\right) + \tfrac{4}{12}I\left(\tfrac{2}{4}, \tfrac{2}{4}\right) + \tfrac{4}{12}I\left(\tfrac{2}{4}, \tfrac{2}{4}\right)\right] = 0.$$

confirming our intuition that *Patrons* is a better attribute to split on. In fact, *Patrons* has the highest gain of any of the attributes and would be chosen by the decision-tree learning algorithm as the root.

Assessing the performance of the learning algorithm

A learning algorithm is good if it produces hypotheses that do a good job of predicting the classifications of unseen examples. In Section 18.5, we will see how prediction quality can be estimated in advance. For now, we will look at a methodology for assessing prediction quality after the fact.

Obviously, a prediction is good if it turns out to be true, so we can assess the quality of a hypothesis by checking its predictions against the correct classification once we know it. We do this on a set of examples known as the **test set**. If we train on all our available examples, then we will have to go out and get some more to test on, so often it is more convenient to adopt the following methodology:

TEST SET

1. Collect a large set of examples.
2. Divide it into two disjoint sets: the **training set** and the **test set**.
3. Apply the learning algorithm to the training set, generating a hypothesis h.
4. Measure the percentage of examples in the test set that are correctly classified by h.
5. Repeat steps 2 to 4 for different sizes of training sets and different randomly selected training sets of each size.

The result of this procedure is a set of data that can be processed to give the average prediction quality as a function of the size of the training set. This function can be plotted on a graph, giving what is called the **learning curve** for the algorithm on the particular domain. The

LEARNING CURVE

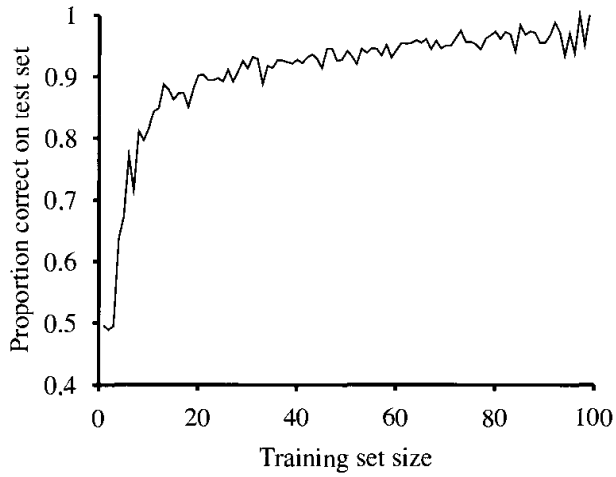


Figure 18.7 A learning curve for the decision tree algorithm on 100 randomly generated examples in the restaurant domain. The graph summarizes 20 trials.

learning curve for DECISION-TREE-LEARNING with the restaurant examples is shown in Figure 18.7. Notice that, as the training set grows, the prediction quality increases. (For this reason, such curves are also called **happy graphs**.) This is a good sign that there is indeed some pattern in the data and the learning algorithm is picking it up.

Obviously, the learning algorithm must not be allowed to "see" the test data before the learned hypothesis is tested on them. Unfortunately, it is all too easy to fall into the trap of **peeking** at the test data. Peeking typically happens as follows: A learning algorithm can have various "knobs" that can be twiddled to tune its behavior—for example, various different criteria for choosing the next attribute in decision tree learning. We generate hypotheses for various different settings of the knobs, measure their performance on the test set, and report the prediction performance of the best hypothesis. Alas, peeking has occurred! The reason is that the hypothesis was selected *on the basis of its test set performance*, so information about the test set has leaked into the learning algorithm. The moral of this tale is that any process that involves comparing the performance of hypotheses on a test set must use a *new* test set to measure the performance of the hypothesis that is finally selected. In practice, this is too difficult, so people continue to run experiments on tainted sets of examples.

PEEKING

Noise and overfitting

We saw earlier that if there are two or more examples with the same description (in terms of the attributes) but different classifications, then the DECISION-TREE-LEARNING algorithm must fail to find a decision tree consistent with all the examples. The solution we mentioned before is to have each leaf node report either the majority classification for its set of examples, if a deterministic hypothesis is required, or report the estimated probabilities of each classification using the relative frequencies. Unfortunately, this is far from the whole story. It is quite possible, and in fact likely, that even when vital information is missing, the decision tree learning algorithm will find a decision tree that is consistent with all the examples. This is because the algorithm can use the *irrelevant* attributes, if any, to make spurious distinctions among the examples.

Consider the problem of trying to predict the roll of a die. Suppose that experiments are carried out during an extended period of time with various dice and that the attributes describing each training example are as follows:

1. *Day*: the day on which the die was rolled (Mon, Tue, Wed, Thu).
2. *Month*: the month in which the die was rolled (Jan or Feb).
3. *Color*: the color of the die (Red or Blue).

As long as no two examples have identical descriptions, DECISION-TREE-LEARNING will find an exact hypothesis. The more attributes there are, the more likely it is that an exact hypothesis will be found. Any such hypothesis will be totally spurious. What we would like is that DECISION-TREE-LEARNING return a single leaf node with probabilities close to 1/6 for each roll, once it has seen enough examples.

Whenever there is a large set of possible hypotheses, one has to be careful not to use the resulting freedom to find meaningless "regularity" in the data. This problem is called **overfitting**. A very general phenomenon, overfitting occurs even when the target function is not at all random. It afflicts every kind of learning algorithm, not just decision trees.

OVERFITTING

A complete mathematical treatment of overfitting is beyond the scope of this book. Here we present a simple technique called **decision tree pruning** to deal with the problem. Pruning works by preventing recursive splitting on attributes that are not clearly relevant, even when the data at that node in the tree are not uniformly classified. The question is, how do we detect an irrelevant attribute?

DECISION TREE PRUNING

Suppose we split a set of examples using an irrelevant attribute. Generally speaking, we would expect the resulting subsets to have roughly the same proportions of each class as the original set. In this case, the information gain will be close to zero.[4] Thus, the information gain is a good clue to irrelevance. Now the question is, how large a gain should we require in order to split on a particular attribute?

We can answer this question by using a statistical **significance test**. Such a test begins by assuming that there is no underlying pattern (the so-called **null hypothesis**). Then the actual data are analyzed to calculate the extent to which they deviate from a perfect absence of pattern. If the degree of deviation is statistically unlikely (usually taken to mean a 5% probability or less), then that is considered to be good evidence for the presence of a significant pattern in the data. The probabilities are calculated from standard distributions of the amount of deviation one would expect to see in random sampling.

SIGNIFICANCE TEST

NULL HYPOTHESIS

In this case, the null hypothesis is that the attribute is irrelevant and, hence, that the information gain for an infinitely large sample would be zero. We need to calculate the probability that, under the null hypothesis, a sample of size v would exhibit the observed deviation from the expected distribution of positive and negative examples. We can measure the deviation by comparing the actual numbers of positive and negative examples in each subset, p_i and n_i, with the expected numbers, $\hat{p}_i$ and $\hat{n}_i$, assuming true irrelevance:

$$\hat{p}_i = p \times \frac{p_i + n_i}{p + n} \qquad \hat{n}_i = n \times \frac{p_i + n_i}{p + n} .$$

[4] In fact, the gain will be positive unless the proportions are all exactly the same. (See Exercise 18.10.)

A convenient measure of the total deviation is given by

$$D = \sum_{i=1}^{v} \frac{(p_i - \hat{p}_i)^2}{\hat{p}_i} + \frac{(n_i - \hat{n}_i)^2}{\hat{n}_i} .$$

Under the null hypothesis, the value of D is distributed according to the χ^2 (chi-squared) distribution with $v - 1$ degrees of freedom. The probability that the attribute is really irrelevant can be calculated with the help of standard χ^2 tables or with statistical software. Exercise 18.11 asks you to make the appropriate changes to DECISION-TREE-LEARNING to implement this form of pruning, which is known as χ^2 **pruning**.

χ^2 PRUNING

With pruning, noise can be tolerated: classification errors give a linear increase in prediction error, whereas errors in the descriptions of examples have an asymptotic effect that gets worse as the tree shrinks down to smaller sets. Trees constructed with pruning perform significantly better than trees constructed without pruning when the data contain a large amount of noise. The pruned trees are often much smaller and hence easier to understand.

CROSS-VALIDATION

Cross-validation is another technique that reduces overfitting. It can be applied to any learning algorithm, not just decision tree learning. The basic idea is to estimate how well each hypothesis will predict unseen data. This is done by setting aside some fraction of the known data and using it to test the prediction performance of a hypothesis induced from the remaining data. K-fold cross-validation means that you run k experiments, each time setting aside a different $1/k$ of the data to test on, and average the results. Popular values for k are 5 and 10. The extreme is $k = n$, also known as leave-one-out cross-validation. Cross-validation can be used in conjunction with any tree-construction method (including pruning) in order to select a tree with good prediction performance. To avoid peeking, we must then measure this performance with a new test set.

Broadening the applicability of decision trees

In order to extend decision tree induction to a wider variety of problems, a number of issues must be addressed. We will briefly mention each, suggesting that a full understanding is best obtained by doing the associated exercises:

- ◇ **Missing data**: In many domains, not all the attribute values will be known for every example. The values might have gone unrecorded, or they might be too expensive to obtain. This gives rise to two problems: First, given a complete decision tree, how should one classify an object that is missing one of the test attributes? Second, how should one modify the information gain formula when some examples have unknown values for the attribute? These questions are addressed in Exercise 18.12.
- ◇ **Multivalued attributes**: When an attribute has many possible values, the information gain measure gives an inappropriate indication of the attribute's usefulness. In the extreme case, we could use an attribute, such as *RestaurantName*, that has a different value for every example. Then each subset of examples would be a singleton with a unique classification, so the information gain measure would have its highest value for this attribute. Nonetheless, the attribute could be irrelevant or useless. One solution is to use the **gain ratio** (Exercise 18.13).

GAIN RATIO

◇ **Continuous and integer-valued input attributes**: Continuous or integer-valued attributes such as *Height* and *Weight*, have an infinite set of possible values. Rather than generate infinitely many branches, decision-tree learning algorithms typically find the **split point** that gives the highest information gain. For example, at a given node in the tree, it might be the case that testing on $Weight > 160$ gives the most information. Efficient dynamic programming methods exist for finding good split points, but it is still by far the most expensive part of real-world decision tree learning applications.

SPLIT POINT

◇ **Continuous-valued output attributes**: If we are trying to predict a numerical value, such as the price of a work of art, rather than a discrete classification, then we need a **regression tree**. Such a tree has at each leaf a linear function of some subset of numerical attributes, rather than a single value. For example, the branch for hand-colored engravings might end with a linear function of area, age, and number of colors. The learning algorithm must decide when to stop splitting and begin applying linear regression using the remaining attributes (or some subset thereof).

REGRESSION TREE

A decision-tree learning system for real-world applications must be able to handle all of these problems. Handling continuous-valued variables is especially important, because both physical and financial processes provide numerical data. Several commercial packages have been built that meet these criteria, and they have been used to develop several hundred fielded systems. In many areas of industry and commerce, decision trees are usually the first method tried when a classification method is to be extracted from a data set. One important property of decision trees is that it is possible for a human to understand the output of the learning algorithm. (Indeed, this is a *legal requirement* for financial decisions that are subject to anti-discrimination laws.) This is a property not shared by neural networks (see Chapter 20).

18.4 Ensemble Learning

ENSEMBLE LEARNING

So far we have looked at learning methods in which a single hypothesis, chosen from a hypothesis space, is used to make predictions. The idea of **ensemble learning** methods is to select a whole collection, or **ensemble**, of hypotheses from the hypothesis space and combine their predictions. For example, we might generate a hundred different decision trees from the same training set and have them vote on the best classification for a new example.

The motivation for ensemble learning is simple. Consider an ensemble of $M = 5$ hypotheses and suppose that we combine their predictions using simple majority voting. For the ensemble to misclassify a new example, *at least three of the five hypotheses have to misclassify it.* The hope is that this is much less likely than a misclassification by a single hypothesis. Suppose we assume that each hypothesis h_i in the ensemble has an error of p—that is, the probability that a randomly chosen example is misclassified by h_i is p. Furthermore, suppose we assume that the errors made by each hypothesis are *independent*. In that case, if p is small, then the probability of a large number of misclassifications occurring is minuscule. For example, a simple calculation (Exercise 18.14) shows that using an ensemble of five hypotheses reduces an error rate of 1 in 10 down to an error rate of less than 1 in 100. Now, obviously

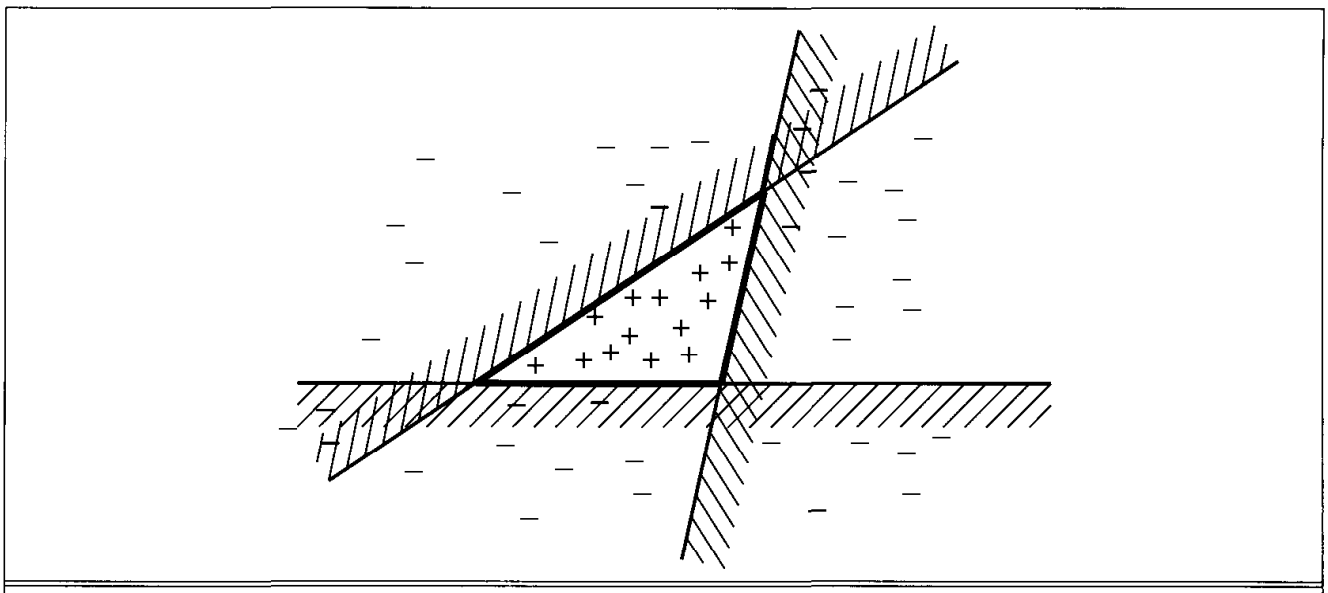


Figure 18.8 Illustration of the increased expressive power obtained by ensemble learning. We take three linear threshold hypotheses, each of which classifies positively on the non-shaded side, and classify as positive any example classified positively by all three. The resulting triangular region is a hypothesis not expressible in the original hypothesis space.

the assumption of independence is unreasonable, because hypotheses are likely to be misled in the same way by any misleading aspects of the training data. But if the hypotheses are at least a little bit different, thereby reducing the correlation between their errors, then ensemble learning can be very useful.

Another way to think about the ensemble idea is as a generic way of enlarging the hypothesis space. That is, think of the ensemble itself as a hypothesis and the new hypothesis space as the set of all possible ensembles constructible from hypotheses in the original space. Figure 18.8 shows how this can result in a more expressive hypothesis space. If the original hypothesis space allows for a simple and efficient learning algorithm, then the ensemble method provides a way to learn a much more expressive class of hypotheses without incurring much additional computational or algorithmic complexity.

BOOSTING

WEIGHTED TRAINING SET

The most widely used ensemble method is called **boosting**. To understand how it works, we need first to explain the idea of a **weighted training set**. In such a training set, each example has an associated weight $w_j \geq 0$. The higher the weight of an example, the higher is the importance attached to it during the learning of a hypothesis. It is straightforward to modify the learning algorithms we have seen so far to operate with weighted training sets.[5]

Boosting starts with $w_j = 1$ for all the examples (i.e., a normal training set). From this set, it generates the first hypothesis, h_1. This hypothesis will classify some of the training examples correctly and some incorrectly. We would like the next hypothesis to do better on the misclassified examples, so we increase their weights while decreasing the weights of the correctly classified examples. From this new weighted training set, we generate hypothesis h_2. The process continues in this way until we have generated M hypotheses, where M is

[5] For learning algorithms in which this is not possible, one can instead create a **replicated training set** where the ith example appears w_j times, using randomization to handle fractional weights.

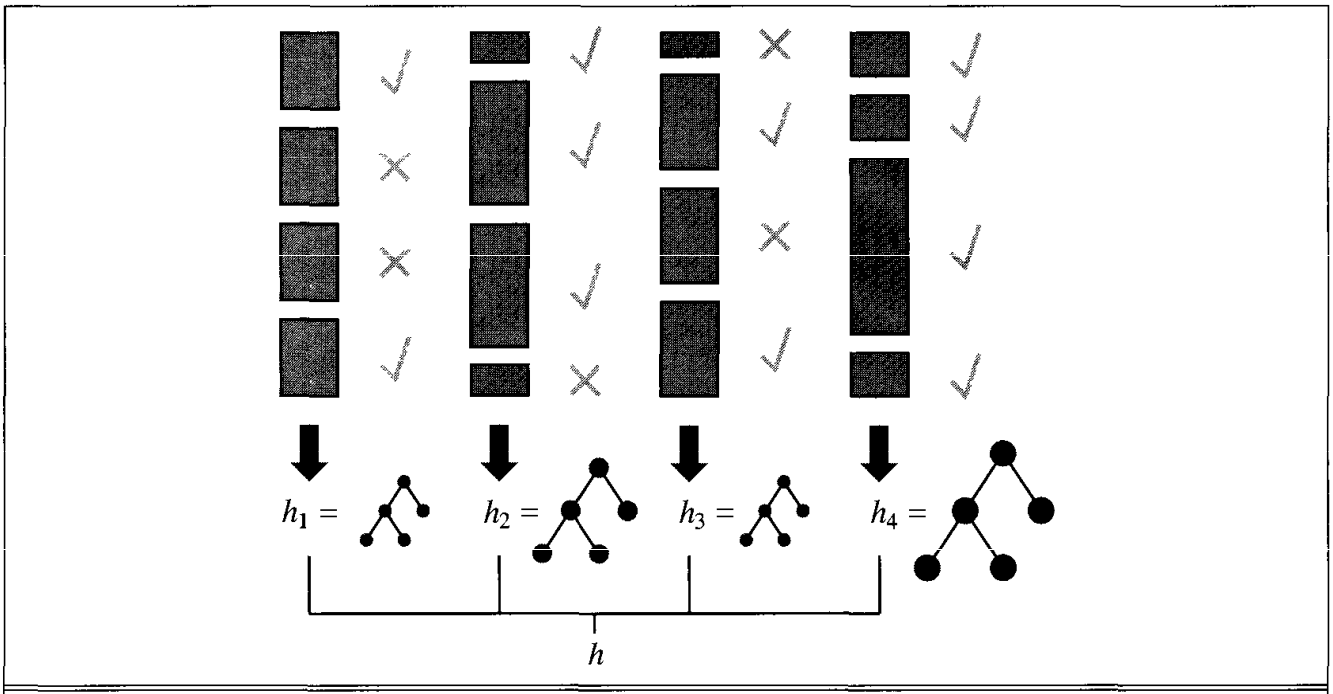


Figure 18.9 How the boosting algorithm works. Each shaded rectangle corresponds to an example; the height of the rectangle corresponds to the weight. The ticks and crosses indicate whether the example was classified correctly by the current hypothesis. The size of the decision tree indicates the weight of that hypothesis in the final ensemble.

an input to the boosting algorithm. The final ensemble hypothesis is a weighted-majority combination of all the M hypotheses, each weighted according to how well it performed on the training set. Figure 18.9 shows how the algorithm works conceptually. There are many variants of the basic boosting idea with different ways of adjusting the weights and combining the hypotheses. One specific algorithm, called ADABOOST, is shown in Figure 18.10. While the details of the weight adjustments are not so important, ADABOOST does have a very important property: if the input learning algorithm L is a **weak learning** algorithm—which means that L always returns a hypothesis with weighted error on the training set that is slightly better than random guessing (i.e., 50% for Boolean classification)—then ADABOOST will return a hypothesis that *classifies the training data perfectly* for large enough M. Thus, the algorithm *boosts* the accuracy of the original learning algorithm on the training data. This result holds no matter how inexpressive the original hypothesis space and no matter how complex the function being learned.

WEAK LEARNING

Let us see how well boosting does on the restaurant data. We will choose as our original hypothesis space the class of **decision stumps**, which are decision trees with just one test at the root. The lower curve in Figure 18.11(a) shows that unboosted decision stumps are not very effective for this data set, reaching a prediction performance of only 81% on 100 training examples. When boosting is applied (with $M = 5$), the performance is better, reaching 93% after 100 examples.

DECISION STUMP

An interesting thing happens as the ensemble size M increases. Figure 18.11(b) shows the training set performance (on 100 examples) as a function of M. Notice that the error reaches zero (as the boosting theorem tells us) when M is 20; that is, a weighted-majority

```
function ADABOOST(examples, L, M) returns a weighted-majority hypothesis
  inputs: examples, set of N labelled examples (x_1, y_1), ..., (x_N, y_N)
          L, a learning algorithm
          M, the number of hypotheses in the ensemble
  local variables: w, a vector of N example weights, initially 1/N
                   h, a vector of M hypotheses
                   z, a vector of M hypothesis weights

  for m = 1 to M do
     h[m] ← L(examples, w)
     error ← 0
     for j = 1 to N do
        if h[m](x_j) ≠ y_j then error ← error + w[j]
     for j = 1 to N do
        if h[m](x_j) = y_j then w[j] ← w[j] · error/(1 − error)
     w ← NORMALIZE(w)
     z[m] ← log (1 − error)/error
  return WEIGHTED-MAJORITY(h, z)
```

Figure 18.10 The ADABOOST variant of the boosting method for ensemble learning. The algorithm generates hypotheses by successively reweighting the trainng examples. The function WEIGHTED-MAJORITY generates a hypothesis that returns the output value with the highest vote from the hypotheses in **h**, with votes weighted by **z**.

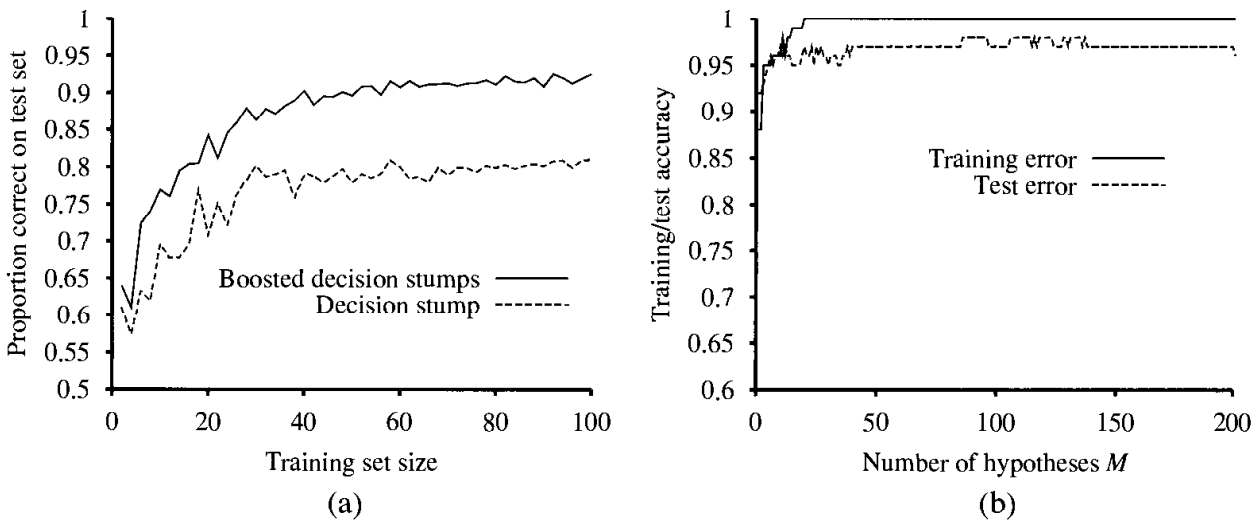


Figure 18.11 (a) Graph showing the performance of boosted decision stumps with $M = 5$ versus decision stumps on the restaurant data. (b) The proportion correct on the training set and the test set as a function of M, the number of hypotheses in the ensemble. Notice that the test set accuracy improves slightly even after the training accuracy reaches 1, i.e., after the ensemble fits the data exactly.

combination of 20 decision stumps suffices to fit the 100 examples exactly. As more stumps are added to the ensemble, the error remains at zero. The graph also shows that *the test set performance continues to increase long after the training set error has reached zero.* At $M = 20$, the test performance is 0.95 (or 0.05 error), and the performance increases to 0.98 as late as $M = 137$, before gradually dropping to 0.95.

This finding, which is quite robust across data sets and hypothesis spaces, came as quite a surprise when it was first noticed. Ockham's razor tells us not to make hypotheses more complex than necessary, but the graph tells us that the predictions *improve* as the ensemble hypothesis gets more complex! Various explanations have been proposed for this. One view is that boosting approximates **Bayesian learning** (see Chapter 20), which can be shown to be an optimal learning algorithm, and the approximation improves as more hypotheses are added. Another possible explanation is that the addition of further hypotheses enables the ensemble to be *more definite* in its distinction between positive and negative examples, which helps it when it comes to classifying new examples.

18.5 Why Learning Works: Computational Learning Theory

The main unanswered question posed in Section 18.2 was this: how can one be sure that one's learning algorithm has produced a theory that will correctly predict the future? In formal terms, how do we know that the hypothesis h is close to the target function f if we don't know what f is? These questions have been pondered for several centuries. Until we find answers, machine learning will, at best, be puzzled by its own success.

COMPUTATIONAL LEARNING THEORY

The approach taken in this section is based on **computational learning theory**, a field at the intersection of AI, statistics, and theoretical computer science. The underlying principle is the following: *any hypothesis that is seriously wrong will almost certainly be "found out" with high probability after a small number of examples, because it will make an incorrect prediction. Thus, any hypothesis that is consistent with a sufficiently large set of training examples is unlikely to be seriously wrong: that is, it must be* ***probably approximately correct.*** Any learning algorithm that returns hypotheses that are probably approximately correct is called a **PAC-learning** algorithm.

PROBABLY APPROXIMATELY CORRECT

PAC-LEARNING

There are some subtleties in the preceding argument. The main question is the connection between the training and the test examples; after all, we want the hypothesis to be approximately correct on the test set, not just on the training set. The key assumption is that the training and test sets are drawn randomly and independently from the same population of examples with the *same probability distribution.* This is called the **stationarity** assumption. Without the stationarity assumption, the theory can make no claims at all about the future, because there would be no necessary connection between future and past. The stationarity assumption amounts to supposing that the process that selects examples is not malevolent. Obviously, if the training set consists only of weird examples—two-headed dogs, for instance—then the learning algorithm cannot help but make unsuccessful generalizations about how to recognize dogs.

STATIONARITY

How many examples are needed?

In order to put these insights into practice, we will need some notation:

- Let **X** be the set of all possible examples.
- Let D be the distribution from which examples are drawn.
- Let **H** be the set of possible hypotheses.
- Let N be the number of examples in the training set.

Initially, we will assume that the true function f is a member of **H**. Now we can define the **error** of a hypothesis h with respect to the true function f given a distribution D over the examples as the probability that h is different from f on an example:

ERROR

$$\text{error}(h) = P(h(x) \neq f(x) | x \text{ drawn from } D) .$$

This is the same quantity being measured experimentally by the learning curves shown earlier.

A hypothesis h is called **approximately correct** if $\text{error}(h) \leq \epsilon$, where ϵ is a small constant. The plan of attack is to show that after seeing N examples, with high probability, all consistent hypotheses will be approximately correct. One can think of an approximately correct hypothesis as being "close" to the true function in hypothesis space: it lies inside what is called the **ϵ-ball** around the true function f. Figure 18.12 shows the set of all hypotheses **H**, divided into the ϵ-ball around f and the remainder, which we call $\mathbf{H}_{\text{bad}}$.

ϵ-BALL

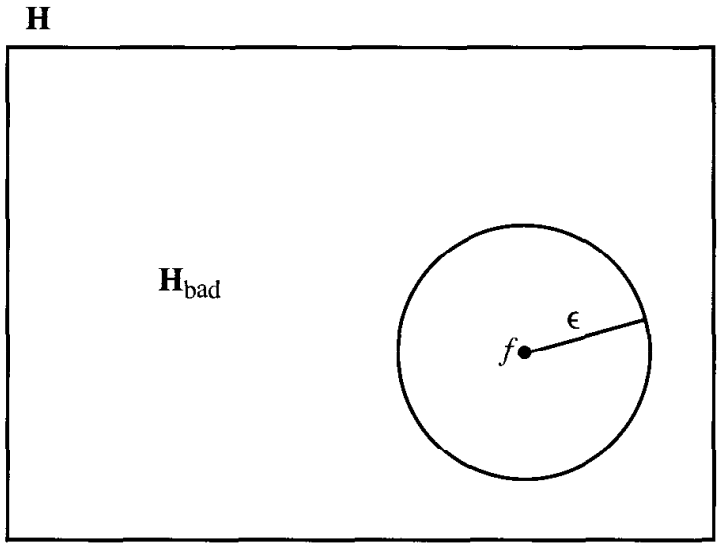


Figure 18.12 Schematic diagram of hypothesis space, showing the "ϵ-ball" around the true function f.

We can calculate the probability that a "seriously wrong" hypothesis $h_b \in \mathbf{H}_{\text{bad}}$ is consistent with the first N examples as follows. We know that $\text{error}(h_b) > \epsilon$. Thus, the probability that it agrees with a given example is at least $1 - \epsilon$. The bound for N examples is

$$P(h_b \text{ agrees with } N \text{ examples}) \leq (1 - \epsilon)^N .$$

The probability that $\mathbf{H}_{\text{bad}}$ contains at least one consistent hypothesis is bounded by the sum of the individual probabilities:

$$P(\mathbf{H}_{\text{bad}} \text{ contains a consistent hypothesis}) \leq |\mathbf{H}_{\text{bad}}|(1 - \epsilon)^N \leq |\mathbf{H}|(1 - \epsilon)^N ,$$

where we have used the fact that $|\mathbf{H}_{\text{bad}}| \leq |\mathbf{H}|$. We would like to reduce the probability of this event below some small number δ:

$$|\mathbf{H}|(1-\epsilon)^N \leq \delta \ .$$

Given that $1-\epsilon \leq e^{-\epsilon}$, we can achieve this if we allow the algorithm to see

$$N \geq \frac{1}{\epsilon}\left(\ln\frac{1}{\delta} + \ln|\mathbf{H}|\right) \tag{18.1}$$

examples. Thus, if a learning algorithm returns a hypothesis that is consistent with this many examples, then with probability at least $1-\delta$, it has error at most ϵ. In other words, it is probably approximately correct. The number of required examples, as a function of ϵ and δ, is called the **sample complexity** of the hypothesis space.

SAMPLE COMPLEXITY

It appears, then, that the key question is the size of the hypothesis space. As we saw earlier, if $\mathbf{H}$ is the set of all Boolean functions on n attributes, then $|\mathbf{H}| = 2^{2^n}$. Thus, the sample complexity of the space grows as 2^n. Because the number of possible examples is also 2^n, this says that any learning algorithm for the space of all Boolean functions will do no better than a lookup table if it merely returns a hypothesis that is consistent with all known examples. Another way to see this is to observe that for any unseen example, the hypothesis space will contain as many consistent hypotheses that predict a positive outcome as it does hypotheses that predict a negative outcome.

The dilemma we face, then, is that unless we restrict the space of functions the algorithm can consider, it will not be able to learn; but if we do restrict the space, we might eliminate the true function altogether. There are two ways to "escape" this dilemma. The first way is to insist that the algorithm return not just any consistent hypothesis, but preferably a simple one (as is done in decision tree learning). The theoretical analysis of such algorithms is beyond the scope of this book, but in cases where finding simple consistent hypotheses is tractable, the sample complexity results are generally better than for analyses based only on consistency. The second escape, which we pursue here, is to focus on learnable subsets of the entire set of Boolean functions. The idea is that in most cases we do not need the full expressive power of Boolean functions, and can get by with more restricted languages. We now examine one such restricted language in more detail.

Learning decision lists

DECISION LIST

A **decision list** is a logical expression of a restricted form. It consists of a series of tests, each of which is a conjunction of literals. If a test succeeds when applied to an example description, the decision list specifies the value to be returned. If the test fails, processing continues with the next test in the list.[6] Decision lists resemble decision trees, but their overall structure is simpler. In contrast, the individual tests are more complex. Figure 18.13 shows a decision list that represents the following hypothesis:

$$\forall x \ \ WillWait(x) \Leftrightarrow Patrons(x, Some) \vee (Patrons(x, Full) \wedge Fri/Sat(x)) \ .$$

If we allow tests of arbitrary size, then decision lists can represent any Boolean function (Exercise 18.15). On the other hand, if we restrict the size of each test to at most k literals,

[6] A decision list is therefore identical in structure to a COND statement in Lisp.

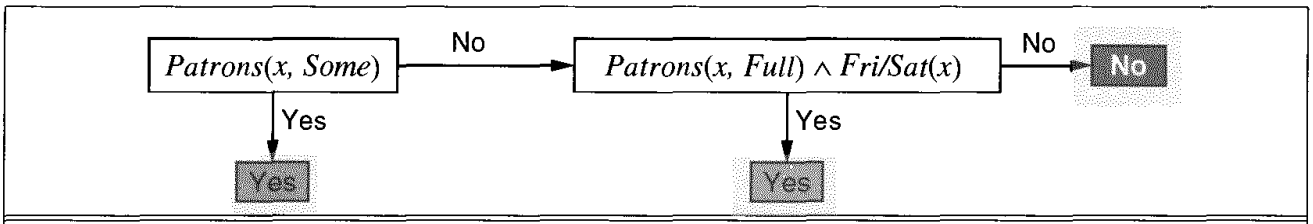


Figure 18.13 A decision list for the restaurant problem.

then it is possible for the learning algorithm to generalize successfully from a small number of examples. We call this language k-**DL**. The example in Figure 18.13 is in 2-DL. It is easy to show (Exercise 18.15) that k-DL includes as a subset the language k-**DT**, the set of all decision trees of depth at most k. It is important to remember that the particular language referred to by k-DL depends on the attributes used to describe the examples. We will use the notation k-DL(n) to denote a k-DL language using n Boolean attributes.

k-DL
k-DT

The first task is to show that k-DL is learnable—that is, that any function in k-DL can be approximated accurately after training on a reasonable number of examples. To do this, we need to calculate the number of hypotheses in the language. Let the language of tests—conjunctions of at most k literals using n attributes—be $Conj(n, k)$. Because a decision list is constructed of tests, and because each test can be attached to either a *Yes* or a *No* outcome or can be absent from the decision list, there are at most $3^{|Conj(n,k)|}$ distinct sets of component tests. Each of these sets of tests can be in any order, so

$$|k\text{-DL}(n)| \leq 3^{|Conj(n,k)|}|Conj(n,k)|!\ .$$

The number of conjunctions of k literals from n attributes is given by

$$|Conj(n,k)| = \sum_{i=0}^{k} \binom{2n}{i} = O(n^k)\ .$$

Hence, after some work, we obtain

$$|k\text{-DL}(n)| = 2^{O(n^k \log_2(n^k))}\ .$$

We can plug this into Equation (18.1) to show that the number of examples needed for PAC-learning a k-DL function is polynomial in n:

$$N \geq \frac{1}{\epsilon}\left(\ln\frac{1}{\delta} + O(n^k \log_2(n^k))\right)\ .$$

Therefore, any algorithm that returns a consistent decision list will PAC-learn a k-DL function in a reasonable number of examples, for small k.

The next task is to find an efficient algorithm that returns a consistent decision list. We will use a greedy algorithm called DECISION-LIST-LEARNING that repeatedly finds a test that agrees exactly with some subset of the training set. Once it finds such a test, it adds it to the decision list under construction and removes the corresponding examples. It then constructs the remainder of the decision list, using just the remaining examples. This is repeated until there are no examples left. The algorithm is shown in Figure 18.14.

This algorithm does not specify the method for selecting the next test to add to the decision list. Although the formal results given earlier do not depend on the selection method,

```
function DECISION-LIST-LEARNING(examples) returns a decision list, or failure

  if examples is empty then return the trivial decision list No
  t ← a test that matches a nonempty subset examples_t of examples
      such that the members of examples_t are all positive or all negative
  if there is no such t then return failure
  if the examples in examples_t are positive then o ← Yes else o ← No
  return a decision list with initial test t and outcome o and remaining tests given by
      DECISION-LIST-LEARNING(examples − examples_t)
```

Figure 18.14 An algorithm for learning decision lists.

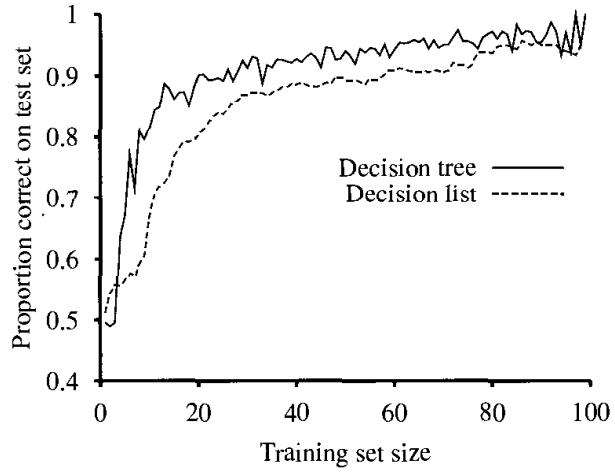


Figure 18.15 Learning curve for DECISION-LIST-LEARNING algorithm on the restaurant data. The curve for DECISION-TREE-LEARNING is shown for comparison.

it would seem reasonable to prefer small tests that match large sets of uniformly classified examples, so that the overall decision list will be as compact as possible. The simplest strategy is to find the smallest test t that matches any uniformly classified subset, regardless of the size of the subset. Even this approach works quite well, as Figure 18.15 suggests.

Discussion

IDENTIFICATION IN THE LIMIT

Computational learning theory has generated a new way of looking at the problem of learning. In the early 1960s, the theory of learning focused on the problem of **identification in the limit**. According to this notion, an identification algorithm must return a hypothesis that exactly matches the true function. One way to do that is as follows: First, order all the hypotheses in **H** according to some measure of simplicity. Then, choose the simplest hypothesis consistent with all the examples so far. As new examples arrive, the method will abandon a simpler hypothesis that is invalidated and adopt a more complex one instead. Once it reaches the true function, it will never abandon it. Unfortunately, in many hypothesis spaces, the number of examples and the computation time required to reach the true function are enormous. Thus, computational learning theory does not insist that the learning agent find the "one true

law" governing its environment, but instead that it find a hypothesis with a certain degree of predictive accuracy. Computational learning theory also brings sharply into focus the tradeoff between the expressiveness of the hypothesis language and the complexity of learning, and has led directly to an important class of learning algorithms called support vector machines.

The PAC-learning results we have shown are worst-case complexity results and do not necessarily reflect the average-case sample complexity as measured by the learning curves we have shown. An average-case analysis must also make assumptions about the distribution of examples and the distribution of true functions that the algorithm will have to learn. As these issues become better understood, computational learning theory continues to provide valuable guidance to machine learning researchers who are interested in predicting or modifying the learning ability of their algorithms. Besides decision lists, results have been obtained for almost all known subclasses of Boolean functions, for sets of first-order logical sentences (see Chapter 19), and for neural networks (see Chapter 20). The results show that the pure inductive learning problem, where the agent begins with no prior knowledge about the target function, is generally very hard. As we show in Chapter 19, the use of prior knowledge to guide inductive learning makes it possible to learn quite large sets of sentences from reasonable numbers of examples, even in a language as expressive as first-order logic.

18.6 SUMMARY

This chapter has concentrated on inductive learning of deterministic functions from examples. The main points were as follows:

- Learning takes many forms, depending on the nature of the performance element, the component to be improved, and the available feedback.
- If the available feedback, either from a teacher or from the environment, provides the correct value for the examples, the learning problem is called **supervised learning**. The task, also called **inductive learning**, is then to learn a function from examples of its inputs and outputs. Learning a discrete-valued function is called **classification**; learning a continuous function is called **regression**.
- Inductive learning involves finding a **consistent** hypothesis that agrees with the examples. **Ockham's razor** suggests choosing the simplest consistent hypothesis. The difficulty of this task depends on the chosen representation.
- **Decision trees** can represent all Boolean functions. The **information gain** heuristic provides an efficient method for finding a simple, consistent decision tree.
- The performance of a learning algorithm is measured by the **learning curve**, which shows the prediction accuracy on the **test set** as a function of the **training set** size.
- Ensemble methods such as **boosting** often perform better than individual methods.
- **Computational learning theory** analyzes the sample complexity and computational complexity of inductive learning. There is a tradeoff between the expressiveness of the hypothesis language and the ease of learning.

BIBLIOGRAPHICAL AND HISTORICAL NOTES

Chapter 1 outlined the history of philosophical investigations into inductive learning. William of Ockham (1280–1349), the most influential philosopher of his century and a major contributer to medieval epistemology, logic, and metaphysics, is credited with a statement called "Ockham's Razor"—in Latin, *Entia non sunt multiplicanda praeter necessitatem*, and in English, "Entities are not to be multiplied beyond necessity." Unfortunately, this laudable piece of advice is nowhere to be found in his writings in precisely these words.

EPAM, the "Elementary Perceiver And Memorizer" (Feigenbaum, 1961), was one of the earliest systems to use decision trees (or **discrimination nets**). EPAM was intended as a cognitive-simulation model of human concept learning. CLS (Hunt *et al.*, 1966) used a heuristic look-ahead method to construct decision trees. ID3 (Quinlan, 1979) added the crucial idea of using information content to provide the heuristic function. Information theory itself was developed by Claude Shannon to aid in the study of communication (Shannon and Weaver, 1949). (Shannon also contributed one of the earliest examples of machine learning, a mechanical mouse named Theseus that learned to navigate through a maze by trial and error.) The χ^2 method of tree pruning was described by Quinlan (1986). C4.5, an industrial-strength decision tree package, can be found in Quinlan (1993). An independent tradition of decision tree learning exists in the statistical literature. *Classification and Regression Trees* (Breiman *et al.*, 1984), known as the "CART book," is the principal reference.

Many other algorithmic approaches to learning have been tried. The **current-best-hypothesis** approach maintains a single hypothesis, specializing it when it proves too broad and generalizing it when it proves too narrow. This is an old idea in philosophy (Mill, 1843). Early work in cognitive psychology also suggested that it is a natural form of concept learning in humans (Bruner *et al.*, 1957). In AI, the approach is most closely associated with the work of Patrick Winston, whose Ph.D. thesis (Winston, 1970) addressed the problem of learning descriptions of complex objects. The **version space** method (Mitchell, 1977, 1982) takes a different approach, maintaining the set of *all* consistent hypotheses and eliminating those found to be inconsistent with new examples. The approach was used in the Meta-DENDRAL expert system for chemistry (Buchanan and Mitchell, 1978), and later in Mitchell's (1983) LEX system, which learns to solve calculus problems. A third influential thread was formed by the work of Michalski and colleagues on the AQ series of algorithms, which learned sets of logical rules (Michalski, 1969; Michalski *et al.*, 1986b).

Ensemble learning is an increasingly popular technique for improving the performance of learning algorithms. **Bagging** (Breiman, 1996), the first effective method, combines hypotheses learned from multiple **bootstrap** data sets, each generated by subsampling the original data set. The **boosting** method described in the chapter originated with theoretical work by Schapire (1990). The ADABOOST algorithm was developed by Freund and Schapire (1996) and analyzed theoretically by Schapire (1999). Friedman *et al.* (2000) explain boosting from a statistician's viewpoint.

BAGGING

Theoretical analysis of learning algorithms began with the work of Gold (1967) on **identification in the limit**. This approach was motivated in part by models of scientific

discovery from the philosophy of science (Popper, 1962), but has been applied mainly to the problem of learning grammars from example sentences (Osherson *et al.*, 1986).

KOLMOGOROV COMPLEXITY

Whereas the identification-in-the-limit approach concentrates on eventual convergence, the study of **Kolmogorov complexity** or **algorithmic complexity**, developed independently by Solomonoff (1964) and Kolmogorov (1965), attempts to provide a formal definition for the notion of simplicity used in Ockham's razor. To escape the problem that simplicity depends on the way in which information is represented, it is proposed that simplicity be measured by the length of the shortest program for a universal Turing machine that correctly reproduces the observed data. Although there are many possible universal Turing machines, and hence many possible "shortest" programs, these programs differ in length by at most a constant that is independent of the amount of data. This beautiful insight, which essentially shows that *any* initial representation bias will eventually be overcome by the data itself, is marred only by the undecidability of computing the length of the shortest program. Approximate measures such as the **minimum description length**, or MDL (Rissanen, 1984) can be used instead and have produced excellent results in practice. The text by Li and Vitanyi (1993) is the best source for Kolmogorov complexity.

MINIMUM DESCRIPTION LENGTH

Computational learning theory—that is, the theory of PAC-learning—was inaugurated by Leslie Valiant (1984). Valiant's work stressed the importance of computational and sample complexity. With Michael Kearns (1990), Valiant showed that several concept classes cannot be PAC-learned tractably, even though sufficient information is available in the examples. Some positive results were obtained for classes such as decision lists (Rivest, 1987).

UNIFORM CONVERGENCE THEORY

VC DIMENSION

An independent tradition of sample complexity analysis has existed in statistics, beginning with the work on **uniform convergence theory** (Vapnik and Chervonenkis, 1971). The so-called **VC dimension** provides a measure roughly analogous to, but more general than, the $\ln |\mathbf{H}|$ measure obtained from PAC analysis. The VC dimension can be applied to continuous function classes, to which standard PAC analysis does not apply. PAC-learning theory and VC theory were first connected by the "four Germans" (none of whom actually is German): Blumer, Ehrenfeucht, Haussler, and Warmuth (1989). Subsequent developments in VC theory led to the invention of the **support vector machine** or SVM (Boser *et al.*, 1992; Vapnik, 1998), which we describe in Chapter 20.

A large number of important papers on machine learning have been collected in *Readings in Machine Learning* (Shavlik and Dietterich, 1990). The two volumes *Machine Learning 1* (Michalski *et al.*, 1983) and *Machine Learning 2* (Michalski *et al.*, 1986a) also contain many important papers, as well as huge bibliographies. Weiss and Kulikowski (1991) provide a broad introduction to function-learning methods from machine learning, statistics, and neural networks. The STATLOG project (Michie *et al.*, 1994) is by far the most exhaustive investigation into the comparative performance of learning algorithms. Good current research in machine learning is published in the annual proceedings of the International Conference on Machine Learning and the conference on Neural Information Processing Systems, in *Machine Learning* and the *Journal of Machine Learning Research*, and in mainstream AI journals. Work in computational learning theory also appears in the annual ACM Workshop on Computational Learning Theory (COLT), and is described in the texts by Kearns and Vazirani (1994) and Anthony and Bartlett (1999).

EXERCISES

18.1 Consider the problem faced by an infant learning to speak and understand a language. Explain how this process fits into the general learning model, identifying each of the components of the model as appropriate.

18.2 Repeat Exercise 18.1 for the case of learning to play tennis (or some other sport with which you are familiar). Is this supervised learning or reinforcement learning?

18.3 Draw a decision tree for the problem of deciding whether to move forward at a road intersection, given that the light has just turned green.

18.4 We never test the same attribute twice along one path in a decision tree. Why not?

18.5 Suppose we generate a training set from a decision tree and then apply decision-tree learning to that training set. Is it the case that the learning algorithm will eventually return the correct tree as the training set size goes to infinity? Why or why not?

18.6 A good "straw man" learning algorithm is as follows: create a table out of all the training examples. Identify which output occurs most often among the training examples; call it d. Then when given an input that is not in the table, just return d. For inputs that *are* in the table, return the output associated with it (or the most frequent output, if there is more than one). Implement this algorithm and see how well it does on the restaurant domain. This should give you an idea of the baseline for the domain—the minimal performance that any algorithm should be able to obtain.

18.7 Suppose you are running a learning experiment on a new algorithm. You have a data set consisting of 25 examples of each of two classes. You plan to use leave-one-out cross-validation. As a baseline, you run your experimental setup on a simple majority classifier. (A majority classifier is given a set of training data and then always outputs the class that is in the majority in the training set, regardless of the input.) You expect the majority classifier to score about 50% on leave-one-out cross-validation, but to your surprise, it scores zero. Can you explain why?

18.8 In the recursive construction of decision trees, it sometimes happens that a mixed set of positive and negative examples remains at a leaf node, even after all the attributes have been used. Suppose that we have p positive examples and n negative examples.

a. Show that the solution used by DECISION-TREE-LEARNING, which picks the majority classification, minimizes the absolute error over the set of examples at the leaf.

CLASS PROBABILITY

b. Show that the **class probability** $p/(p+n)$ minimizes the sum of squared errors.

18.9 Suppose that a learning algorithm is trying to find a consistent hypothesis when the classifications of examples are actually random. There are n Boolean attributes, and examples are drawn uniformly from the set of 2^n possible examples. Calculate the number of examples required before the probability of finding a contradiction in the data reaches 0.5.

18.10 Suppose that an attribute splits the set of examples E into subsets E_i and that each subset has p_i positive examples and n_i negative examples. Show that the attribute has strictly positive information gain unless the ratio $p_i/(p_i + n_i)$ is the same for all i.

18.11 Modify DECISION-TREE-LEARNING to include χ^2-pruning. You might wish to consult Quinlan (1986) for details.

18.12 The standard DECISION-TREE-LEARNING algorithm described in the chapter does not handle cases in which some examples have missing attribute values.

a. First, we need to find a way to classify such examples, given a decision tree that includes tests on the attributes for which values can be missing. Suppose that an example X has a missing value for attribute A and that the decision tree tests for A at a node that X reaches. One way to handle this case is to pretend that the example has *all* possible values for the attribute, but to weight each value according to its frequency among all of the examples that reach that node in the decision tree. The classification algorithm should follow all branches at any node for which a value is missing and should multiply the weights along each path. Write a modified classification algorithm for decision trees that has this behavior.

b. Now modify the information gain calculation so that in any given collection of examples C at a given node in the tree during the construction process, the examples with missing values for any of the remaining attributes are given "as-if" values according to the frequencies of those values in the set C.

18.13 In the chapter, we noted that attributes with many different possible values can cause problems with the gain measure. Such attributes tend to split the examples into numerous small classes or even singleton classes, thereby appearing to be highly relevant according to the gain measure. The **gain ratio** criterion selects attributes according to the ratio between their gain and their intrinsic information content—that is, the amount of information contained in the answer to the question, "What is the value of this attribute?" The gain ratio criterion therefore tries to measure how efficiently an attribute provides information on the correct classification of an example. Write a mathematical expression for the information content of an attribute, and implement the gain ratio criterion in DECISION-TREE-LEARNING.

18.14 Consider an ensemble learning algorithm that uses simple majority voting among M learned hypotheses. Suppose that each hypothesis has error ϵ and that the errors made by each hypothesis are independent of the others'. Calculate a formula for the error of the ensemble algorithm in terms of M and ϵ, and evaluate it for the cases where $M = 5$, 10, and 20 and $\epsilon = 0.1$, 0.2, and 0.4. If the independence assumption is removed, is it possible for the ensemble error to be *worse* than ϵ?

18.15 This exercise concerns the expressiveness of decision lists (Section 18.5).

a. Show that decision lists can represent any Boolean function, if the size of the tests is not limited.

b. Show that if the tests can contain at most k literals each, then decision lists can represent any function that can be represented by a decision tree of depth k.

19 KNOWLEDGE IN LEARNING

In which we examine the problem of learning when you know something already.

In all of the approaches to learning described in the previous three chapters, the idea is to construct a function that has the input/output behavior observed in the data. In each case, the learning methods can be understood as searching a hypothesis space to find a suitable function, starting from only a very basic assumption about the form of the function, such as "second degree polynomial" or "decision tree" and a bias such as "simpler is better." Doing this amounts to saying that before you can learn something new, you must first forget (almost) everything you know. In this chapter, we study learning methods that can take advantage of **prior knowledge** about the world. In most cases, the prior knowledge is represented as general first-order logical theories; thus for the first time we bring together the work on knowledge representation and learning.

PRIOR KNOWLEDGE

19.1 A LOGICAL FORMULATION OF LEARNING

Chapter 18 defined pure inductive learning as a process of finding a hypothesis that agrees with the observed examples. Here, we specialize this definition to the case where the hypothesis is represented by a set of logical sentences. Example descriptions and classifications will also be logical sentences, and a new example can be classified by inferring a classification sentence from the hypothesis and the example description. This approach allows for incremental construction of hypotheses, one sentence at a time. It also allows for prior knowledge, because sentences that are already known can assist in the classification of new examples. The logical formulation of learning may seem like a lot of extra work at first, but it turns out to clarify many of the issues in learning. It enables us to go well beyond the simple learning methods of Chapter 18 by using the full power of logical inference in the service of learning.

Examples and hypotheses

Recall from Chapter 18 the restaurant learning problem: learning a rule for deciding whether to wait for a table. Examples were described by **attributes** such as *Alternate*, *Bar*, *Fri/Sat*,

and so on. In a logical setting, an example is an object that is described by a logical sentence; the attributes become unary predicates. Let us generically call the ith example X_i. For instance, the first example from Figure 18.3 is described by the sentences

$$Alternate(X_1) \wedge \neg Bar(X_1) \wedge \neg Fri/Sat(X_1) \wedge Hungry(X_1) \wedge \ldots$$

We will use the notation $D_i(X_i)$ to refer to the description of X_i, where D_i can be any logical expression taking a single argument. The classification of the object is given by the sentence

$$WillWait(X_1) \ .$$

We will use the generic notation $Q(X_i)$ if the example is positive, and $\neg Q(X_i)$ if the example is negative. The complete training set is then just the conjunction of all the description and classification sentences.

The aim of inductive learning in the logical setting is to find an equivalent logical expression for the goal predicate Q that we can use to classify examples correctly. Each hypothesis proposes such an expression, which we call a **candidate definition** of the goal predicate. Using C_i to denote the candidate definition, each hypothesis H_i is a sentence of the form $\forall x \quad Q(x) \quad \Leftrightarrow \quad C_i(x)$. For example, a decision tree asserts that the goal predicate is true of an object if only if one of the branches leading to *true* is satisfied. Thus, the Figure 18.6 expresses the following logical definition (which we will call H_r for future reference):

CANDIDATE DEFINITION

$$\begin{aligned} \forall r \quad WillWait(r) \ \Leftrightarrow \ & Patrons(r, Some) \\ \vee \ & Patrons(r, Full) \wedge Hungry(r) \wedge Type(r, French) \\ \vee \ & Patrons(r, Full) \wedge Hungry(r) \wedge Type(r, Thai) \\ & \quad \wedge Fri/Sat(r) \\ \vee \ & Patrons(r, Full) \wedge Hungry(r) \wedge Type(r, Burger) \ . \end{aligned} \tag{19.1}$$

Each hypothesis predicts that a certain set of examples—namely, those that satisfy its candidate definition—will be examples of the goal predicate. This set is called the **extension** of the predicate. Two hypotheses with different extensions are therefore logically inconsistent with each other, because they disagree on their predictions for at least one example. If they have the same extension, they are logically equivalent.

EXTENSION

The hypothesis space **H** is the set of all hypotheses $\{H_1, \ldots, H_n\}$ that the learning algorithm is designed to entertain. For example, the DECISION-TREE-LEARNING algorithm can entertain any decision tree hypothesis defined in terms of the attributes provided; its hypothesis space therefore consists of all these decision trees. Presumably, the learning algorithm believes that one of the hypotheses is correct; that is, it believes the sentence

$$H_1 \vee H_2 \vee H_3 \vee \ldots \vee H_n \ . \tag{19.2}$$

As the examples arrive, hypotheses that are not **consistent** with the examples can be ruled out. Let us examine this notion of consistency more carefully. Obviously, if hypothesis H_i is consistent with the entire training set, it has to be consistent with each example. What would it mean for it to be inconsistent with an example? This can happen in one of two ways:

FALSE NEGATIVE

- An example can be a **false negative** for the hypothesis, if the hypothesis says it should be negative but in fact it is positive. For instance, the new example X_{13} described by $Patrons(X_{13}, Full) \wedge Wait(X_{13}, 0\text{-}10) \wedge \neg Hungry(X_{13}) \wedge \ldots \wedge WillWait(X_{13})$

would be a false negative for the hypothesis H_r given earlier. From H_r and the example description, we can deduce both $WillWait(X_{13})$, which is what the example says, and $\neg WillWait(X_{13})$, which is what the hypothesis predicts. The hypothesis and the example are therefore logically inconsistent.

FALSE POSITIVE

- An example can be a **false positive** for the hypothesis, if the hypothesis says it should be positive but in fact it is negative.[1]

If an example is a false positive or false negative for a hypothesis, then the example and the hypothesis are logically inconsistent with each other. Assuming that the example is a correct observation of fact, then the hypothesis can be ruled out. Logically, this is exactly analogous to the resolution rule of inference (see Chapter 9), where the disjunction of hypotheses corresponds to a clause and the example corresponds to a literal that resolves against one of the literals in the clause. An ordinary logical inference system therefore could, in principle, learn from the example by eliminating one or more hypotheses. Suppose, for example, that the example is denoted by the sentence I_1, and the hypothesis space is $H_1 \vee H_2 \vee H_3 \vee H_4$. Then if I_1 is inconsistent with H_2 and H_3, the logical inference system can deduce the new hypothesis space $H_1 \vee H_4$.

We therefore can characterize inductive learning in a logical setting as a process of gradually eliminating hypotheses that are inconsistent with the examples, narrowing down the possibilities. Because the hypothesis space is usually vast (or even infinite in the case of first-order logic), we do not recommend trying to build a learning system using resolution-based theorem proving and a complete enumeration of the hypothesis space. Instead, we will describe two approaches that find logically consistent hypotheses with much less effort.

Current-best-hypothesis search

CURRENT-BEST-HYPOTHESIS

The idea behind **current-best-hypothesis** search is to maintain a single hypothesis, and to adjust it as new examples arrive in order to maintain consistency. The basic algorithm was described by John Stuart Mill (1843), and may well have appeared even earlier.

Suppose we have some hypothesis such as H_r, of which we have grown quite fond. As long as each new example is consistent, we need do nothing. Then along comes a false negative example, X_{13}. What do we do? Figure 19.1(a) shows H_r schematically as a region: everything inside the rectangle is part of the extension of H_r. The examples that have actually been seen so far are shown as "+" or "–", and we see that H_r correctly categorizes all the examples as positive or negative examples of $WillWait$. In Figure 19.1(b), a new example (circled) is a false negative: the hypothesis says it should be negative but it is actually positive. The extension of the hypothesis must be increased to include it. This is called **generalization**; one possible generalization is shown in Figure 19.1(c). Then in Figure 19.1(d), we see a false positive: the hypothesis says the new example (circled) should be positive, but it actually is negative. The extension of the hypothesis must be decreased to exclude the example. This is called **specialization**; in Figure 19.1(e) we see one possible specialization of the hypothesis.

GENERALIZATION

SPECIALIZATION

[1] The terms "false positive" and "false negative" are used in medicine to describe erroneous results from lab tests. A result is a false positive if it indicates that the patient has the disease when in fact no disease is present.

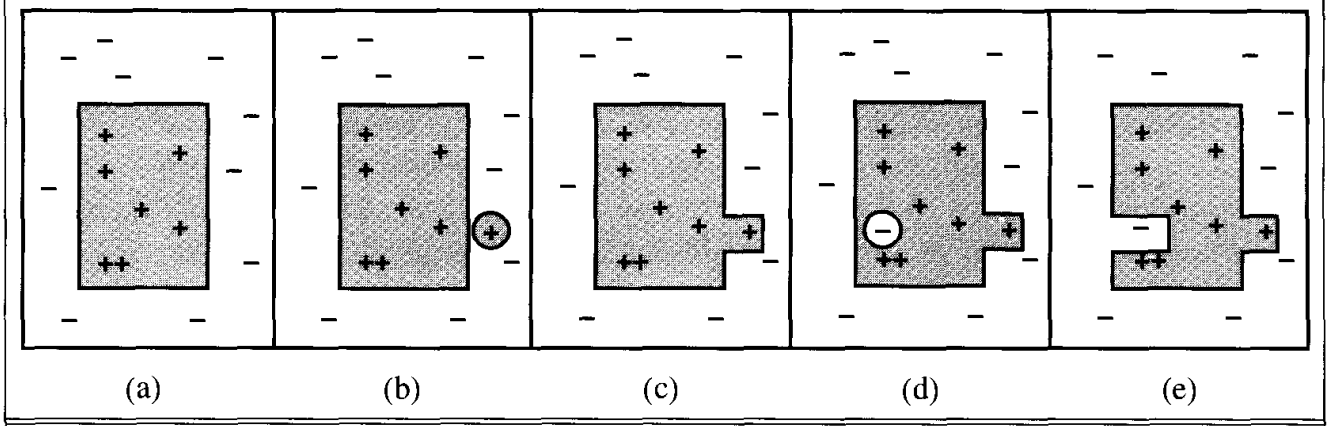


Figure 19.1 (a) A consistent hypothesis. (b) A false negative. (c) The hypothesis is generalized. (d) A false positive. (e) The hypothesis is specialized.

The "more general than" and "more specific than" relations between hypotheses provide the logical structure on the hypothesis space that makes efficient search possible.

We can now specify the CURRENT-BEST-LEARNING algorithm, shown in Figure 19.2. Notice that each time we consider generalizing or specializing the hypothesis, we must check for consistency with the other examples, because an arbitrary increase/decrease in the extension might include/exclude previously seen negative/positive examples.

```
function CURRENT-BEST-LEARNING(examples) returns a hypothesis

  H ← any hypothesis consistent with the first example in examples
  for each remaining example in examples do
     if e is false positive for H then
        H ← choose a specialization of H consistent with examples
     else if e is false negative for H then
        H ← choose a generalization of H consistent with examples
     if no consistent specialization/generalization can be found then fail
  return H
```

Figure 19.2 The current-best-hypothesis learning algorithm. It searches for a consistent hypothesis and backtracks when no consistent specialization/generalization can be found.

We have defined generalization and specialization as operations that change the *extension* of a hypothesis. Now we need to determine exactly how they can be implemented as syntactic operations that change the candidate definition associated with the hypothesis, so that a program can carry them out. This is done by first noting that generalization and specialization are also *logical* relationships between hypotheses. If hypothesis H_1, with definition C_1, is a generalization of hypothesis H_2 with definition C_2, then we must have

$$\forall x \;\; C_2(x) \;\Rightarrow\; C_1(x) \; .$$

Therefore in order to construct a generalization of H_2, we simply need to find a definition C_1 that is logically implied by C_2. This is easily done. For example, if $C_2(x)$ is

DROPPING CONDITIONS

$Alternate(x) \wedge Patrons(x, Some)$, then one possible generalization is given by $C_1(x) \equiv Patrons(x, Some)$. This is called **dropping conditions**. Intuitively, it generates a weaker definition and therefore allows a larger set of positive examples. There are a number of other generalization operations, depending on the language being operated on. Similarly, we can specialize a hypothesis by adding extra conditions to its candidate definition or by removing disjuncts from a disjunctive definition. Let us see how this works on the restaurant example, using the data in Figure 18.3.

- The first example X_1 is positive. $Alternate(X_1)$ is true, so let the initial hypothesis be

$$H_1 : \ \forall x \ \ WillWait(x) \Leftrightarrow Alternate(x) \ .$$

- The second example X_2 is negative. H_1 predicts it to be positive, so it is a false positive. Therefore, we need to specialize H_1. This can be done by adding an extra condition that will rule out X_2. One possibility is

$$H_2 : \ \forall x \ \ WillWait(x) \Leftrightarrow Alternate(x) \wedge Patrons(x, Some) \ .$$

- The third example X_3 is positive. H_2 predicts it to be negative, so it is a false negative. Therefore, we need to generalize H_2. We drop the $Alternate$ condition, yielding

$$H_3 : \ \forall x \ \ WillWait(x) \Leftrightarrow Patrons(x, Some) \ .$$

- The fourth example X_4 is positive. H_3 predicts it to be negative, so it is a false negative. We therefore need to generalize H_3. We cannot drop the $Patrons$ condition, because that would yield an all-inclusive hypothesis that would be inconsistent with X_2. One possibility is to add a disjunct:

$$\begin{aligned} H_4 : \ \forall x \ \ WillWait(x) \Leftrightarrow \ & Patrons(x, Some) \\ & \vee (Patrons(x, Full) \wedge Fri/Sat(x)) \ . \end{aligned}$$

Already, the hypothesis is starting to look reasonable. Obviously, there are other possibilities consistent with the first four examples; here are two of them:

$$H'_4 : \ \forall x \ \ WillWait(x) \Leftrightarrow \neg WaitEstimate(x, 30\text{-}60) \ .$$

$$\begin{aligned} H''_4 : \ \forall x \ \ WillWait(x) \Leftrightarrow \ & Patrons(x, Some) \\ & \vee (Patrons(x, Full) \wedge WaitEstimate(x, 10\text{-}30)) \ . \end{aligned}$$

The CURRENT-BEST-LEARNING algorithm is described nondeterministically, because at any point, there may be several possible specializations or generalizations that can be applied. The choices that are made will not necessarily lead to the simplest hypothesis, and may lead to an unrecoverable situation where no simple modification of the hypothesis is consistent with all of the data. In such cases, the program must backtrack to a previous choice point.

The CURRENT-BEST-LEARNING algorithm and its variants have been used in many machine learning systems, starting with Patrick Winston's (1970) "arch-learning" program. With a large number of instances and a large space, however, some difficulties arise:

1. Checking all the previous instances over again for each modification is very expensive.
2. The search process may involve a great deal of backtracking. As we saw in Chapter 18, hypothesis space can be a doubly exponentially large place.

Least-commitment search

Backtracking arises because the current-best-hypothesis approach has to *choose* a particular hypothesis as its best guess even though it does not have enough data yet to be sure of the choice. What we can do instead is to keep around all and only those hypotheses that are consistent with all the data so far. Each new instance will either have no effect or will get rid of some of the hypotheses. Recall that the original hypothesis space can be viewed as a disjunctive sentence

$$H_1 \vee H_2 \vee H_3 \ldots \vee H_n \ .$$

As various hypotheses are found to be inconsistent with the examples, this disjunction shrinks, retaining only those hypotheses not ruled out. Assuming that the original hypothesis space does in fact contain the right answer, the reduced disjunction must still contain the right answer because only incorrect hypotheses have been removed. The set of hypotheses remaining is called the **version space**, and the learning algorithm (sketched in Figure 19.3) is called the version space learning algorithm (also the **candidate elimination** algorithm).

VERSION SPACE

CANDIDATE ELIMINATION

```
function VERSION-SPACE-LEARNING(examples) returns a version space
  local variables: V, the version space: the set of all hypotheses

  V ← the set of all hypotheses
  for each example e in examples do
     if V is not empty then V ← VERSION-SPACE-UPDATE(V, e)
  return V
```

```
function VERSION-SPACE-UPDATE(V, e) returns an updated version space
  V ← {h ∈ V : h is consistent with e}
```

Figure 19.3 The version space learning algorithm. It finds a subset of V that is consistent with the *examples*.

One important property of this approach is that it is *incremental*: one never has to go back and reexamine the old examples. All remaining hypotheses are guaranteed to be consistent with them anyway. It is also a **least-commitment** algorithm because it makes no arbitrary choices (cf. the partial-order planning algorithm in Chapter 11). But there is an obvious problem. We already said that the hypothesis space is enormous, so how can we possibly write down this enormous disjunction?

The following simple analogy is very helpful. How do you represent all the real numbers between 1 and 2? After all, there is an infinite number of them! The answer is to use an interval representation that just specifies the boundaries of the set: [1,2]. It works because we have an *ordering* on the real numbers.

We also have an ordering on the hypothesis space, namely, generalization/specialization. This is a partial ordering, which means that each boundary will not be a point but rather a set of hypotheses called a **boundary set**. The great thing is that we can represent the entire

BOUNDARY SET

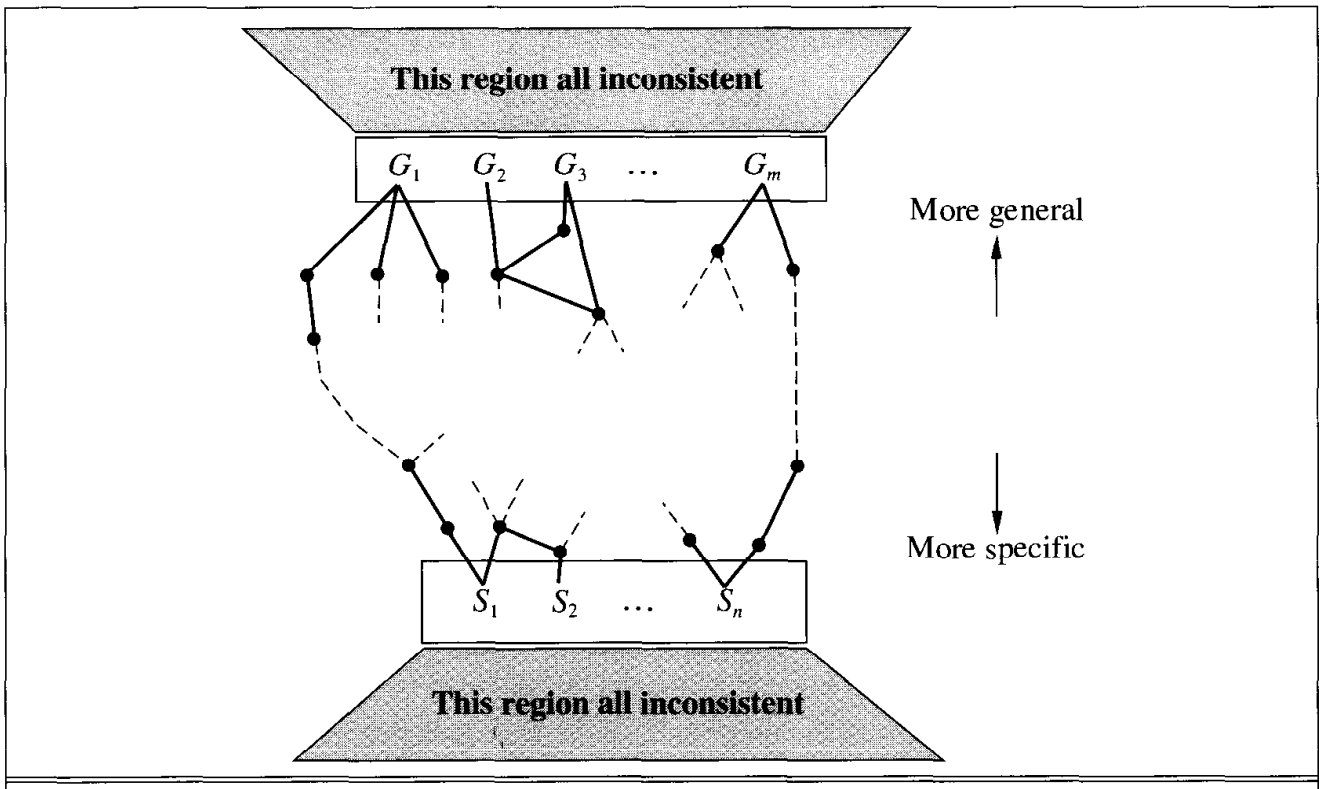


Figure 19.4 The version space contains all hypotheses consistent with the examples.

G-SET
S-SET

version space using just two boundary sets: a most general boundary (the **G-set**) and a most specific boundary (the **S-set**). *Everything in between is guaranteed to be consistent with the examples.* Before we prove this, let us recap:

- The current version space is the set of hypotheses consistent with all the examples so far. It is represented by the S-set and G-set, each of which is a set of hypotheses.
- Every member of the S-set is consistent with all observations so far, and there are no consistent hypotheses that are more specific.
- Every member of the G-set is consistent with all observations so far, and there are no consistent hypotheses that are more general.

We want the initial version space (before any examples have been seen) to represent all possible hypotheses. We do this by setting the G-set to contain *True* (the hypothesis that contains everything), and the S-set to contain *False* (the hypothesis whose extension is empty).

Figure 19.4 shows the general structure of the boundary set representation of the version space. To show that the representation is sufficient, we need the following two properties:

1. Every consistent hypothesis (other than those in the boundary sets) is more specific than some member of the G-set, and more general than some member of the S-set. (That is, there are no "stragglers" left outside.) This follows directly from the definitions of S and G. If there were a straggler h, then it would have to be no more specific than any member of G, in which case it belongs in G; or no more general than any member of S, in which case it belongs in S.
2. Every hypothesis more specific than some member of the G-set and more general than some member of the S-set is a consistent hypothesis. (That is, there are no "holes" be-

tween the boundaries.) Any h between S and G must reject all the negative examples rejected by each member of G (because it is more specific), and must accept all the positive examples accepted by any member of S (because it is more general). Thus, h must agree with all the examples, and therefore cannot be inconsistent. Figure 19.5 shows the situation: there are no known examples outside S but inside G, so any hypothesis in the gap must be consistent.

We have therefore shown that *if* S and G are maintained according to their definitions, then they provide a satisfactory representation of the version space. The only remaining problem is how to *update* S and G for a new example (the job of the VERSION-SPACE-UPDATE function). This may appear rather complicated at first, but from the definitions and with the help of Figure 19.4, it is not too hard to reconstruct the algorithm.

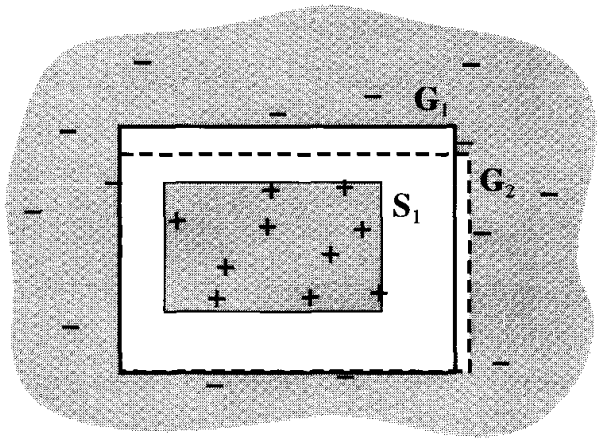


Figure 19.5 The extensions of the members of G and S. No known examples lie in between the two sets of boundaries.

We need to worry about the members S_i and G_i of the S- and G-sets. For each one, the new instance may be a false positive or a false negative.

1. False positive for S_i: This means S_i is too general, but there are no consistent specializations of S_i (by definition), so we throw it out of the S-set.
2. False negative for S_i: This means S_i is too specific, so we replace it by all its immediate generalizations, provided they are more specific than some member of G.
3. False positive for G_i: This means G_i is too general, so we replace it by all its immediate specializations, provided they are more general than some member of S.
4. False negative for G_i: This means G_i is too specific, but there are no consistent generalizations of G_i (by definition) so we throw it out of the G-set.

We continue these operations for each new instance until one of three things happens:

1. We have exactly one concept left in the version space, in which case we return it as the unique hypothesis.
2. The version space *collapses*—either S or G becomes empty, indicating that there are no consistent hypotheses for the training set. This is the same case as the failure of the simple version of the decision tree algorithm.

3. We run out of examples with several hypotheses remaining in the version space. This means the version space represents a disjunction of hypotheses. For any new example, if all the disjuncts agree, then we can return their classification of the example. If they disagree, one possibility is to take the majority vote.

We leave as an exercise the application of the VERSION-SPACE-LEARNING algorithm to the restaurant data.

There are two principal drawbacks to the version-space approach:

- If the domain contains noise or insufficient attributes for exact classification, the version space will always collapse.
- If we allow unlimited disjunction in the hypothesis space, the S-set will always contain a single most-specific hypothesis, namely, the disjunction of the descriptions of the positive examples seen to date. Similarly, the G-set will contain just the negation of the disjunction of the descriptions of the negative examples.
- For some hypothesis spaces, the number of elements in the S-set of G-set may grow exponentially in the number of attributes, even though efficient learning algorithms exist for those hypothesis spaces.

To date, no completely successful solution has been found for the problem of noise. The problem of disjunction can be addressed by allowing limited forms of disjunction or by including a **generalization hierarchy** of more general predicates. For example, instead of using the disjunction $WaitEstimate(x, 30\text{-}60) \vee WaitEstimate(x, >60)$, we might use the single literal $LongWait(x)$. The set of generalization and specialization operations can be easily extended to handle this.

GENERALIZATION HIERARCHY

The pure version space algorithm was first applied in the Meta-DENDRAL system, which was designed to learn rules for predicting how molecules would break into pieces in a mass spectrometer (Buchanan and Mitchell, 1978). Meta-DENDRAL was able to generate rules that were sufficiently novel to warrant publication in a journal of analytical chemistry—the first real scientific knowledge generated by a computer program. It was also used in the elegant LEX system (Mitchell *et al.*, 1983), which was able to learn to solve symbolic integration problems by studying its own successes and failures. Although version space methods are probably not practical in most real-world learning problems, mainly because of noise, they provide a good deal of insight into the logical structure of hypothesis space.

19.2 KNOWLEDGE IN LEARNING

The preceding section described the simplest setting for inductive learning. To understand the role of prior knowledge, we need to talk about the logical relationships among hypotheses, example descriptions, and classifications. Let *Descriptions* denote the conjunction of all the example descriptions in the training set, and let *Classifications* denote the conjunction of all the example classifications. Then a *Hypothesis* that "explains the observations" must satisfy

the following property (recall that $\models$ means "logically entails"):

$$Hypothesis \wedge Descriptions \models Classifications \ . \tag{19.3}$$

ENTAILMENT CONSTRAINT

We call this kind of relationship an **entailment constraint**, in which *Hypothesis* is the "unknown." Pure inductive learning means solving this constraint, where *Hypothesis* is drawn from some predefined hypothesis space. For example, if we consider a decision tree as a logical formula (see Equation (19.1) on page 679), then a decision tree that is consistent with all the examples will satisfy Equation (19.3). If we place *no* restrictions on the logical form of the hypothesis, of course, then *Hypothesis* = *Classifications* also satisfies the constraint. Ockham's razor tells us to prefer *small*, consistent hypotheses, so we try to do better than simply memorizing the examples.

This simple knowledge-free picture of inductive learning persisted until the early 1980s. The modern approach is to design agents that *already know something* and are trying to learn some more. This may not sound like a terrifically deep insight, but it makes quite a difference to the way we design agents. It might also have some relevance to our theories about how science itself works. The general idea is shown schematically in Figure 19.6.

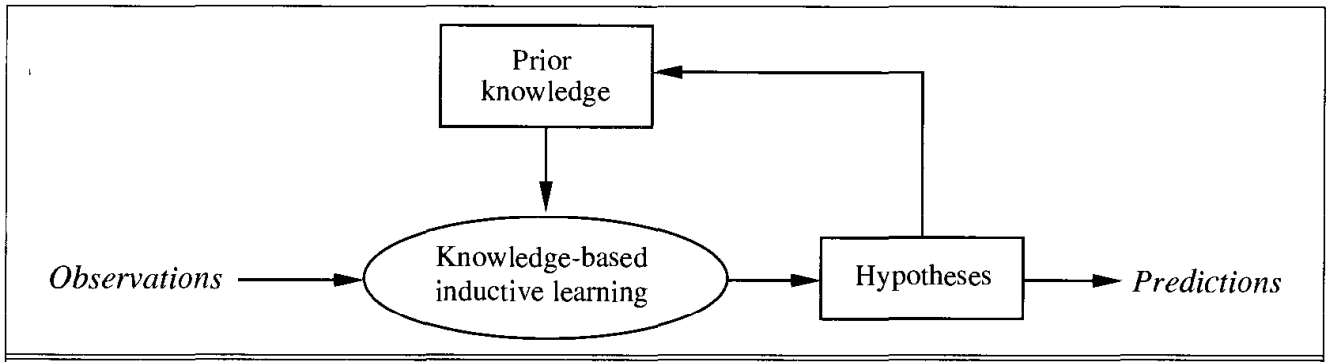


Figure 19.6 A cumulative learning process uses, and adds to, its stock of background knowledge over time.

If we want to build an autonomous learning agent that uses background knowledge, the agent must have some method for obtaining the background knowledge in the first place, in order for it to be used in the new learning episodes. This method must itself be a learning process. The agent's life history will therefore be characterized by *cumulative*, or *incremental*, development. Presumably, the agent could start out with nothing, performing inductions *in vacuo* like a good little pure induction program. But once it has eaten from the Tree of Knowledge, it can no longer pursue such naive speculations and should use its background knowledge to learn more and more effectively. The question is then how to actually do this.

Some simple examples

Let us consider some commonsense examples of learning with background knowledge. Many apparently rational cases of inferential behavior in the face of observations clearly do not follow the simple principles of pure induction.

- Sometimes one leaps to general conclusions after only one observation. Gary Larson once drew a cartoon in which a bespectacled caveman, Zog, is roasting his lizard on

the end of a pointed stick. He is watched by an amazed crowd of his less intellectual contemporaries, who have been using their bare hands to hold their victuals over the fire. This enlightening experience is enough to convince the watchers of a general principle of painless cooking.

- Or consider the case of the traveller to Brazil meeting her first Brazilian. On hearing him speak Portuguese, she immediately concludes that Brazilians speak Portuguese, yet on discovering that his name is Fernando, she does not conclude that all Brazilians are called Fernando. Similar examples appear in science. For example, when a freshman physics student measures the density and conductance of a sample of copper at a particular temperature, she is quite confident in generalizing those values to all pieces of copper. Yet when she measures its mass, she does not even consider the hypothesis that all pieces of copper have that mass. On the other hand, it would be quite reasonable to make such a generalization over all pennies.
- Finally, consider the case of a pharmacologically ignorant but diagnostically sophisticated medical student observing a consulting session between a patient and an expert internist. After a series of questions and answers, the expert tells the patient to take a course of a particular antibiotic. The medical student infers the general rule that that particular antibiotic is effective for a particular type of infection.

These are all cases in which *the use of background knowledge allows much faster learning than one might expect from a pure induction program.*

Some general schemes

In each of the preceding examples, one can appeal to prior knowledge to try to justify the generalizations chosen. We will now look at what kinds of entailment constraints are operating in each case. The constraints will involve the *Background* knowledge, in addition to the *Hypothesis* and the observed *Descriptions* and *Classifications*.

In the case of lizard toasting, the cavemen generalize by *explaining* the success of the pointed stick: it supports the lizard while keeping the hand away from the fire. From this explanation, they can infer a general rule: that any long, rigid, sharp object can be used to toast small, soft-bodied edibles. This kind of generalization process has been called **explanation-based learning**, or **EBL**. Notice that the general rule *follows logically* from the background knowledge possessed by the cavemen. Hence, the entailment constraints satisfied by EBL are the following:

EXPLANATION-BASED LEARNING

$$Hypothesis \wedge Descriptions \models Classifications$$
$$Background \models Hypothesis \ .$$

Because EBL uses Equation (19.3), it was initially thought to be a better way to learn from examples. But because it requires that the background knowledge be sufficient to explain the *Hypothesis*, which in turn explains the observations, *the agent does not actually learn anything factually new from the instance.* The agent *could have* derived the example from what it already knew, although that might have required an unreasonable amount of computation. EBL is now viewed as a method for converting first-principles theories into useful, special-purpose knowledge. We describe algorithms for EBL in Section 19.3.

The situation of our traveler in Brazil is quite different, for she cannot necessarily explain why Fernando speaks the way he does, unless she knows her Papal bulls. Moreover, the same generalization would be forthcoming from a traveler entirely ignorant of colonial history. The relevant prior knowledge in this case is that, within any given country, most people tend to speak the same language; on the other hand, Fernando is not assumed to be the name of all Brazilians because this kind of regularity does not hold for names. Similarly, the freshman physics student also would be hard put to explain the particular values that she discovers for the conductance and density of copper. She does know, however, that the material of which an object is composed and its temperature together determine its conductance.

RELEVANCE

In each case, the prior knowledge *Background* concerns the **relevance** of a set of features to the goal predicate. This knowledge, *together with the observations*, allows the agent to infer a new, general rule that explains the observations:

$$\begin{aligned} &Hypothesis \wedge Descriptions \models Classifications \ , \\ &Background \wedge Descriptions \wedge Classifications \models Hypothesis \ . \end{aligned} \tag{19.4}$$

RELEVANCE-BASED LEARNING

We call this kind of generalization **relevance-based learning**, or **RBL** (although the name is not standard). Notice that whereas RBL does make use of the content of the observations, it does not produce hypotheses that go beyond the logical content of the background knowledge and the observations. It is a *deductive* form of learning and cannot by itself account for the creation of new knowledge starting from scratch.

In the case of the medical student watching the expert, we assume that the student's prior knowledge is sufficient to infer the patient's disease D from the symptoms. This is not, however, enough to explain the fact that the doctor prescribes a particular medicine M. The student needs to propose another rule, namely, that M generally is effective against D. Given this rule and the student's prior knowledge, the student can now explain why the expert prescribes M in this particular case. We can generalize this example to come up with the entailment constraint:

$$Background \wedge Hypothesis \wedge Descriptions \models Classifications \ . \tag{19.5}$$

That is, *the background knowledge and the new hypothesis combine to explain the examples.* As with pure inductive learning, the learning algorithm should propose hypotheses that are as simple as possible, consistent with this constraint. Algorithms that satisfy constraint (19.5) are called **knowledge-based inductive learning**, or **KBIL**, algorithms.

KNOWLEDGE-BASED INDUCTIVE LEARNING

KBIL algorithms, which are described in detail in Section 19.5, have been studied mainly in the field of **inductive logic programming**, or **ILP**. In ILP systems, prior knowledge plays two key roles in reducing the complexity of learning:

INDUCTIVE LOGIC PROGRAMMING

1. Because any hypothesis generated must be consistent with the prior knowledge as well as with the new observations, the effective hypothesis space size is reduced to include only those theories that are consistent with what is already known.
2. For any given set of observations, the size of the hypothesis required to construct an explanation for the observations can be much reduced, because the prior knowledge will be available to help out the new rules in explaining the observations. The smaller the hypothesis, the easier it is to find.

In addition to allowing the use of prior knowledge in induction, ILP systems can formulate hypotheses in general first-order logic, rather than in the restricted attribute-based language of Chapter 18. This means that they can learn in environments that cannot be understood by simpler systems.

19.3 EXPLANATION-BASED LEARNING

As we explained in the introduction to this chapter, explanation-based learning is a method for extracting general rules from individual observations. As an example, consider the problem of differentiating and simplifying algebraic expressions (Exercise 9.15). If we differentiate an expression such as X^2 with respect to X, we obtain $2X$. (Notice that we use a capital letter for the arithmetic unknown X, to distinguish it from the logical variable x.) In a logical reasoning system, the goal might be expressed as $\text{ASK}(Derivative(X^2, X) = d,\ KB)$, with solution $d = 2X$.

Anyone who knows differential calculus can see this solution "by inspection" as a result of practice in solving such problems. A student encountering such problems for the first time, or a program with no experience, will have a much more difficult job. Application of the standard rules of differentiation eventually yields the expression $1 \times (2 \times (X^{(2-1)}))$, and eventually this simplifies to $2X$. In the authors' logic programming implementation, this takes 136 proof steps, of which 99 are on dead-end branches in the proof. After such an experience, we would like the program to solve the same problem much more quickly the next time it arises..

MEMOIZATION

The technique of **memoization** has long been used in computer science to speed up programs by saving the results of computation. The basic idea of memo functions is to accumulate a database of input/output pairs; when the function is called, it first checks the database to see whether it can avoid solving the problem from scratch. Explanation-based learning takes this a good deal further, by creating *general* rules that cover an entire class of cases. In the case of differentiation, memoization would remember that the derivative of X^2 with respect to X is $2X$, but would leave the agent to calculate the derivative of Z^2 with respect to Z from scratch. We would like to be able to extract the general rule[2] that for any arithmetic unknown u, the derivative of u^2 with respect to u is $2u$. In logical terms, this is expressed by the rule

$$ArithmeticUnknown(u) \Rightarrow Derivative(u^2, u) = 2u\ .$$

If the knowledge base contains such a rule, then any new case that is an instance of this rule can be solved immediately.

This is, of course, merely a trivial example of a very general phenomenon. Once something is understood, it can be generalized and reused in other circumstances. It becomes an "obvious" step and can then be used as a building block in solving problems still more complex. Alfred North Whitehead (1911), co-author with Bertrand Russell of *Principia Mathe-*

[2] Of course, a general rule for u^n can also be produced, but the current example suffices to make the point.

matica, wrote *"Civilization advances by extending the number of important operations that we can do without thinking about them,"* perhaps himself applying EBL to his understanding of events such as Zog's discovery. If you have understood the basic idea of the differentiation example, then your brain is already busily trying to extract the general principles of explanation-based learning from it. Notice that you hadn't *already* invented EBL before you saw the example. Like the cavemen watching Zog, you (and we) needed an example before we could generate the basic principles. This is because *explaining why* something is a good idea is much easier than coming up with the idea in the first place.

Extracting general rules from examples

The basic idea behind EBL is first to construct an explanation of the observation using prior knowledge, and then to establish a definition of the class of cases for which the same explanation structure can be used. This definition provides the basis for a rule covering all of the cases in the class. The "explanation" can be a logical proof, but more generally it can be any reasoning or problem-solving process whose steps are well defined. The key is to be able to identify the necessary conditions for those same steps to apply to another case.

We will use for our reasoning system the simple backward-chaining theorem prover described in Chapter 9. The proof tree for $Derivative(X^2, X) = 2X$ is too large to use as an example, so we will use a simpler problem to illustrate the generalization method. Suppose our problem is to simplify $1 \times (0 + X)$. The knowledge base includes the following rules:

$$
\begin{aligned}
&Rewrite(u, v) \wedge Simplify(v, w) \Rightarrow Simplify(u, w) .\\
&Primitive(u) \Rightarrow Simplify(u, u) .\\
&ArithmeticUnknown(u) \Rightarrow Primitive(u) .\\
&Number(u) \Rightarrow Primitive(u) .\\
&Rewrite(1 \times u, u) .\\
&Rewrite(0 + u, u) .\\
&\quad\vdots
\end{aligned}
$$

The proof that the answer is X is shown in the top half of Figure 19.7. The EBL method actually constructs two proof trees simultaneously. The second proof tree uses a *variabilized* goal in which the constants from the original goal are replaced by variables. As the original proof proceeds, the variabilized proof proceeds in step, using *exactly the same rule applications*. This could cause some of the variables to become instantiated. For example, in order to use the rule $Rewrite(1 \times u, u)$, the variable x in the subgoal $Rewrite(x \times (y + z), v)$ must be bound to 1. Similarly, y must be bound to 0 in the subgoal $Rewrite(y + z, v')$ in order to use the rule $Rewrite(0 + u, u)$. Once we have the generalized proof tree, we take the leaves (with the necessary bindings) and form a general rule for the goal predicate:

$$
\begin{aligned}
&Rewrite(1 \times (0 + z), 0 + z) \wedge Rewrite(0 + z, z) \wedge ArithmeticUnknown(z)\\
&\quad \Rightarrow Simplify(1 \times (0 + z), z) .
\end{aligned}
$$

Notice that the first two conditions on the left-hand side are true *regardless of the value of* z. We can therefore drop them from the rule, yielding

$$ArithmeticUnknown(z) \Rightarrow Simplify(1 \times (0 + z), z) .$$

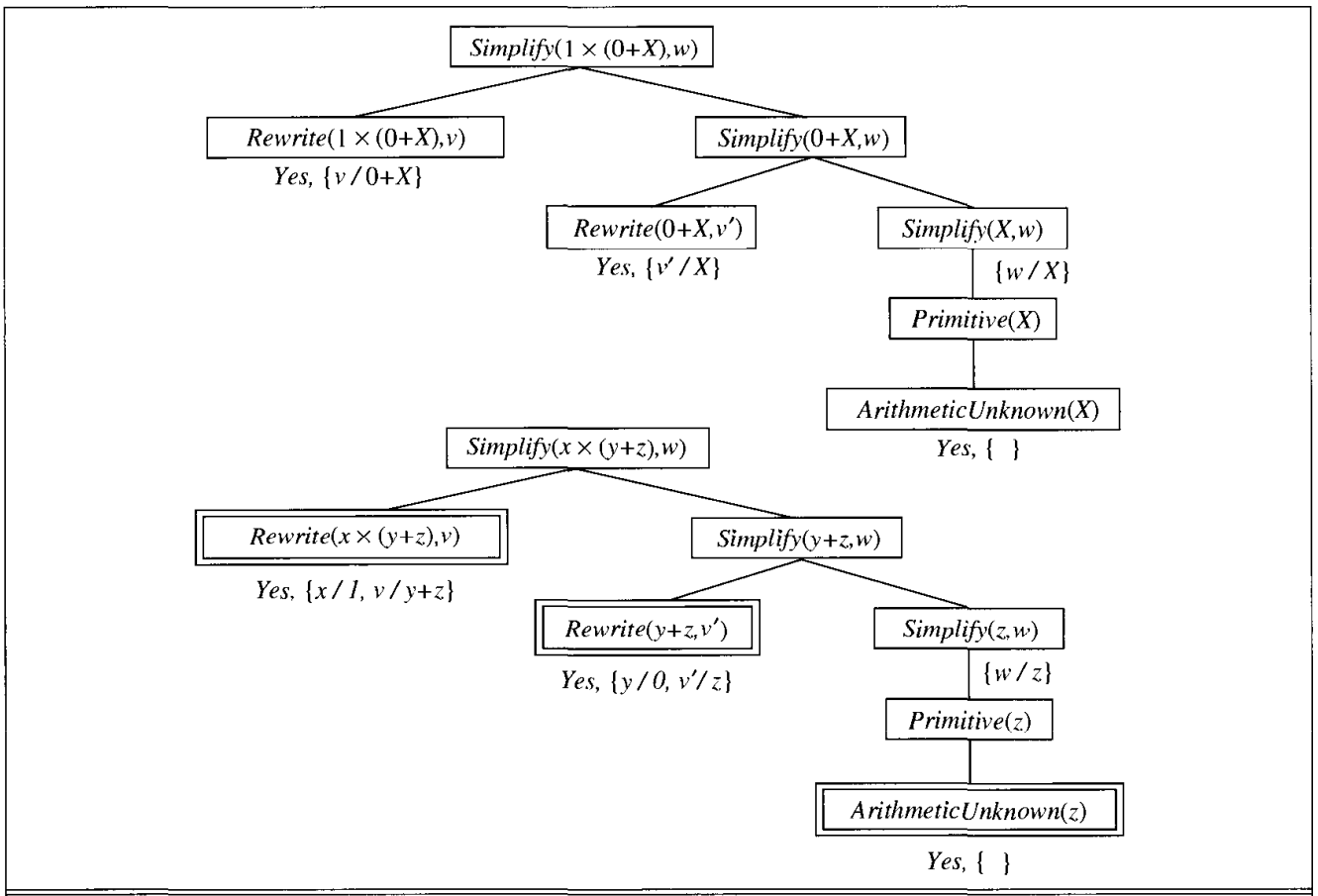


Figure 19.7 Proof trees for the simplification problem. The first tree shows the proof for the original problem instance, from which we can derive

$$ArithmeticUnknown(z) \Rightarrow Simplify(1 \times (0 + z), z) .$$

The second shows the proof for a problem instance with all constants replaced by variables, from which we can derive a variety of other rules.

In general, conditions can be dropped from the final rule if they impose no constraints on the variables on the right-hand side of the rule, because the resulting rule will still be true and will be more efficient. Notice that we cannot drop the condition $ArithmeticUnknown(z)$, because not all possible values of z are arithmetic unknowns. Values other than arithmetic unknowns might require different forms of simplification: for example, if z were 2×3, then the correct simplification of $1 \times (0 + (2 \times 3))$ would be 6 and not 2×3.

To recap, the basic EBL process works as follows:

1. Given an example, construct a proof that the goal predicate applies to the example using the available background knowledge.
2. In parallel, construct a generalized proof tree for the variabilized goal using the same inference steps as in the original proof.
3. Construct a new rule whose left-hand side consists of the leaves of the proof tree and whose right-hand side is the variabilized goal (after applying the necessary bindings from the generalized proof).
4. Drop any conditions that are true regardless of the values of the variables in the goal.

Improving efficiency

The generalized proof tree in Figure 19.7 actually yields more than one generalized rule. For example, if we terminate, or **prune**, the growth of the right-hand branch in the proof tree when it reaches the *Primitive* step, we get the rule

$$Primitive(z) \Rightarrow Simplify(1 \times (0 + z), z) .$$

This rule is as valid as, but *more general* than, the rule using *ArithmeticUnknown*, because it covers cases where z is a number. We can extract a still more general rule by pruning after the step $Simplify(y + z, w)$, yielding the rule

$$Simplify(y + z, w) \Rightarrow Simplify(1 \times (y + z), w) .$$

In general, a rule can be extracted from *any partial subtree* of the generalized proof tree. Now we have a problem: which of these rules do we choose?

The choice of which rule to generate comes down to the question of efficiency. There are three factors involved in the analysis of efficiency gains from EBL:

1. Adding large numbers of rules can slow down the reasoning process, because the inference mechanism must still check those rules even in cases where they do not yield a solution. In other words, it increases the **branching factor** in the search space.
2. To compensate for the slowdown in reasoning, the derived rules must offer significant increases in speed for the cases that they do cover. These increases come about mainly because the derived rules avoid dead ends that would otherwise be taken, but also because they shorten the proof itself.
3. Derived rules should be as general as possible, so that they apply to the largest possible set of cases.

OPERATIONALITY

A common approach to ensuring that derived rules are efficient is to insist on the **operationality** of each subgoal in the rule. A subgoal is operational if it is "easy" to solve. For example, the subgoal $Primitive(z)$ is easy to solve, requiring at most two steps, whereas the subgoal $Simplify(y + z, w)$ could lead to an arbitrary amount of inference, depending on the values of y and z. If a test for operationality is carried out at each step in the construction of the generalized proof, then we can prune the rest of a branch as soon as an operational subgoal is found, keeping just the operational subgoal as a conjunct of the new rule.

Unfortunately, there is usually a tradeoff between operationality and generality. More specific subgoals are generally easier to solve but cover fewer cases. Also, operationality is a matter of degree: one or 2 steps is definitely operational, but what about 10 or 100? Finally, the cost of solving a given subgoal depends on what other rules are available in the knowledge base. It can go up or down as more rules are added. Thus, EBL systems really face a very complex optimization problem in trying to maximize the efficiency of a given initial knowledge base. It is sometimes possible to derive a mathematical model of the effect on overall efficiency of adding a given rule and to use this model to select the best rule to add. The analysis can become very complicated, however, especially when recursive rules are involved. One promising approach is to address the problem of efficiency empirically, simply by adding several rules and seeing which ones are useful and actually speed things up.

Empirical analysis of efficiency is actually at the heart of EBL. What we have been calling loosely the "efficiency of a given knowledge base" is actually the average-case complexity on a distribution of problems. *By generalizing from past example problems, EBL makes the knowledge base more efficient for the kind of problems that it is reasonable to expect.* This works as long as the distribution of past examples is roughly the same as for future examples—the same assumption used for PAC-learning in Section 18.5. If the EBL system is carefully engineered, it is possible to obtain significant speedups. For example, a very large Prolog-based natural language system designed for speech-to-speech translation between Swedish and English was able to achieve real-time performance only by the application of EBL to the parsing process (Samuelsson and Rayner, 1991).

19.4 Learning Using Relevance Information

Our traveler in Brazil seems to be able to make a confident generalization concerning the language spoken by other Brazilians. The inference is sanctioned by her background knowledge, namely, that people in a given country (usually) speak the same language. We can express this in first-order logic as follows:[3]

$$Nationality(x,n) \wedge Nationality(y,n) \wedge Language(x,l) \Rightarrow Language(y,l) \ . \quad (19.6)$$

(Literal translation: "If x and y have the same nationality n and x speaks language l, then y also speaks it.") It is not difficult to show that, from this sentence and the observation that

$$Nationality(Fernando, Brazil) \wedge Language(Fernando, Portuguese) \ ,$$

the following conclusion is entailed (see Exercise 19.1):

$$Nationality(x, Brazil) \Rightarrow Language(x, Portuguese) \ .$$

Sentences such as (19.6) express a strict form of relevance: given nationality, language is fully determined. (Put another way: language is a function of nationality.) These sentences are called **functional dependencies** or **determinations**. They occur so commonly in certain kinds of applications (e.g., defining database designs) that a special syntax is used to write them. We adopt the notation of Davies (1985):

FUNCTIONAL DEPENDENCIES

DETERMINATIONS

$$Nationality(x,n) \succ Language(x,l) \ .$$

As usual, this is simply a syntactic sugaring, but it makes it clear that the determination is really a relationship between the predicates: nationality determines language. The relevant properties determining conductance and density can be expressed similarly:

$$Material(x,m) \wedge Temperature(x,t) \succ Conductance(x,\rho) \ ;$$
$$Material(x,m) \wedge Temperature(x,t) \succ Density(x,d) \ .$$

The corresponding generalizations follow logically from the determinations and observations.

[3] We assume for the sake of simplicity that a person speaks only one language. Clearly, the rule also would have to be amended for countries such as Switzerland and India.

Determining the hypothesis space

Although the determinations sanction general conclusions concerning all Brazilians, or all pieces of copper at a given temperature, they cannot, of course, yield a general predictive theory for *all* nationalities, or for *all* temperatures and materials, from a single example. Their main effect can be seen as limiting the space of hypotheses that the learning agent need consider. In predicting conductance, for example, one need consider only material and temperature and can ignore mass, ownership, day of the week, the current president, and so on. Hypotheses can certainly include terms that are in turn determined by material and temperature, such as molecular structure, thermal energy, or free-electron density. *Determinations specify a sufficient basis vocabulary from which to construct hypotheses concerning the target predicate.* This statement can be proven by showing that a given determination is logically equivalent to a statement that the correct definition of the target predicate is one of the set of all definitions expressible using the predicates on the left-hand side of the determination.

Intuitively, it is clear that a reduction in the hypothesis space size should make it easier to learn the target predicate. Using the basic results of computational learning theory (Section 18.5), we can quantify the possible gains. First, recall that for Boolean functions, $\log(|\mathbf{H}|)$ examples are required to converge to a reasonable hypothesis, where $|\mathbf{H}|$ is the size of the hypothesis space. If the learner has n Boolean features with which to construct hypotheses, then, in the absence of further restrictions, $|\mathbf{H}| = O(2^{2^n})$, so the number of examples is $O(2^n)$. If the determination contains d predicates in the left-hand side, the learner will require only $O(2^d)$ examples, a reduction of $O(2^{n-d})$. For biased hypothesis spaces, such as a conjunctively biased space, the reduction will be less dramatic, but still significant.

Learning and using relevance information

As we stated in the introduction to this chapter, prior knowledge is useful in learning, but it too has to be learned. In order to provide a complete story of relevance-based learning, we must therefore provide a learning algorithm for determinations. The learning algorithm we now present is based on a straightforward attempt to find the simplest determination consistent with the observations. A determination $P \succ Q$ says that if any examples match on P, then they must also match on Q. A determination is therefore consistent with a set of examples if every pair that matches on the predicates on the left-hand side also matches on the target predicate—that is, has the same classification. For example, suppose we have the following examples of conductance measurements on material samples:

Sample	Mass	Temperature	Material	Size	Conductance
S1	12	26	Copper	3	0.59
S1	12	100	Copper	3	0.57
S2	24	26	Copper	6	0.59
S3	12	26	Lead	2	0.05
S3	12	100	Lead	2	0.04
S4	24	26	Lead	4	0.05

```
function MINIMAL-CONSISTENT-DET(E, A) returns a set of attributes
  inputs: E, a set of examples
          A, a set of attributes, of size n

  for i ← 0, ..., n do
     for each subset A_i of A of size i do
        if CONSISTENT-DET?(A_i, E) then return A_i
```

```
function CONSISTENT-DET?(A, E) returns a truth-value
  inputs: A, a set of attributes
          E, a set of examples
  local variables: H, a hash table

  for each example e in E do
     if some example in H has the same values as e for the attributes A
       but a different classification then return false
     store the class of e in H, indexed by the values for attributes A of the example e
  return true
```

Figure 19.8 An algorithm for finding a minimal consistent determination.

The minimal consistent determination is $Material \wedge Temperature \succ Conductance$. There is a nonminimal but consistent determination, namely, $Mass \wedge Size \wedge Temperature \succ Conductance$. This is consistent with the examples because mass and size determine density and, in our data set, we do not have two different materials with the same density. As usual, we would need a larger sample set in order to eliminate a nearly correct hypothesis.

There are several possible algorithms for finding minimal consistent determinations. The most obvious approach is to conduct a search through the space of determinations, checking all determinations with one predicate, two predicates, and so on, until a consistent determination is found. We will assume a simple attribute-based representation, like that used for decision-tree learning in Chapter 18. A determination d will be represented by the set of attributes on the left-hand side, because the target predicate is assumed fixed. The basic algorithm is outlined in Figure 19.8.

The time complexity of this algorithm depends on the size of the smallest consistent determination. Suppose this determination has p attributes out of the n total attributes. Then the algorithm will not find it until searching the subsets of A of size p. There are $\binom{n}{p} = O(n^p)$ such subsets; hence the algorithm is exponential in the size of the minimal determination. It turns out that the problem is NP-complete, so we cannot expect to do better in the general case. In most domains, however, there will be sufficient local structure (see Chapter 14 for a definition of locally structured domains) that p will be small.

Given an algorithm for learning determinations, a learning agent has a way to construct a minimal hypothesis within which to learn the target predicate. For example, we can combine MINIMAL-CONSISTENT-DET with the DECISION-TREE-LEARNING algorithm. This yields a relevance-based decision-tree learning algorithm RBDTL that first identifies a minimal

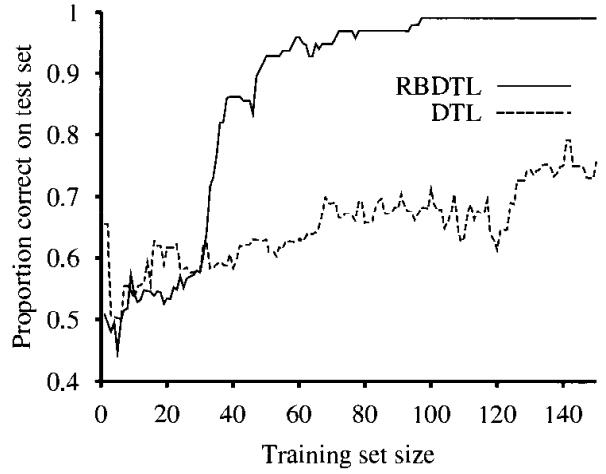


Figure 19.9 A performance comparison between RBDTL and DECISION-TREE-LEARNING on randomly generated data for a target function that depends on only 5 of 16 attributes.

set of relevant attributes and then passes this set to the decision tree algorithm for learning. Unlike DECISION-TREE-LEARNING, RBDTL simultaneously learns and uses relevance information in order to minimize its hypothesis space. We expect that RBDTL will learn faster than DECISION-TREE-LEARNING, and this is in fact the case. Figure 19.9 shows the learning performance for the two algorithms on randomly generated data for a function that depends on only 5 of 16 attributes. Obviously, in cases where all the available attributes are relevant, RBDTL will show no advantage.

DECLARATIVE BIAS

This section has only scratched the surface of the field of **declarative bias**, which aims to understand how prior knowledge can be used to identify the appropriate hypothesis space within which to search for the correct target definition. There are many unanswered questions:

- How can the algorithms be extended to handle noise?
- Can we handle continuous-valued variables?
- How can other kinds of prior knowledge be used, besides determinations?
- How can the algorithms be generalized to cover any first-order theory, rather than just an attribute-based representation?

Some of these questions are addressed in the next section.

19.5 INDUCTIVE LOGIC PROGRAMMING

Inductive logic programming (ILP) combines inductive methods with the power of first-order representations, concentrating in particular on the representation of theories as logic programs.[4] It has gained popularity for three reasons. First, ILP offers a rigorous approach to

[4] It might be appropriate at this point for the reader to refer to Chapter 9 for some of the underlying concepts, including Horn clauses, conjunctive normal form, unification, and resolution.

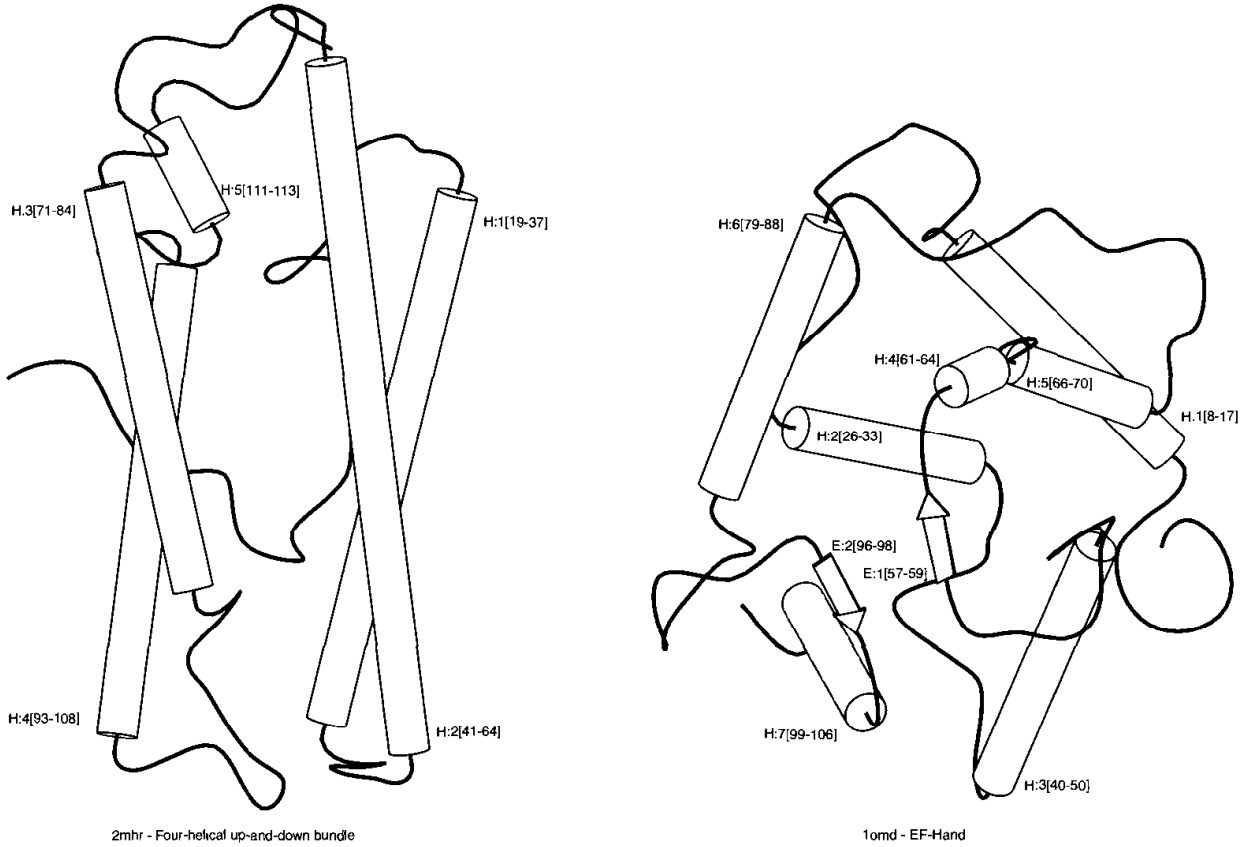


Figure 19.10 (a) and (b) show positive and negative examples, respectively, of the "four-helical up-and-down bundle" concept in the domain of protein folding. Each example structure is coded into a logical expression of about 100 conjuncts such as $TotalLength(D2mhr, 118) \wedge NumberHelices(D2mhr, 6) \wedge \ldots$. From these descriptions and from classifications such as $Fold(\text{FOUR-HELICAL-UP-AND-DOWN-BUNDLE}, D2mhr)$, the inductive logic programming system PROGOL (Muggleton, 1995) learned the following rule:

$$Fold(\text{FOUR-HELICAL-UP-AND-DOWN-BUNDLE}, p) \Leftarrow$$
$$Helix(p, h_1) \wedge Length(h_1, \text{HIGH}) \wedge Position(p, h_1, n)$$
$$\wedge\ (1 \leq n \leq 3) \wedge Adjacent(p, h_1, h_2) \wedge Helix(p, h_2)\ .$$

This kind of rule could not be learned, or even represented, by an attribute-based mechanism such as we saw in previous chapters. The rule can be translated into English as

> The protein P has fold class "Four helical up and down bundle" if it contains a long helix h_1 at a secondary structure position between 1 and 3 and h_1 is next to a second helix.

the general knowledge-based inductive learning problem. Second, it offers complete algorithms for inducing general, first-order theories from examples, which can therefore learn successfully in domains where attribute-based algorithms are hard to apply. An example is in learning how protein structures fold (Figure 19.10). The three-dimensional configuration of a protein molecule cannot be represented reasonably by a set of attributes, because the configuration inherently refers to *relationships* between objects, not to attributes of a single

object. First-order logic is an appropriate language for describing the relationships. Third, inductive logic programming produces hypotheses that are (relatively) easy for humans to read. For example, the English translation in Figure 19.10 can be scrutinized and criticized by working biologists. This means that inductive logic programming systems can participate in the scientific cycle of experimentation, hypothesis generation, debate, and refutation. Such participation would not be possible for systems that generate "black-box" classifiers, such as neural networks.

An example

Recall from Equation (19.5) that the general knowledge-based induction problem is to "solve" the entailment constraint

$$Background \wedge Hypothesis \wedge Descriptions \models Classifications$$

for the unknown *Hypothesis*, given the *Background* knowledge and examples described by *Descriptions* and *Classifications*. To illustrate this, we will use the problem of learning family relationships from examples. The descriptions will consist of an extended family tree, described in terms of *Mother*, *Father*, and *Married* relations and *Male* and *Female* properties. As an example, we will use the family tree from Exercise 8.11, shown here in Figure 19.11. The corresponding descriptions are as follows:

$$\begin{array}{lll} Father(Philip, Charles) & Father(Philip, Anne) & \ldots \\ Mother(Mum, Margaret) & Mother(Mum, Elizabeth) & \ldots \\ Married(Diana, Charles) & Married(Elizabeth, Philip) & \ldots \\ Male(Philip) & Male(Charles) & \ldots \\ Female(Beatrice) & Female(Margaret) & \ldots \end{array}$$

The sentences in *Classifications* depend on the target concept being learned. We might want to learn *Grandparent*, *BrotherInLaw*, or *Ancestor*, for example. For *Grandparent*, the complete set of *Classifications* contains $20 \times 20 = 400$ conjuncts of the form

$$\begin{array}{lll} Grandparent(Mum, Charles) & Grandparent(Elizabeth, Beatrice) & \ldots \\ \neg Grandparent(Mum, Harry) & \neg Grandparent(Spencer, Peter) & \ldots \end{array}$$

We could of course learn from a subset of this complete set.

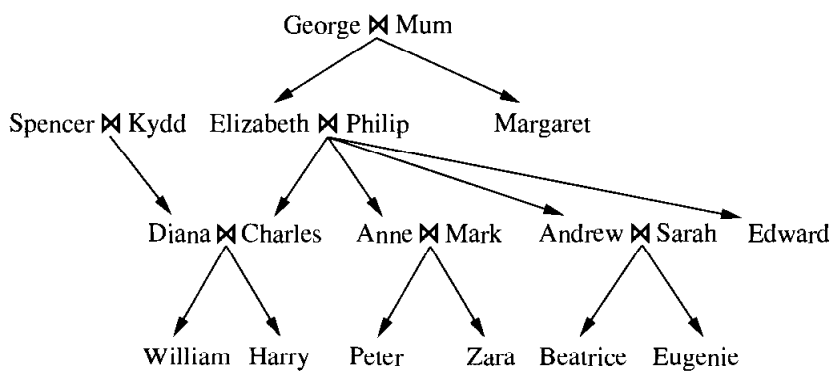


Figure 19.11 A typical family tree.

The object of an inductive learning program is to come up with a set of sentences for the *Hypothesis* such that the entailment constraint is satisfied. Suppose, for the moment, that the agent has no background knowledge: *Background* is empty. Then one possible solution for *Hypothesis* is the following:

$$\begin{array}{lll} Grandparent(x,y) & \Leftrightarrow & [\exists\, z\;\; Mother(x,z) \wedge Mother(z,y)] \\ & \vee & [\exists\, z\;\; Mother(x,z) \wedge Father(z,y)] \\ & \vee & [\exists\, z\;\; Father(x,z) \wedge Mother(z,y)] \\ & \vee & [\exists\, z\;\; Father(x,z) \wedge Father(z,y)]\ . \end{array}$$

Notice that an attribute-based learning algorithm, such as DECISION-TREE-LEARNING, will get nowhere in solving this problem. In order to express *Grandparent* as an attribute (i.e., a unary predicate), we would need to make *pairs* of people into objects:

$$Grandparent(\langle Mum,\ Charles\rangle)\ldots$$

Then we get stuck in trying to represent the example descriptions. The only possible attributes are horrible things such as

$$FirstElementIsMotherOfElizabeth(\langle Mum,\ Charles\rangle)\ .$$

The definition of *Grandparent* in terms of these attributes simply becomes a large disjunction of specific cases that does not generalize to new examples at all. *Attribute-based learning algorithms are incapable of learning relational predicates.* Thus, one of the principal advantages of ILP algorithms is their applicability to a much wider range of problems, including relational problems.

The reader will certainly have noticed that a little bit of background knowledge would help in the representation of the *Grandparent* definition. For example, if *Background* included the sentence

$$Parent(x,y) \Leftrightarrow [Mother(x,y) \vee Father(x,y)]\ ,$$

then the definition of *Grandparent* would be reduced to

$$Grandparent(x,y) \Leftrightarrow [\exists\, z\;\; Parent(x,z) \wedge Parent(z,y)]\ .$$

This shows how background knowledge can dramatically reduce the size of hypotheses required to explain the observations.

It is also possible for ILP algorithms to *create* new predicates in order to facilitate the expression of explanatory hypotheses. Given the example data shown earlier, it is entirely reasonable for the ILP program to propose an additional predicate, which we would call "*Parent*," in order to simplify the definitions of the target predicates. Algorithms that can generate new predicates are called **constructive induction** algorithms. Clearly, constructive induction is a necessary part of the picture of cumulative learning sketched in the introduction. It has been one of the hardest problems in machine learning, but some ILP techniques provide effective mechanisms for achieving it.

CONSTRUCTIVE INDUCTION

In the rest of this chapter, we will study the two principal approaches to ILP. The first uses a generalization of decision-tree methods, and the second uses techniques based on inverting a resolution proof.

Top-down inductive learning methods

The first approach to ILP works by starting with a very general rule and gradually specializing it so that it fits the data. This is essentially what happens in decision-tree learning, where a decision tree is gradually grown until it is consistent with the observations. To do ILP we use first-order literals instead of attributes, and the hypothesis is a set of clauses instead of a decision tree. This section describes FOIL (Quinlan, 1990), one of the first ILP programs.

Suppose we are trying to learn a definition of the $Grandfather(x, y)$ predicate, using the same family data as before. As with decision-tree learning, we can divide the examples into positive and negative examples. Positive examples are

$$\langle George, Anne\rangle,\ \langle Philip, Peter\rangle,\ \langle Spencer, Harry\rangle,\ \ldots$$

and negative examples are

$$\langle George, Elizabeth\rangle,\ \langle Harry, Zara\rangle,\ \langle Charles, Philip\rangle,\ \ldots$$

Notice that each example is a *pair* of objects, because *Grandfather* is a binary predicate. In all, there are 12 positive examples in the family tree and 388 negative examples (all the other pairs of people).

FOIL constructs a set of clauses, each with $Grandfather(x, y)$ as the head. The clauses must classify the 12 positive examples as instances of the $Grandfather(x, y)$ relationship, while ruling out the 388 negative examples. The clauses are Horn clauses, extended with negated literals using negation as failure, as in Prolog. The initial clause has an empty body:

$$\Rightarrow\ Grandfather(x, y)\ .$$

This clause classifies every example as positive, so it needs to be specialized. We do this by adding literals one at a time to the left-hand side. Here are three potential additions:

$$Father(x, y)\ \Rightarrow\ Grandfather(x, y)\ .$$
$$Parent(x, z)\ \Rightarrow\ Grandfather(x, y)\ .$$
$$Father(x, z)\ \Rightarrow\ Grandfather(x, y)\ .$$

(Notice that we are assuming that a clause defining *Parent* is already part of the background knowledge.) The first of these three clauses incorrectly classifies all of the 12 positive examples as negative and can thus be ignored. The second and third agree with all of the positive examples, but the second is incorrect on a larger fraction of the negative examples—twice as many, because it allows mothers as well as fathers. Hence, we prefer the third clause.

Now we need to specialize this clause further, to rule out the cases in which x is the father of some z, but z is not a parent of y. Adding the single literal $Parent(z, y)$ gives

$$Father(x, z) \wedge Parent(z, y)\ \Rightarrow\ Grandfather(x, y)\ ,$$

which correctly classifies all the examples. FOIL will find and choose this literal, thereby solving the learning task. In general, FOIL will have to search through many unsuccessful clauses before finding a correct solution.

This example is a very simple illustration of how FOIL operates. A sketch of the complete algorithm is shown in Figure 19.12. Essentially, the algorithm repeatedly constructs a clause, literal by literal, until it agrees with some subset of the positive examples and none of

```
function FOIL(examples, target) returns a set of Horn clauses
  inputs: examples, set of examples
          target, a literal for the goal predicate
  local variables: clauses, set of clauses, initially empty

  while examples contains positive examples do
     clause ← NEW-CLAUSE(examples, target)
     remove examples covered by clause from examples
     add clause to clauses
  return clauses
```

```
function NEW-CLAUSE(examples, target) returns a Horn clause
  local variables: clause, a clause with target as head and an empty body
                   l, a literal to be added to the clause
                   extended_examples, a set of examples with values for new variables

  extended_examples ← examples
  while extended_examples contains negative examples do
     l ← CHOOSE-LITERAL(NEW-LITERALS(clause), extended_examples)
     append l to the body of clause
     extended_examples ← set of examples created by applying EXTEND-EXAMPLE
        to each example in extended_examples
  return clause
```

```
function EXTEND-EXAMPLE(example, literal) returns
  if example satisfies literal
     then return the set of examples created by extending example with
       each possible constant value for each new variable in literal
  else return the empty set
```

Figure 19.12 Sketch of the FOIL algorithm for learning sets of first-order Horn clauses from examples. NEW-LITERAL and CHOOSE-LITERAL are explained in the text.

the negative examples. Then the positive examples covered by the clause are removed from the training set, and the process continues until no positive examples remain. The two main subroutines to be explained are NEW-LITERALS, which constructs all possible new literals to add to the clause, and CHOOSE-LITERAL, which selects a literal to add.

NEW-LITERALS takes a clause and constructs all possible "useful" literals that could be added to the clause. Let us use as an example the clause

$$Father(x, z) \Rightarrow Grandfather(x, y) .$$

There are three kinds of literals that can be added:

1. *Literals using predicates*: the literal can be negated or unnegated, any existing predicate (including the goal predicate) can be used, and the arguments must all be variables. Any variable can be used for any argument of the predicate, with one restriction: each literal

must include *at least one* variable from an earlier literal or from the head of the clause. Literals such as $Mother(z, u)$, $Married(z, z)$, $\neg Male(y)$, and $Grandfather(v, x)$ are allowed, whereas $Married(u, v)$ is not. Notice that the use of the predicate from the head of the clause allows FOIL to learn *recursive* definitions.

2. *Equality and inequality literals*: these relate variables already appearing in the clause. For example, we might add $z \neq x$. These literals can also include user-specified constants. For learning arithmetic we might use 0 and 1, and for learning list functions we might use the empty list [].
3. *Arithmetic comparisons*: when dealing with functions of continuous variables, literals such as $x > y$ and $y \leq z$ can be added. As in decision-tree learning, a constant threshold value can be chosen to maximize the discriminatory power of the test.

The resulting branching factor in this search space is very large (see Exercise 19.6), but FOIL can also use type information to reduce it. For example, if the domain included numbers as well as people, type restrictions would prevent NEW-LITERALS from generating literals such as $Parent(x, n)$, where x is a person and n is a number.

CHOOSE-LITERAL uses a heuristic somewhat similar to information gain (see page 660) to decide which literal to add. The exact details are not so important here, and a number of different variations have been tried. One interesting additional feature of FOIL is the use of Ockham's razor to eliminate some hypotheses. If a clause becomes longer (according to some metric) than the total length of the positive examples that the clause explains, that clause is not considered as a potential hypothesis. This technique provides a way to avoid overcomplex clauses that fit noise in the data. For an explanation of the connection between noise and clause length, see page 715.

FOIL and its relatives have been used to learn a wide variety of definitions. One of the most impressive demonstrations (Quinlan and Cameron-Jones, 1993) involved solving a long sequence of exercises on list-processing functions from Bratko's (1986) Prolog textbook. In each case, the program was able to learn a correct definition of the function from a small set of examples, using the previously learned functions as background knowledge.

Inductive learning with inverse deduction

INVERSE RESOLUTION

The second major approach to ILP involves inverting the normal deductive proof process. **Inverse resolution** is based on the observation that if the example *Classifications* follow from $Background \wedge Hypothesis \wedge Descriptions$, then one must be able to prove this fact by resolution (because resolution is complete). If we can "run the proof backward," then we can find a *Hypothesis* such that the proof goes through. The key, then, is to find a way to invert the resolution process.

We will show a backward proof process for inverse resolution that consists of individual backward steps. Recall that an ordinary resolution step takes two clauses C_1 and C_2 and resolves them to produce the **resolvent** C. An inverse resolution step takes a resolvent C and produces two clauses C_1 and C_2, such that C is the result of resolving C_1 and C_2. Alternatively, it may take a resolvent C and clause C_1 and produce a clause C_2 such that C is the result of resolving C_1 and C_2.

The early steps in an inverse resolution process are shown in Figure 19.13, where we focus on the positive example $Grandparent(George, Anne)$. The process begins at the end of the proof (shown at the bottom of the figure). We take the resolvent C to be empty clause (i.e. a contradiction) and C_2 to be $\neg Grandparent(George, Anne)$, which is the negation of the goal example. The first inverse step takes C and C_2 and generates the clause $Grandparent(George, Anne)$ for C_1. The next step takes this clause as C and the clause $Parent(Elizabeth, Anne)$ as C_2, and generates the clause

$$\neg Parent(Elizabeth, y) \vee Grandparent(George, y)$$

as C_1. The final step treats this clause as the resolvent. With $Parent(George, Elizabeth)$ as C_2, one possible clause C_1 is the hypothesis

$$Parent(x, z) \wedge Parent(z, y) \Rightarrow Parent(x, y) .$$

Now we have a resolution proof that the hypothesis, descriptions, and background knowledge entail the classification $Grandparent(George, Anne)$.

Clearly, inverse resolution involves a search. Each inverse resolution step is nondeterministic, because for any C, there can be many or even an infinite number of clauses C_1 and C_2 that resolve to C. For example, instead of choosing $\neg Parent(Elizabeth, y) \vee Grandparent(George, y)$ for C_1 in the last step of Figure 19.13, the inverse resolution step might have chosen any of the following sentences:

$$\neg Parent(Elizabeth, Anne) \vee Grandparent(George, Anne) .$$
$$\neg Parent(z, Anne) \vee Grandparent(George, Anne) .$$
$$\neg Parent(z, y) \vee Grandparent(George, y) .$$
$$\vdots$$

(See Exercises 19.4 and 19.5.) Furthermore, the clauses that participate in each step can be chosen from the *Background* knowledge, from the example *Descriptions*, from the negated

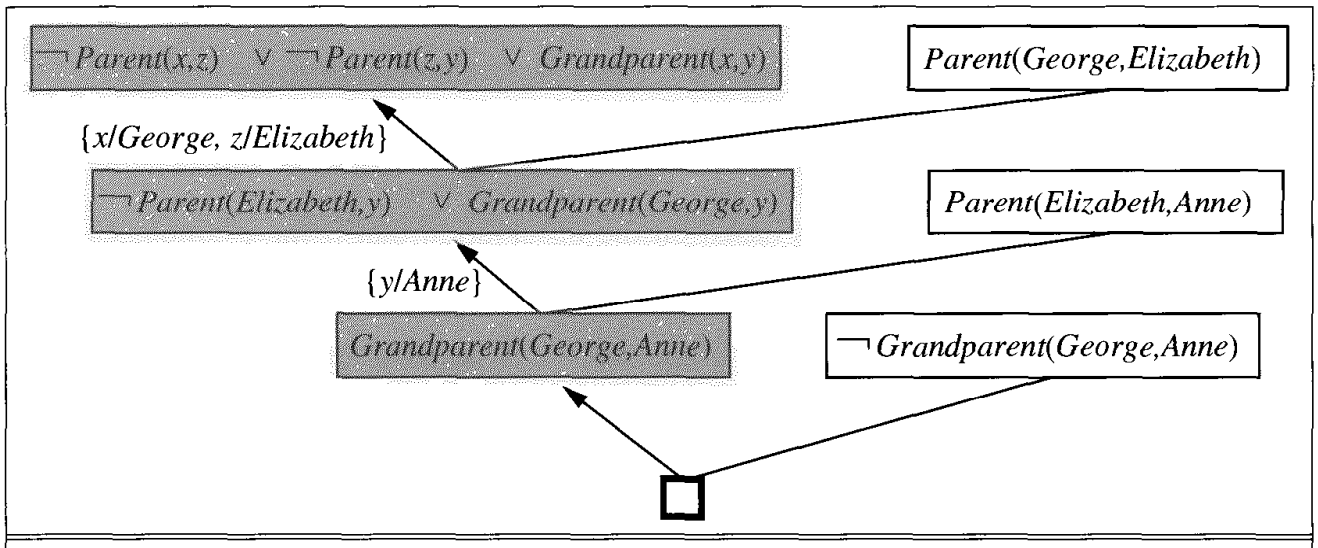


Figure 19.13 Early steps in an inverse resolution process. The shaded clauses are generated by inverse resolution steps from the clause to the right and the clause below. The unshaded clauses are from the *Descriptions* and *Classifications*.

Classifications, or from hypothesized clauses that have already been generated in the inverse resolution tree. The large number of possibilities means a large branching factor (and therefore an inefficient search) without additional controls. A number of approaches to taming the search have been tried in implemented ILP systems:

1. Redundant choices can be eliminated—for example, by generating only the most specific hypotheses possible and by requiring that all the hypothesized clauses be consistent with each other, and with the observations. This last criterion would rule out the clause $\neg Parent(z, y) \vee Grandparent(George, y)$, listed before.
2. The proof strategy can be restricted. For example, we saw in Chapter 9 that **linear resolution** is a complete, restricted strategy that allows proof trees to have only a linear branching structure (as in Figure 19.13).
3. The representation language can be restricted, for example by eliminating function symbols or by allowing only Horn clauses. For instance, PROGOL operates with Horn clauses using **inverse entailment**. The idea is to change the entailment constraint

   INVERSE ENTAILMENT

   $$Background \wedge Hypothesis \wedge Descriptions \models Classifications$$

   to the logically equivalent form

   $$Background \wedge Descriptions \wedge \neg Classifications \models \neg Hypothesis.$$

   From this, one can use a process similar to the normal Prolog Horn-clause deduction, with negation-as-failure to derive *Hypothesis*. Because it is restricted to Horn clauses, this is an incomplete method, but it can be more efficient than full resolution. It is also possible to apply complete inference with inverse entailment Inoue (2001).
4. Inference can be done with model checking rather than theorem proving. The PROGOL system (Muggleton, 1995) uses a form of model checking to limit the search. That is, like answer set programming, it generates possible values for logical variables, and checks for consistency.
5. Inference can be done with ground propositional clauses rather than in first-order logic. The LINUS system (Lavrač and Džeroski, 1994) works by translating first-order theories into propositional logic, solving them with a propositional learning system, and then translating back. Working with propositional formulas can be more efficient on some problems, as we saw with SATPLAN in Chapter 11.

Making discoveries with inductive logic programming

An inverse resolution procedure that inverts a complete resolution strategy is, in principle, a complete algorithm for learning first-order theories. That is, if some unknown *Hypothesis* generates a set of examples, then an inverse resolution procedure can generate *Hypothesis* from the examples. This observation suggests an interesting possibility: Suppose that the available examples include a variety of trajectories of falling bodies. Would an inverse resolution program be theoretically capable of inferring the law of gravity? The answer is clearly yes, because the law of gravity allows one to explain the examples, given suitable background mathematics. Similarly, one can imagine that electromagnetism, quantum mechanics, and the

theory of relativity are also within the scope of ILP programs. Of course, they are also within the scope of a monkey with a typewriter; we still need better heuristics and new ways to structure the search space.

One thing that inverse resolution systems *will* do for you is invent new predicates. This ability is often seen as somewhat magical, because computers are often thought of as "merely working with what they are given." In fact, new predicates fall directly out of the inverse resolution step. The simplest case arises in hypothesizing two new clauses C_1 and C_2, given a clause C. The resolution of C_1 and C_2 eliminates a literal that the two clauses share; hence, it is quite possible that the eliminated literal contained a predicate that does not appear in C. Thus, when working backward, one possibility is to generate a new predicate from which to reconstruct the missing literal.

Figure 19.14 shows an example in which the new predicate P is generated in the process of learning a definition for $Ancestor$. Once generated, P can be used in later inverse resolution steps. For example, a later step might hypothesize that $Mother(x, y) \Rightarrow P(x, y)$. Thus, the new predicate P has its meaning constrained by the generation of hypotheses that involve it. Another example might lead to the constraint $Father(x, y) \Rightarrow P(x, y)$. In other words, the predicate P is what we usually think of as the $Parent$ relationship. As we mentioned earlier, the invention of new predicates can significantly reduce the size of the definition of the goal predicate. Hence, by including the ability to invent new predicates, inverse resolution systems can often solve learning problems that are infeasible with other techniques.

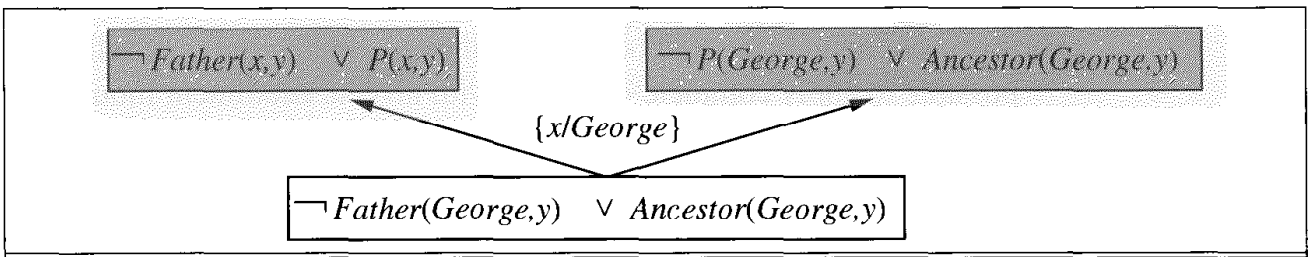


Figure 19.14 An inverse resolution step that generates a new predicate P.

Some of the deepest revolutions in science come from the invention of new predicates and functions—for example, Galileo's invention of acceleration or Joule's invention of thermal energy. Once these terms are available, the discovery of new laws becomes (relatively) easy. The difficult part lies in realizing that some new entity, with a specific relationship to existing entities, will allow an entire body of observations to be explained with a much simpler and more elegant theory than previously existed.

As yet, ILP systems have not made discoveries on the level of Galileo or Joule, but their discoveries have been deemed publishable in the scientific literature. For example, in the *Journal of Molecular Biology*, Turcotte *et al.* (2001) describe the automated discovery of rules for protein folding by the ILP program PROGOL. Many of the rules discovered by PROGOL could have been derived from known principles, but most had not been previously published as part of a standard biological database. (See Figure 19.10 for an example.). In related work, Srinivasan *et al.* (1994) dealt with the problem of discovering molecular-structure-based rules for the mutagenicity of nitroaromatic compounds. These compounds are found in

automobile exhaust fumes. For 80% of the compounds in a standard database, it is possible to identify four important features, and linear regression on these features outperforms ILP. For the remaining 20%, the features alone are not predictive, and ILP identifies relationships which allow it to outperform linear regression, neural nets, and decision trees. King *et al.* (1992) showed how to predict the therapeutic efficacy of various drugs from their molecular structures. For all these examples it appears that both the ability to represent relations and to use background knowledge contribute to ILP's high performance. The fact that the rules found by ILP can be interpreted by humans contributes to the acceptance of these techniques in biology journals rather than just computer science journals.

ILP has made contributions to other sciences besides biology. One of the most important is natural language processing, where ILP has been used to extract complex relational information from text. These results are summarized in Chapter 23.

19.6 SUMMARY

This chapter has investigated various ways in which prior knowledge can help an agent to learn from new experiences. Because much prior knowledge is expressed in terms of relational models rather than attribute-based models, we have also covered systems that allow learning of relational models. The important points are:

- The use of prior knowledge in learning leads to a picture of **cumulative learning**, in which learning agents improve their learning ability as they acquire more knowledge.
- Prior knowledge helps learning by eliminating otherwise consistent hypotheses and by "filling in" the explanation of examples, thereby allowing for shorter hypotheses. These contributions often result in faster learning from fewer examples.
- Understanding the different logical roles played by prior knowledge, as expressed by **entailment constraints**, helps to define a variety of learning techniques.
- **Explanation-based learning** (EBL) extracts general rules from single examples by *explaining* the examples and generalizing the explanation. It provides a deductive method turning first-principles knowledge into useful, efficient, special-purpose expertise.
- **Relevance-based learning** (RBL) uses prior knowledge in the form of determinations to identify the relevant attributes, thereby generating a reduced hypothesis space and speeding up learning. RBL also allows deductive generalizations from single examples.
- **Knowledge-based inductive learning** (KBIL) finds inductive hypotheses that explain sets of observations with the help of background knowledge.
- **Inductive logic programming** (ILP) techniques perform KBIL on knowledge that is expressed in first-order logic. ILP methods can learn relational knowledge that is not expressible in attribute-based systems.
- ILP can be done with a top-down approach of refining a very general rule or through a bottom-up approach of inverting the deductive process.
- ILP methods naturally generate new predicates with which concise new theories can be expressed and show promise as general-purpose scientific theory formation systems.

BIBLIOGRAPHICAL AND HISTORICAL NOTES

Although the use of prior knowledge in learning would seem to be a natural topic for philosophers of science, little formal work was done until quite recently. *Fact, Fiction, and Forecast*, by the philosopher Nelson Goodman (1954), refuted the earlier supposition that induction was simply a matter of seeing enough examples of some universally quantified proposition and then adopting it as a hypothesis. Consider, for example, the hypothesis "All emeralds are grue," where **grue** means "green if observed before time t, but blue if observed thereafter." At any time up to t, we might have observed millions of instances confirming the rule that emeralds are grue, and no disconfirming instances, and yet we are unwilling to adopt the rule. This can be explained only by appeal to the role of relevant prior knowledge in the induction process. Goodman proposes a variety of different kinds of prior knowledge that might be useful, including a version of determinations called **overhypotheses**. Unfortunately, Goodman's ideas were never pursued in machine learning.

GRUE

EBL had its roots in the techniques used by the STRIPS planner (Fikes *et al.*, 1972). When a plan was constructed, a generalized version of it was saved in a plan library and used in later planning as a **macro-operator**. Similar ideas appeared in Anderson's ACT* architecture, under the heading of **knowledge compilation** (Anderson, 1983), and in the SOAR architecture, as **chunking** (Laird *et al.*, 1986). **Schema acquisition** (DeJong, 1981), **analytical generalization** (Mitchell, 1982), and **constraint-based generalization** (Minton, 1984) were immediate precursors of the rapid growth of interest in EBL stimulated by the papers of Mitchell *et al.* (1986) and DeJong and Mooney (1986). Hirsh (1987) introduced the EBL algorithm described in the text, showing how it could be incorporated directly into a logic programming system. Van Harmelen and Bundy (1988) explain EBL as a variant of the **partial evaluation** method used in program analysis systems (Jones *et al.*, 1993).

More recently, rigorous analysis has led to a better understanding of the potential costs and benefits of EBL in terms of problem-solving speed. Minton (1988) showed that, without extensive extra work, EBL could easily slow down a program significantly. Tambe *et al.* (1990) found a similar problem with chunking and proposed a reduction in the expressive power of the rule language in order to minimize the cost of matching rules against working memory. This work has strong parallels with recent results on the complexity of inference in restricted versions of first-order logic. (See Chapter 9.) Formal probabilistic analysis of the expected payoff of EBL can be found in Greiner (1989) and Subramanian and Feldman (1990). An excellent survey appears in Dietterich (1990).

Instead of using examples as foci for generalization, one can use them directly to solve new problems, in a process known as **analogical reasoning**. This form of reasoning ranges from a form of plausible reasoning based on degree of similarity (Gentner, 1983), through a form of deductive inference based on determinations but requiring the participation of the example (Davies and Russell, 1987), to a form of "lazy" EBL that tailors the direction of generalization of the old example to fit the needs of the new problem. This latter form of analogical reasoning is found most commonly in **case-based reasoning** (Kolodner, 1993) and **derivational analogy** (Veloso and Carbonell, 1993).

ANALOGICAL REASONING

Relevance information in the form of functional dependencies was first developed in the database community, where it is used to structure large sets of attributes into manageable subsets. Functional dependencies were used for analogical reasoning by Carbonell and Collins (1973) and were given a more logical flavor by Bobrow and Raphael (1974). Dependencies were independently rediscovered and given a full logical analysis by Davies and Russell (Davies, 1985; Davies and Russell, 1987). They were used for declarative bias by Russell and Grosof (1987). The equivalence of determinations to a restricted-vocabulary hypothesis space was proved in Russell (1988). Learning algorithms for determinations and the improved performance obtained by RBDTL were first shown in the FOCUS algorithm in Almuallim and Dietterich (1991). Tadepalli (1993) describes an ingenious algorithm for learning with determinations that shows large improvements in learning speed.

The idea that inductive learning can be performed by inverse deduction can be traced to W. S. Jevons (1874), who wrote, "The study both of Formal Logic and of the Theory of Probabilities has led me to adopt the opinion that there is no such thing as a distinct method of induction as contrasted with deduction, but that induction is simply an inverse employment of deduction." Computational investigations began with the remarkable Ph.D. thesis by Gordon Plotkin (1971) at Edinburgh. Although Plotkin developed many of the theorems and methods that are in current use in ILP, he was discouraged by some undecidability results for certain subproblems in induction. MIS (Shapiro, 1981) reintroduced the problem of learning logic programs, but was seen mainly as a contribution to the theory of automated debugging. Work on rule induction, such as the ID3 (Quinlan, 1986) and CN2 (Clark and Niblett, 1989) systems, led to FOIL (Quinlan, 1990), which for the first time allowed practical induction of relational rules. The field of relational learning was reinvigorated by Muggleton and Buntine (1988), whose CIGOL program incorporated a slightly incomplete version of inverse resolution and was capable of generating new predicates.[5] Wirth and O'Rorke (1991) also cover predicate invention. The next major system was GOLEM (Muggleton and Feng, 1990), which uses a covering algorithm based on Plotkin's concept of relative least general generalization. Where FOIL was top-down, CIGOL and GOLEM worked bottom-up. ITOU (Rouveirol and Puget, 1989) and CLINT (De Raedt, 1992) were other systems of that era. More recently, PROGOL (Muggleton, 1995) has taken a hybrid (top-down and bottom-up) approach to inverse entailment and has been applied to a number of practical problems, particularly in biology and natural language processing. Muggleton (2000) describes an extension of PROGOL to handle uncertainty in the form of stochastic logic programs.

A formal analysis of ILP methods appears in Muggleton (1991), a large collection of papers in Muggleton (1992), and a collection of techniques and applications in the book by Lavrač and Džeroski (1994). Page and Srinivasan (2002) give a more recent overview of the field's history and challenges for the future. Early complexity results by Haussler (1989) suggested that learning first-order sentences was hopelessly complex. However, with better understanding of the importance of various kinds of syntactic restrictions on clauses, positive results have been obtained even for clauses with recursion (Dzeroski *et al.*, 1992). Learnability results for ILP are surveyed by Kietz and Dzeroski (1994) and Cohen and Page (1995).

[5] The inverse resolution method also appears in (Russell, 1986), with a simple algorithm given in a footnote.

DISCOVERY SYSTEMS

Although ILP now seems to be the dominant approach to constructive induction, it has not been the only approach taken. So-called **discovery systems** aim to model the process of scientific discovery of new concepts, usually by a direct search in the space of concept definitions. Doug Lenat's Automated Mathematician, or AM, (Davis and Lenat, 1982) used discovery heuristics expressed as expert system rules to guide its search for concepts and conjectures in elementary number theory. Unlike most systems designed for mathematical reasoning, AM lacked a concept of proof and could only make conjectures. It rediscovered Goldbach's conjecture and the Unique Prime Factorization theorem. AM's architecture was generalized in the EURISKO system (Lenat, 1983) by adding a mechanism capable of rewriting the system's own discovery heuristics. EURISKO was applied in a number of areas other than mathematical discovery, although with less success than AM. The methodology of AM and EURISKO has been controversial (Ritchie and Hanna, 1984; Lenat and Brown, 1984).

Another class of discovery systems aims to operate with real scientific data to find new laws. The systems DALTON, GLAUBER, and STAHL (Langley *et al.*, 1987) are rule-based systems that look for quantitative relationships in experimental data from physical systems; in each case, the system has been able to recapitulate a well-known discovery from the history of science. Discovery systems based on probabilistic techniques—especially clustering algorithms that discover new categories—are discussed in Chapter 20.

EXERCISES

19.1 Show, by translating into conjunctive normal form and applying resolution, that the conclusion drawn on page 694 concerning Brazilians is sound.

19.2 For each of the following determinations, write down the logical representation and explain why the determination is true (if it is):

a. Zip code determines the state (U.S.).

b. Design and denomination determine the mass of a coin.

c. For a given program, input determines output.

d. Climate, food intake, exercise, and metabolism determine weight gain and loss.

e. Baldness is determined by the baldness (or lack thereof) of one's maternal grandfather.

19.3 Would a probabilistic version of determinations be useful? Suggest a definition.

19.4 Fill in the missing values for the clauses C_1 or C_2 (or both) in the following sets of clauses, given that C is the resolvent of C_1 and C_2:

a. $C = True \Rightarrow P(A, B)$, $C_1 = P(x, y) \Rightarrow Q(x, y)$, $C_2 = ??$.

b. $C = True \Rightarrow P(A, B)$, $C_1 = ??$, $C_2 = ??$.

c. $C = P(x, y) \Rightarrow P(x, f(y))$, $C_1 = ??$, $C_2 = ??$.

If there is more than one possible solution, provide one example of each different kind.

19.5 Suppose one writes a logic program that carries out a resolution inference step. That is, let $Resolve(c_1, c_2, c)$ succeed if c is the result of resolving c_1 and c_2. Normally, *Resolve* would be used as part of a theorem prover by calling it with c_1 and c_2 instantiated to particular clauses, thereby generating the resolvent c. Now suppose instead that we call it with c instantiated and c_1 and c_2 uninstantiated. Will this succeed in generating the appropriate results of an inverse resolution step? Would you need any special modifications to the logic programming system for this to work?

19.6 Suppose that FOIL is considering adding a literal to a clause using a binary predicate P and that previous literals (including the head of the clause) contain five different variables.

a. How many functionally different literals can be generated? Two literals are functionally identical if they differ only in the names of the *new* variables that they contain.

b. Can you find a general formula for the number of different literals with a predicate of arity r when there are n variables previously used?

c. Why does FOIL not allow literals that contain no previously used variables?

19.7 Using the data from the family tree in Figure 19.11, or a subset thereof, apply the FOIL algorithm to learn a definition for the *Ancestor* predicate.

20 STATISTICAL LEARNING METHODS

In which we view learning as a form of uncertain reasoning from observations.

Part V pointed out the prevalence of uncertainty in real environments. Agents can handle uncertainty by using the methods of probability and decision theory, but first they must learn their probabilistic theories of the world from experience. This chapter explains how they can do that. We will see how to formulate the learning task itself as a process of probabilistic inference (Section 20.1). We will see that a Bayesian view of learning is extremely powerful, providing general solutions to the problems of noise, overfitting, and optimal prediction. It also takes into account the fact that a less-than-omniscient agent can never be certain about which theory of the world is correct, yet must still make decisions by using some theory of the world.

We describe methods for learning probability models—primarily Bayesian networks—in Sections 20.2 and 20.3. Section 20.4 looks at learning methods that store and recall specific instances. Section 20.5 covers **neural network** learning and Section 20.6 introduces **kernel machines**. Some of the material in this chapter is fairly mathematical (requiring a basic understanding of multivariate calculus), although the general lessons can be understood without plunging into the details. It may benefit the reader at this point to review the material in Chapters 13 and 14 and to peek at the mathematical background in Appendix A.

20.1 STATISTICAL LEARNING

The key concepts in this chapter, just as in Chapter 18, are **data** and **hypotheses**. Here, the data are **evidence**—that is, instantiations of some or all of the random variables describing the domain. The hypotheses are probabilistic theories of how the domain works, including logical theories as a special case.

Let us consider a *very* simple example. Our favorite Surprise candy comes in two flavors: cherry (yum) and lime (ugh). The candy manufacturer has a peculiar sense of humor and wraps each piece of candy in the same opaque wrapper, regardless of flavor. The candy is sold in very large bags, of which there are known to be five kinds—again, indistinguishable from the outside:

h_1: 100% cherry
h_2: 75% cherry + 25% lime
h_3: 50% cherry + 50% lime
h_4: 25% cherry + 75% lime
h_5: 100% lime

Given a new bag of candy, the random variable H (for *hypothesis*) denotes the type of the bag, with possible values h_1 through h_5. H is not directly observable, of course. As the pieces of candy are opened and inspected, data are revealed—$D_1, D_2, \ldots, D_N$, where each D_i is a random variable with possible values *cherry* and *lime*. The basic task faced by the agent is to predict the flavor of the next piece of candy.[1] Despite its apparent triviality, this scenario serves to introduce many of the major issues. The agent really does need to infer a theory of its world, albeit a very simple one.

BAYESIAN LEARNING

Bayesian learning simply calculates the probability of each hypothesis, given the data, and makes predictions on that basis. That is, the predictions are made by using *all* the hypotheses, weighted by their probabilities, rather than by using just a single "best" hypothesis. In this way, learning is reduced to probabilistic inference. Let $\mathbf{D}$ represent all the data, with observed value $\mathbf{d}$; then the probability of each hypothesis is obtained by Bayes' rule:

$$P(h_i|\mathbf{d}) = \alpha P(\mathbf{d}|h_i)P(h_i) \ . \tag{20.1}$$

Now, suppose we want to make a prediction about an unknown quantity X. Then we have

$$\mathbf{P}(X|\mathbf{d}) = \sum_i \mathbf{P}(X|\mathbf{d}, h_i)\mathbf{P}(h_i|\mathbf{d}) = \sum_i \mathbf{P}(X|h_i)P(h_i|\mathbf{d}) \ , \tag{20.2}$$

where we have assumed that each hypothesis determines a probability distribution over X. This equation shows that predictions are weighted averages over the predictions of the individual hypotheses. The hypotheses themselves are essentially "intermediaries" between the raw data and the predictions. The key quantities in the Bayesian approach are the **hypothesis prior**, $P(h_i)$, and the **likelihood** of the data under each hypothesis, $P(\mathbf{d}|h_i)$.

HYPOTHESIS PRIOR

LIKELIHOOD

For our candy example, we will assume for the time being that the prior distribution over $h_1, \ldots, h_5$ is given by $\langle 0.1, 0.2, 0.4, 0.2, 0.1 \rangle$, as advertised by the manufacturer. The likelihood of the data is calculated under the assumption that the observations are **i.i.d.**—that is, independently and identically distributed—so that

I.I.D.

$$P(\mathbf{d}|h_i) = \prod_j P(d_j|h_i) \ . \tag{20.3}$$

For example, suppose the bag is really an all-lime bag (h_5) and the first 10 candies are all lime; then $P(\mathbf{d}|h_3)$ is 0.5^{10}, because half the candies in an h_3 bag are lime.[2] Figure 20.1(a) shows how the posterior probabilities of the five hypotheses change as the sequence of 10 lime candies is observed. Notice that the probabilities start out at their prior values, so h_3 is initially the most likely choice and remains so after 1 lime candy is unwrapped. After 2

[1] Statistically sophisticated readers will recognize this scenario as a variant of the **urn-and-ball** setup. We find urns and balls less compelling than candy; furthermore, candy lends itself to other tasks, such as deciding whether to trade the bag with a friend—see Exercise 20.3.

[2] We stated earlier that the bags of candy are very large; otherwise, the i.i.d. assumption fails to hold. Technically, it is more correct (but less hygienic) to rewrap each candy after inspection and return it to the bag.

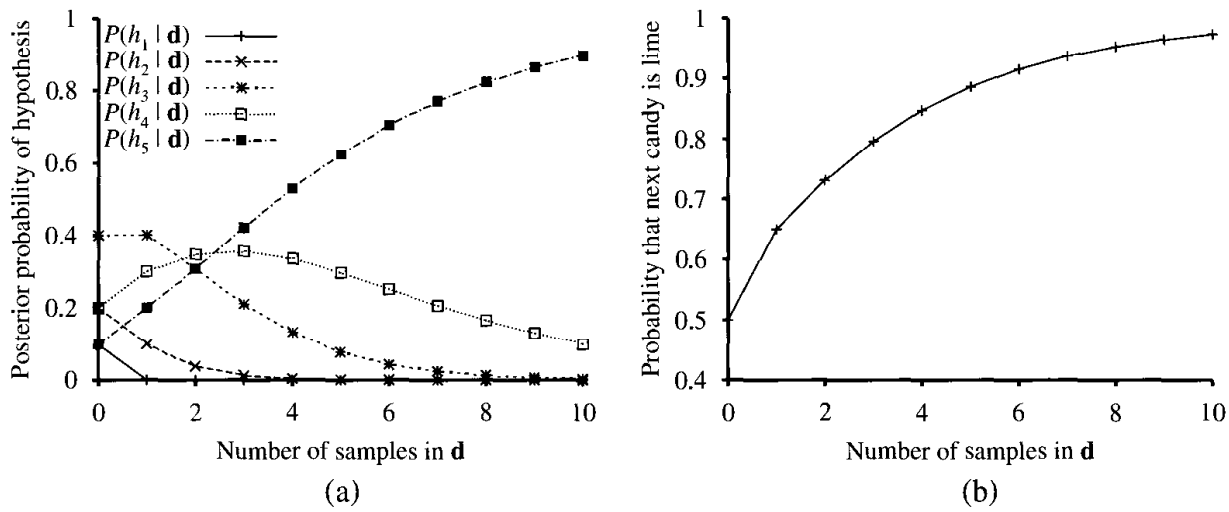


Figure 20.1 (a) Posterior probabilities $P(h_i|d_1, \ldots, d_N)$ from Equation (20.1). The number of observations N ranges from 1 to 10, and each observation is of a lime candy. (b) Bayesian prediction $P(d_{N+1} = lime|d_1, \ldots, d_N)$ from Equation (20.2).

lime candies are unwrapped, h_4 is most likely; after 3 or more, h_5 (the dreaded all-lime bag) is the most likely. After 10 in a row, we are fairly certain of our fate. Figure 20.1(b) shows the predicted probability that the next candy is lime, based on Equation (20.2). As we would expect, it increases monotonically toward 1.

The example shows that *the true hypothesis eventually dominates the Bayesian prediction.* This is characteristic of Bayesian learning. For any fixed prior that does not rule out the true hypothesis, the posterior probability of any false hypothesis will eventually vanish, simply because the probability of generating "uncharacteristic" data indefinitely is vanishingly small. (This point is analogous to one made in the discussion of PAC learning in Chapter 18.) More importantly, the Bayesian prediction is *optimal*, whether the data set be small or large. Given the hypothesis prior, any other prediction will be correct less often.

The optimality of Bayesian learning comes at a price, of course. For real learning problems, the hypothesis space is usually very large or infinite, as we saw in Chapter 18. In some cases, the summation in Equation (20.2) (or integration, in the continuous case) can be carried out tractably, but in most cases we must resort to approximate or simplified methods.

A very common approximation—one that is usually adopted in science—is to make predictions based on a single *most probable* hypothesis—that is, an h_i that maximizes $P(h_i|\mathbf{d})$. This is often called a **maximum a posteriori** or MAP (pronounced "em-ay-pee") hypothesis. Predictions made according to an MAP hypothesis h_{MAP} are approximately Bayesian to the extent that $\mathbf{P}(X|\mathbf{d}) \approx \mathbf{P}(X|h_{\text{MAP}})$. In our candy example, $h_{\text{MAP}} = h_5$ after three lime candies in a row, so the MAP learner then predicts that the fourth candy is lime with probability 1.0—a much more dangerous prediction than the Bayesian prediction of 0.8 shown in Figure 20.1. As more data arrive, the MAP and Bayesian predictions become closer, because the competitors to the MAP hypothesis become less and less probable. Although our example doesn't show it, finding MAP hypotheses is often much easier than Bayesian learn-

MAXIMUM A POSTERIORI

ing, because it requires solving an optimization problem instead of a large summation (or integration) problem. We will see examples of this later in the chapter.

In both Bayesian learning and MAP learning, the hypothesis prior $P(h_i)$ plays an important role. We saw in Chapter 18 that **overfitting** can occur when the hypothesis space is too expressive, so that it contains many hypotheses that fit the data set well. Rather than placing an arbitrary limit on the hypotheses to be considered, Bayesian and MAP learning methods use the prior to *penalize complexity*. Typically, more complex hypotheses have a lower prior probability—in part because there are usually many more complex hypotheses than simple hypotheses. On the other hand, more complex hypotheses have a greater capacity to fit the data. (In the extreme case, a lookup table can reproduce the data exactly with probability 1.) Hence, the hypothesis prior embodies a trade-off between the complexity of a hypothesis and its degree of fit to the data.

We can see the effect of this trade-off most clearly in the logical case, where H contains only *deterministic* hypotheses. In that case, $P(\mathbf{d}|h_i)$ is 1 if h_i is consistent and 0 otherwise. Looking at Equation (20.1), we see that h_{MAP} will then be the *simplest logical theory that is consistent with the data.* Therefore, maximum *a posteriori* learning provides a natural embodiment of Ockham's razor.

Another insight into the trade-off between complexity and degree of fit is obtained by taking the logarithm of Equation (20.1). Choosing h_{MAP} to maximize $P(\mathbf{d}|h_i)P(h_i)$ is equivalent to minimizing

$$-\log_2 P(\mathbf{d}|h_i) - \log_2 P(h_i) \ .$$

Using the connection between information encoding and probability that we introduced in Chapter 18, we see that the $-\log_2 P(h_i)$ term equals the number of bits required to specify the hypothesis h_i. Furthermore, $-\log_2 P(\mathbf{d}|h_i)$ is the additional number of bits required to specify the data, given the hypothesis. (To see this, consider that no bits are required if the hypothesis predicts the data exactly—as with h_5 and the string of lime candies—and $\log_2 1 = 0$.) Hence, MAP learning is choosing the hypothesis that provides maximum *compression* of the data. The same task is addressed more directly by the **minimum description length**, or MDL, learning method, which attempts to minimize the size of hypothesis and data encodings rather than work with probabilities.

MINIMUM DESCRIPTION LENGTH

A final simplification is provided by assuming a **uniform** prior over the space of hypotheses. In that case, MAP learning reduces to choosing an h_i that maximizes $P(\mathbf{d}|H_i)$. This is called a **maximum-likelihood** (ML) hypothesis, h_{ML}. Maximum-likelihood learning is very common in statistics, a discipline in which many researchers distrust the subjective nature of hypothesis priors. It is a reasonable approach when there is no reason to prefer one hypothesis over another *a priori*—for example, when all hypotheses are equally complex. It provides a good approximation to Bayesian and MAP learning when the data set is large, because the data swamps the prior distribution over hypotheses, but it has problems (as we shall see) with small data sets.

MAXIMUM-LIKELIHOOD

20.2 Learning with Complete Data

PARAMETER LEARNING

COMPLETE DATA

Our development of statistical learning methods begins with the simplest task: **parameter learning** with **complete data**. A parameter learning task involves finding the numerical parameters for a probability model whose structure is fixed. For example, we might be interested in learning the conditional probabilities in a Bayesian network with a given structure. Data are complete when each data point contains values for every variable in the probability model being learned. Complete data greatly simplify the problem of learning the parameters of a complex model. We will also look briefly at the problem of learning structure.

Maximum-likelihood parameter learning: Discrete models

Suppose we buy a bag of lime and cherry candy from a new manufacturer whose lime–cherry proportions are completely unknown—that is, the fraction could be anywhere between 0 and 1. In that case, we have a continuum of hypotheses. The **parameter** in this case, which we call θ, is the proportion of cherry candies, and the hypothesis is h_θ. (The proportion of limes is just $1-\theta$.) If we assume that all proportions are equally likely *a priori*, then a maximum-likelihood approach is reasonable. If we model the situation with a Bayesian network, we need just one random variable, *Flavor* (the flavor of a randomly chosen candy from the bag). It has values *cherry* and *lime*, where the probability of *cherry* is θ (see Figure 20.2(a)). Now suppose we unwrap N candies, of which c are cherries and $\ell = N - c$ are limes. According to Equation (20.3), the likelihood of this particular data set is

$$P(\mathbf{d}|h_\theta) = \prod_{j=1}^{N} P(d_j|h_\theta) = \theta^c \cdot (1-\theta)^\ell \,.$$

LOG LIKELIHOOD

The maximum-likelihood hypothesis is given by the value of θ that maximizes this expression. The same value is obtained by maximizing the **log likelihood**,

$$L(\mathbf{d}|h_\theta) = \log P(\mathbf{d}|h_\theta) = \sum_{j=1}^{N} \log P(d_j|h_\theta) = c\log\theta + \ell\log(1-\theta) \,.$$

(By taking logarithms, we reduce the product to a sum over the data, which is usually easier to maximize.) To find the maximum-likelihood value of θ, we differentiate L with respect to θ and set the resulting expression to zero:

$$\frac{dL(\mathbf{d}|h_\theta)}{d\theta} = \frac{c}{\theta} - \frac{\ell}{1-\theta} = 0 \qquad \Rightarrow \qquad \theta = \frac{c}{c+\ell} = \frac{c}{N} \,.$$

In English, then, the maximum-likelihood hypothesis h_{ML} asserts that the actual proportion of cherries in the bag is equal to the observed proportion in the candies unwrapped so far!

It appears that we have done a lot of work to discover the obvious. In fact, though, we have laid out one standard method for maximum-likelihood parameter learning:

1. Write down an expression for the likelihood of the data as a function of the parameter(s).
2. Write down the derivative of the log likelihood with respect to each parameter.
3. Find the parameter values such that the derivatives are zero.

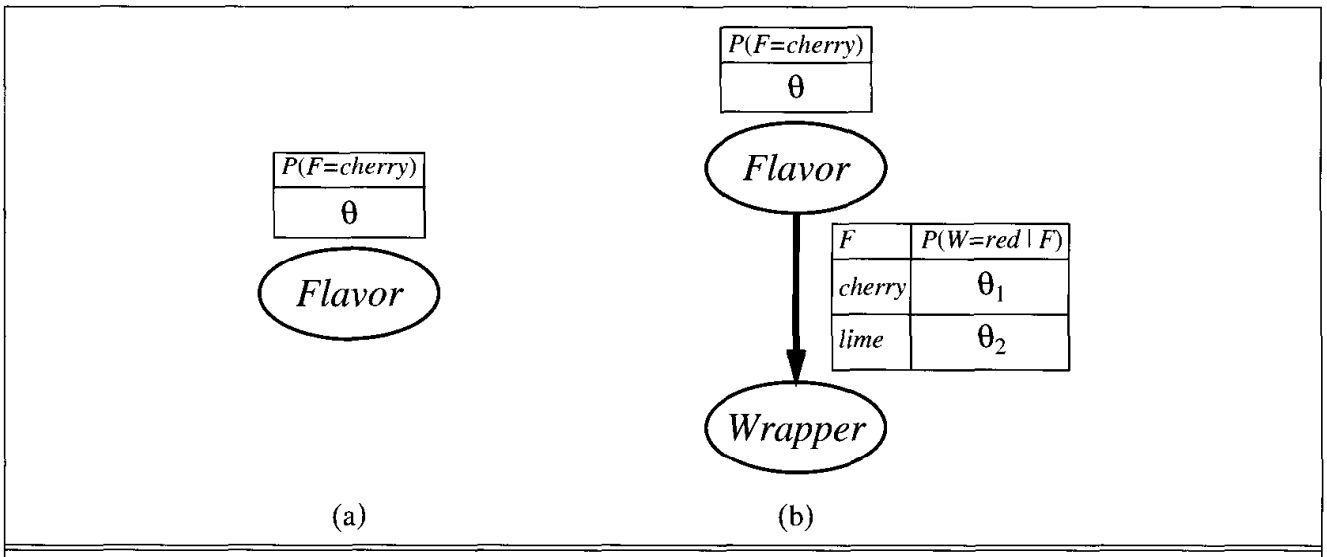


Figure 20.2 (a) Bayesian network model for the case of candies with an unknown proportion of cherries and limes. (b) Model for the case where the wrapper color depends (probabilistically) on the candy flavor.

The trickiest step is usually the last. In our example, it was trivial, but we will see that in many cases we need to resort to iterative solution algorithms or other numerical optimization techniques, as described in Chapter 4. The example also illustrates a significant problem with maximum-likelihood learning in general: *when the data set is small enough that some events have not yet been observed—for instance, no cherry candies—the maximum likelihood hypothesis assigns zero probability to those events.* Various tricks are used to avoid this problem, such as initializing the counts for each event to 1 instead of zero.

Let us look at another example. Suppose this new candy manufacturer wants to give a little hint to the consumer and uses candy wrappers colored red and green. The *Wrapper* for each candy is selected *probabilistically*, according to some unknown conditional distribution, depending on the flavor. The corresponding probability model is shown in Figure 20.2(b). Notice that it has three parameters: θ, θ_1, and θ_2. With these parameters, the likelihood of seeing, say, a cherry candy in a green wrapper can be obtained from the standard semantics for Bayesian networks (page 495):

$$\begin{aligned} & P(Flavor = cherry,\ Wrapper = green | h_{\theta,\theta_1,\theta_2}) \\ &= P(Flavor = cherry | h_{\theta,\theta_1,\theta_2}) P(Wrapper = green | Flavor = cherry, h_{\theta,\theta_1,\theta_2}) \\ &= \theta \cdot (1 - \theta_1) \ . \end{aligned}$$

Now, we unwrap N candies, of which c are cherries and ℓ are limes. The wrapper counts are as follows: r_c of the cherries have red wrappers and g_c have green, while r_ℓ of the limes have red and g_ℓ have green. The likelihood of the data is given by

$$P(\mathbf{d} | h_{\theta,\theta_1,\theta_2}) = \theta^c (1-\theta)^\ell \cdot \theta_1^{r_c} (1-\theta_1)^{g_c} \cdot \theta_2^{r_\ell} (1-\theta_2)^{g_\ell} \ .$$

This looks pretty horrible, but taking logarithms helps:

$$L = [c \log \theta + \ell \log(1-\theta)] + [r_c \log \theta_1 + g_c \log(1-\theta_1)] + [r_\ell \log \theta_2 + g_\ell \log(1-\theta_2)] \ .$$

The benefit of taking logs is clear: the log likelihood is the sum of three terms, each of which contains a single parameter. When we take derivatives with respect to each parameter and set

them to zero, we get three independent equations, each containing just one parameter:

$$\begin{aligned}
\frac{\partial L}{\partial \theta} &= \frac{c}{\theta} - \frac{\ell}{1-\theta} = 0 & \Rightarrow \quad \theta &= \frac{c}{c+\ell} \\
\frac{\partial L}{\partial \theta_1} &= \frac{r_c}{\theta_1} - \frac{g_c}{1-\theta_1} = 0 & \Rightarrow \quad \theta_1 &= \frac{r_c}{r_c+g_c} \\
\frac{\partial L}{\partial \theta_2} &= \frac{r_\ell}{\theta_2} - \frac{g_\ell}{1-\theta_2} = 0 & \Rightarrow \quad \theta_2 &= \frac{r_\ell}{r_\ell+g_\ell} .
\end{aligned}$$

The solution for θ is the same as before. The solution for θ_1, the probability that a cherry candy has a red wrapper, is the observed fraction of cherry candies with red wrappers, and similarly for θ_2.

These results are very comforting, and it is easy to see that they can be extended to any Bayesian network whose conditional probabilities are represented as tables. The most important point is that, *with complete data, the maximum-likelihood parameter learning problem for a Bayesian network decomposes into separate learning problems, one for each parameter.*[3] The second point is that the parameter values for a variable, given its parents, are just the observed frequencies of the variable values for each setting of the parent values. As before, we must be careful to avoid zeroes when the data set is small.

Naive Bayes models

Probably the most common Bayesian network model used in machine learning is the **naive Bayes** model. In this model, the "class" variable C (which is to be predicted) is the root and the "attribute" variables X_i are the leaves. The model is "naive" because it assumes that the attributes are conditionally independent of each other, given the class. (The model in Figure 20.2(b) is a naive Bayes model with just one attribute.) Assuming Boolean variables, the parameters are

$$\theta = P(C = true), \theta_{i1} = P(X_i = true | C = true), \theta_{i2} = P(X_i = true | C = false).$$

The maximum-likelihood parameter values are found in exactly the same way as for Figure 20.2(b). Once the model has been trained in this way, it can be used to classify new examples for which the class variable C is unobserved. With observed attribute values $x_1, \ldots, x_n$, the probability of each class is given by

$$\mathbf{P}(C|x_1, \ldots, x_n) = \alpha \, \mathbf{P}(C) \prod_i \mathbf{P}(x_i|C) .$$

A deterministic prediction can be obtained by choosing the most likely class. Figure 20.3 shows the learning curve for this method when it is applied to the restaurant problem from Chapter 18. The method learns fairly well but not as well as decision-tree learning; this is presumably because the true hypothesis—which is a decision tree—is not representable exactly using a naive Bayes model. Naive Bayes learning turns out to do surprisingly well in a wide range of applications; the boosted version (Exercise 20.5) is one of the most effective general-purpose learning algorithms. Naive Bayes learning scales well to very large problems: with n Boolean attributes, there are just $2n + 1$ parameters, and *no search is required to find h_{ML}, the maximum-likelihood naive Bayes hypothesis.* Finally, naive Bayes learning has no difficulty with noisy data and can give probabilistic predictions when appropriate.

[3] See Exercise 20.7 for the nontabulated case, where each parameter affects several conditional probabilities.

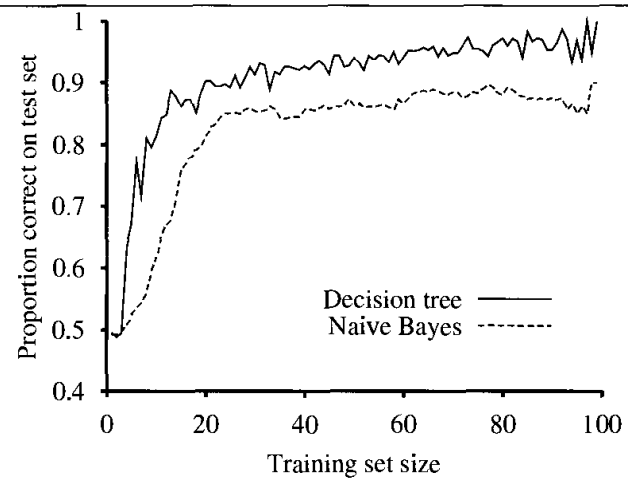


Figure 20.3 The learning curve for naive Bayes learning applied to the restaurant problem from Chapter 18; the learning curve for decision-tree learning is shown for comparison.

Maximum-likelihood parameter learning: Continuous models

Continuous probability models such as the **linear-Gaussian** model were introduced in Section 14.3. Because continuous variables are ubiquitous in real-world applications, it is important to know how to learn continuous models from data. The principles for maximum-likelihood learning are identical to those of the discrete case.

Let us begin with a very simple case: learning the parameters of a Gaussian density function on a single variable. That is, the data are generated as follows:

$$P(x) = \frac{1}{\sqrt{2\pi}\sigma} e^{-\frac{(x-\mu)^2}{2\sigma^2}} .$$

The parameters of this model are the mean μ and the standard deviation σ. (Notice that the normalizing "constant" depends on σ, so we cannot ignore it.) Let the observed values be $x_1, \ldots, x_N$. Then the log likelihood is

$$L = \sum_{j=1}^{N} \log \frac{1}{\sqrt{2\pi}\sigma} e^{-\frac{(x_j-\mu)^2}{2\sigma^2}} = N(-\log\sqrt{2\pi} - \log\sigma) - \sum_{j=1}^{N} \frac{(x_j-\mu)^2}{2\sigma^2} .$$

Setting the derivatives to zero as usual, we obtain

$$\begin{aligned} \frac{\partial L}{\partial \mu} &= -\tfrac{1}{\sigma^2}\textstyle\sum_{j=1}^{N}(x_j-\mu) = 0 & \Rightarrow \quad \mu &= \frac{\sum_j x_j}{N} \\ \frac{\partial L}{\partial \sigma} &= -\tfrac{N}{\sigma} + \tfrac{1}{\sigma^3}\textstyle\sum_{j=1}^{N}(x_j-\mu)^2 = 0 & \Rightarrow \quad \sigma &= \sqrt{\frac{\sum_j (x_j-\mu)^2}{N}} . \end{aligned} \tag{20.4}$$

That is, the maximum-likelihood value of the mean is the sample average and the maximum-likelihood value of the standard deviation is the square root of the sample variance. Again, these are comforting results that confirm "commonsense" practice.

Now consider a linear Gaussian model with one continuous parent X and a continuous child Y. As explained on page 502, Y has a Gaussian distribution whose mean depends linearly on the value of X and whose standard deviation is fixed. To learn the conditional

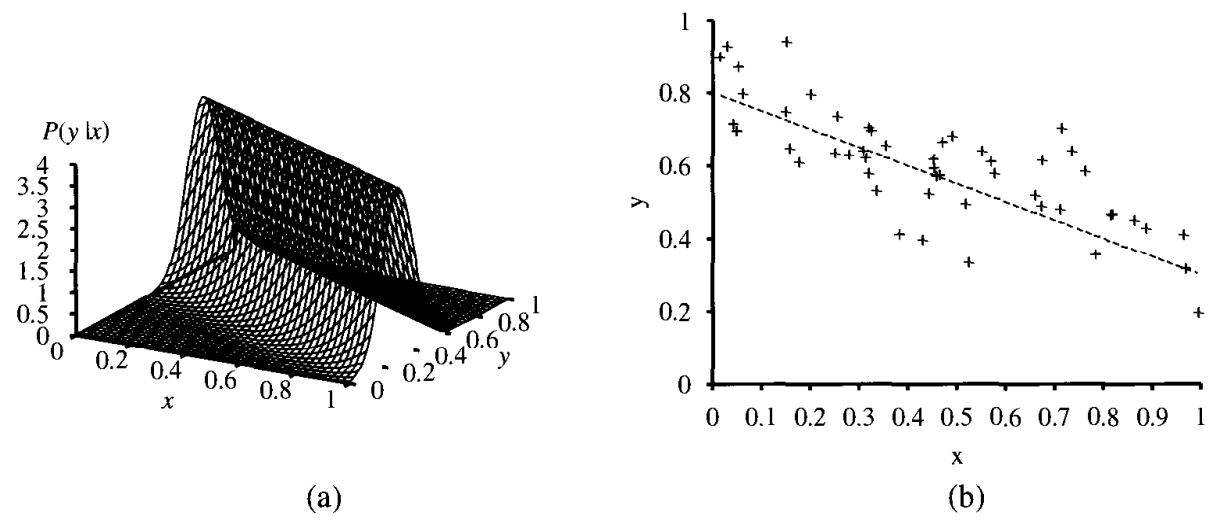


Figure 20.4 (a) A linear Gaussian model described as $y = \theta_1 x + \theta_2$ plus Gaussian noise with fixed variance. (b) A set of 50 data points generated from this model.

distribution $P(Y|X)$, we can maximize the conditional likelihood

$$P(y|x) = \frac{1}{\sqrt{2\pi}\sigma} e^{-\frac{(y-(\theta_1 x+\theta_2))^2}{2\sigma^2}} . \tag{20.5}$$

Here, the parameters are θ_1, θ_2, and σ. The data are a collection of (x_j, y_j) pairs, as illustrated in Figure 20.4. Using the usual methods (Exercise 20.6), we can find the maximum-likelihood values of the parameters. Here, we want to make a different point. If we consider just the parameters θ_1 and θ_2 that define the linear relationship between x and y, it becomes clear that maximizing the log likelihood with respect to these parameters is the same as *minimizing* the numerator in the exponent of Equation (20.5):

$$E = \sum_{j=1}^{N} (y_j - (\theta_1 x_j + \theta_2))^2 .$$

ERROR

SUM OF SQUARED ERRORS

LINEAR REGRESSION

The quantity $(y_j - (\theta_1 x_j + \theta_2))$ is the **error** for (x_j, y_j)—that is, the difference between the actual value y_j and the predicted value $(\theta_1 x_j + \theta_2)$—so E is the well-known **sum of squared errors**. This is the quantity that is minimized by the standard **linear regression** procedure. Now we can understand why: minimizing the sum of squared errors gives the maximum-likelihood straight-line model, *provided that the data are generated with Gaussian noise of fixed variance.*

Bayesian parameter learning

Maximum-likelihood learning gives rise to some very simple procedures, but it has some serious deficiencies with small data sets. For example, after seeing one cherry candy, the maximum-likelihood hypothesis is that the bag is 100% cherry (i.e., $\theta = 1.0$). Unless one's hypothesis prior is that bags must be either all cherry or all lime, this is not a reasonable conclusion. The Bayesian approach to parameter learning places a hypothesis prior over the possible values of the parameters and updates this distribution as data arrive.

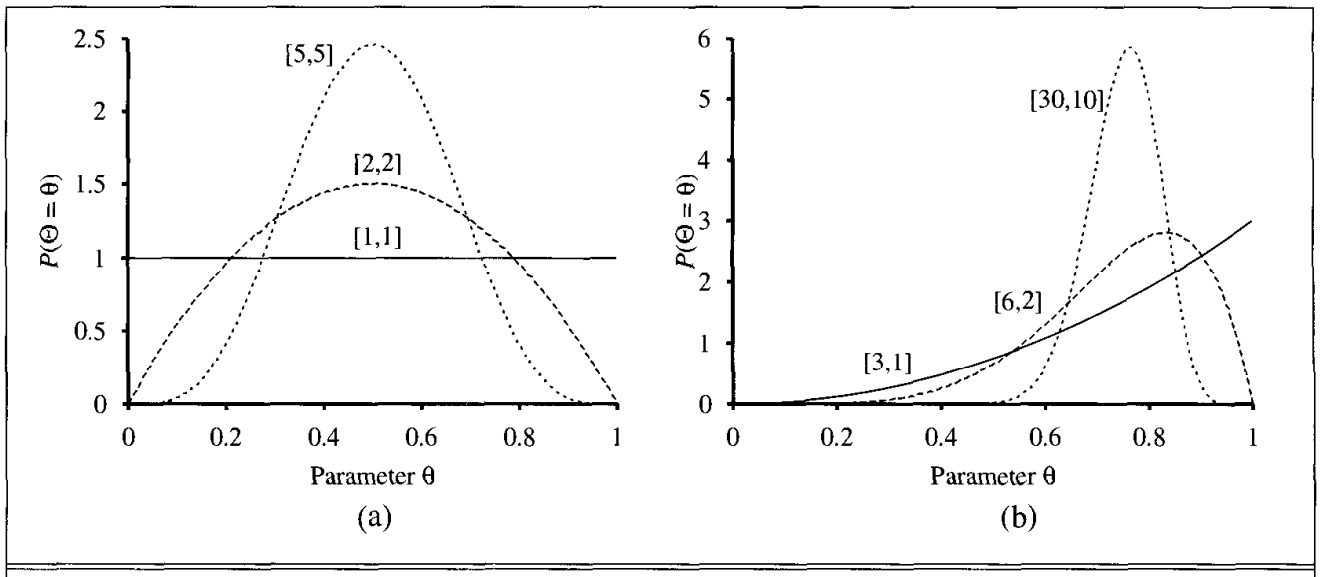


Figure 20.5 Examples of the beta$[a, b]$ distribution for different values of $[a, b]$.

The candy example in Figure 20.2(a) has one parameter, θ: the probability that a randomly selected piece of candy is cherry flavored. In the Bayesian view, θ is the (unknown) value of a random variable Θ; the hypothesis prior is just the prior distribution $\mathbf{P}(\Theta)$. Thus, $P(\Theta = \theta)$ is the prior probability that the bag has a fraction θ of cherry candies.

If the parameter θ can be any value between 0 and 1, then $\mathbf{P}(\Theta)$ must be a continuous distribution that is nonzero only between 0 and 1 and that integrates to 1. The uniform density $P(\theta) = U[0, 1](\theta)$ is one candidate. (See Chapter 13.) It turns out that the uniform density is a member of the family of **beta distributions**. Each beta distribution is defined by two **hyperparameters**[4] a and b such that

BETA DISTRIBUTIONS

HYPERPARAMETER

$$\text{beta}[a, b](\theta) = \alpha\, \theta^{a-1}(1-\theta)^{b-1}\,, \tag{20.6}$$

for θ in the range $[0, 1]$. The normalization constant α depends on a and b. (See Exercise 20.8.) Figure 20.5 shows what the distribution looks like for various values of a and b. The mean value of the distribution is $a/(a + b)$, so larger values of a suggest a belief that Θ is closer to 1 than to 0. Larger values of $a + b$ make the distribution more peaked, suggesting greater certainty about the value of Θ. Thus, the beta family provides a useful range of possibilities for the hypothesis prior.

Besides its flexibility, the beta family has another wonderful property: if Θ has a prior beta$[a, b]$, then, after a data point is observed, the posterior distribution for Θ is also a beta distribution. The beta family is called the **conjugate prior** for the family of distributions for a Boolean variable.[5] Let's see how this works. Suppose we observe a cherry candy; then

CONJUGATE PRIOR

$$\begin{aligned} P(\theta|D_1 = cherry) &= \alpha\, P(D_1 = cherry|\theta)P(\theta) \\ &= \alpha'\, \theta \cdot \text{beta}[a, b](\theta) = \alpha'\, \theta \cdot \theta^{a-1}(1-\theta)^{b-1} \\ &= \alpha'\, \theta^{a}(1-\theta)^{b-1} = \text{beta}[a+1, b](\theta)\,. \end{aligned}$$

[4] They are called hyperparameters because they parameterize a distribution over θ, which is itself a parameter.

[5] Other conjugate priors include the **Dirichlet** family for the parameters of a discrete multivalued distribution and the **Normal–Wishart** family for the parameters of a Gaussian distribution. See Bernardo and Smith (1994).

Thus, after seeing a cherry candy, we simply increment the a parameter to get the posterior; similarly, after seeing a lime candy, we increment the b parameter. Thus, we can view the a and b hyperparameters as **virtual counts**, in the sense that a prior $\text{beta}[a, b]$ behaves exactly as if we had started out with a uniform prior $\text{beta}[1, 1]$ and seen $a - 1$ actual cherry candies and $b - 1$ actual lime candies.

VIRTUAL COUNTS

By examining a sequence of beta distributions for increasing values of a and b, keeping the proportions fixed, we can see vividly how the posterior distribution over the parameter Θ changes as data arrive. For example, suppose the actual bag of candy is 75% cherry. Figure 20.5(b) shows the sequence $\text{beta}[3, 1]$, $\text{beta}[6, 2]$, $\text{beta}[30, 10]$. Clearly, the distribution is converging to a narrow peak around the true value of Θ. For large data sets, then, Bayesian learning (at least in this case) converges to give the same results as maximum-likelihood learning.

The network in Figure 20.2(b) has three parameters, θ, θ_1, and θ_2, where θ_1 is the probability of a red wrapper on a cherry candy and θ_2 is the probability of a red wrapper on a lime candy. The Bayesian hypothesis prior must cover all three parameters—that is, we need to specify $\mathbf{P}(\Theta, \Theta_1, \Theta_2)$. Usually, we assume **parameter independence**:

PARAMETER INDEPENDENCE

$$\mathbf{P}(\Theta, \Theta_1, \Theta_2) = \mathbf{P}(\Theta)\mathbf{P}(\Theta_1)\mathbf{P}(\Theta_2) \ .$$

With this assumption, each parameter can have its own beta distribution that is updated separately as data arrive.

Once we have the idea that unknown parameters can be represented by random variables such as Θ, it is natural to incorporate them into the Bayesian network itself. To do this, we also need to make copies of the variables describing each instance. For example, if we have observed three candies then we need $Flavor_1$, $Flavor_2$, $Flavor_3$ and $Wrapper_1$, $Wrapper_2$, $Wrapper_3$. The parameter variable Θ determines the probability of each $Flavor_i$ variable:

$$P(Flavor_i = cherry | \Theta = \theta) = \theta \ .$$

Similarly, the wrapper probabilities depend on Θ_1 and Θ_2, For example,

$$P(Wrapper_i = red | Flavor_i = cherry, \Theta_1 = \theta_1) = \theta_1 \ .$$

Now, the entire Bayesian learning process can be formulated as an *inference* problem in a suitably constructed Bayes net, as shown in Figure 20.6. Prediction for a new instance is done simply by adding new instance variables to the network, some of which are queried. This formulation of learning and prediction makes it clear that Bayesian learning requires no extra "principles of learning." Furthermore, *there is, in essence, just one learning algorithm,* i.e., the inference algorithm for Bayesian networks.

Learning Bayes net structures

So far, we have assumed that the structure of the Bayes net is given and we are just trying to learn the parameters. The structure of the network represents basic causal knowledge about the domain that is often easy for an expert, or even a naive user, to supply. In some cases, however, the causal model may be unavailable or subject to dispute—for example, certain corporations have long claimed that smoking does not cause cancer—so it is important to

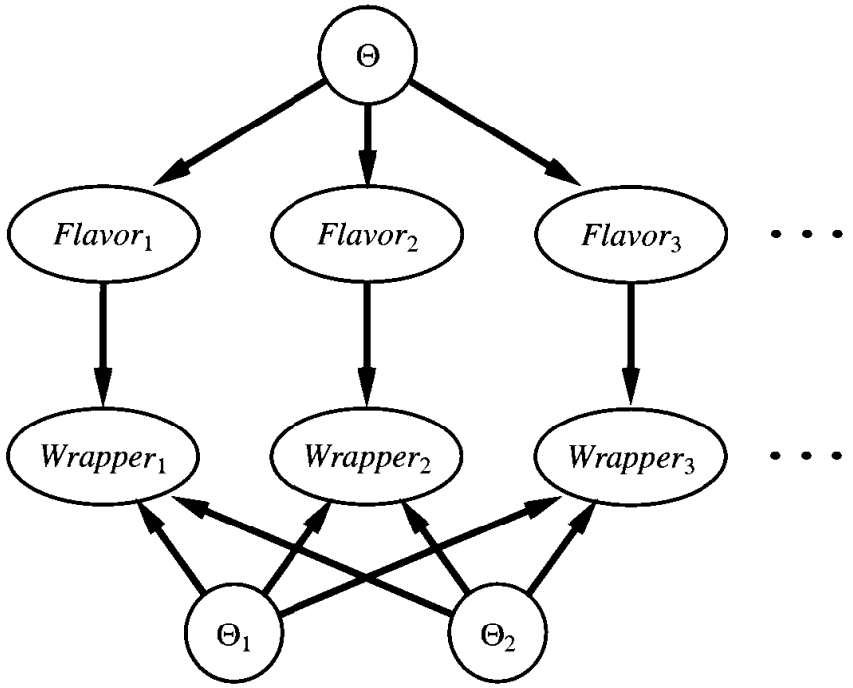


Figure 20.6 A Bayesian network that corresponds to a Bayesian learning process. Posterior distributions for the parameter variables Θ, Θ_1, and Θ_2 can be inferred from their prior distributions and the evidence in the $Flavor_i$ and $Wrapper_i$ variables.

understand how the structure of a Bayes net can be learned from data. At present, structural learning algorithms are in their infancy, so we will give only a brief sketch of the main ideas.

The most obvious approach is to *search* for a good model. We can start with a model containing no links and begin adding parents for each node, fitting the parameters with the methods we have just covered and measuring the accuracy of the resulting model. Alternatively, we can start with an initial guess at the structure and use hill-climbing or simulated annealing search to make modifications, retuning the parameters after each change in the structure. Modifications can include reversing, adding, or deleting arcs. We must not introduce cycles in the process, so many algorithms assume that an ordering is given for the variables, and that a node can have parents only among those nodes that come earlier in the ordering (just as in the construction process Chapter 14). For full generality, we also need to search over possible orderings.

There are two alternative methods for deciding when a good structure has been found. The first is to test whether the conditional independence assertions implicit in the structure are actually satisfied in the data. For example, the use of a naive Bayes model for the restaurant problem assumes that

$$\mathbf{P}(Fri/Sat, Bar \mid WillWait) = \mathbf{P}(Fri/Sat \mid WillWait)\mathbf{P}(Bar \mid WillWait)$$

and we can check in the data that the same equation holds between the corresponding conditional frequencies. Now, even if the structure describes the true causal nature of the domain, statistical fluctuations in the data set mean that the equation will never be satisfied *exactly*, so we need to perform a suitable statistical test to see if there is sufficient evidence that the independence hypothesis is violated. The complexity of the resulting network will depend

on the threshold used for this test—the stricter the independence test, the more links will be added and the greater the danger of overfitting.

An approach more consistent with the ideas in this chapter is to the degree to which the proposed model explains the data (in a probabilistic sense). We must be careful how we measure this, however. If we just try to find the maximum-likelihood hypothesis, we will end up with a fully connected network, because adding more parents to a node cannot decrease the likelihood (Exercise 20.9). We are forced to penalize model complexity in some way. The MAP (or MDL) approach simply subtracts a penalty from the likelihood of each structure (after parameter tuning) before comparing different structures. The Bayesian approach places a joint prior over structures and parameters. There are usually far too many structures to sum over (superexponential in the number of variables), so most practitioners use MCMC to sample over structures.

Penalizing complexity (whether by MAP or Bayesian methods) introduces an important connection between the optimal structure and the nature of the representation for the conditional distributions in the network. With tabular distributions, the complexity penalty for a node's distribution grows exponentially with the number of parents, but with, say, noisy-OR distributions, it grows only linearly. This means that learning with noisy-OR (or other compactly parameterized) models tends to produce learned structures with more parents than does learning with tabular distributions.

20.3 Learning with Hidden Variables: The EM Algorithm

LATENT VARIABLES

The preceding section dealt with the fully observable case. Many real-world problems have **hidden variables** (sometimes called **latent variables**) which are not observable in the data that are available for learning. For example, medical records often include the observed symptoms, the treatment applied, and perhaps the outcome of the treatment, but they seldom contain a direct observation of the disease itself![6] One might ask, "If the disease is not observed, why not construct a model without it?" The answer appears in Figure 20.7, which shows a small, fictitious diagnostic model for heart disease. There are three observable predisposing factors and three observable symptoms (which are too depressing to name). Assume that each variable has three possible values (e.g., *none*, *moderate*, and *severe*). Removing the hidden variable from the network in (a) yields the network in (b); the total number of parameters increases from 78 to 708. Thus, *latent variables can dramatically reduce the number of parameters required to specify a Bayesian network.* This, in turn, can dramatically reduce the amount of data needed to learn the parameters.

Hidden variables are important, but they do complicate the learning problem. In Figure 20.7(a), for example, it is not obvious how to learn the conditional distribution for *HeartDisease*, given its parents, because we do not know the value of *HeartDisease* in each case; the same problem arises in learning the distributions for the symptoms. This section

[6] Some records contain the diagnosis suggested by the physician, but this is a causal consequence of the symptoms, which are in turn caused by the disease.

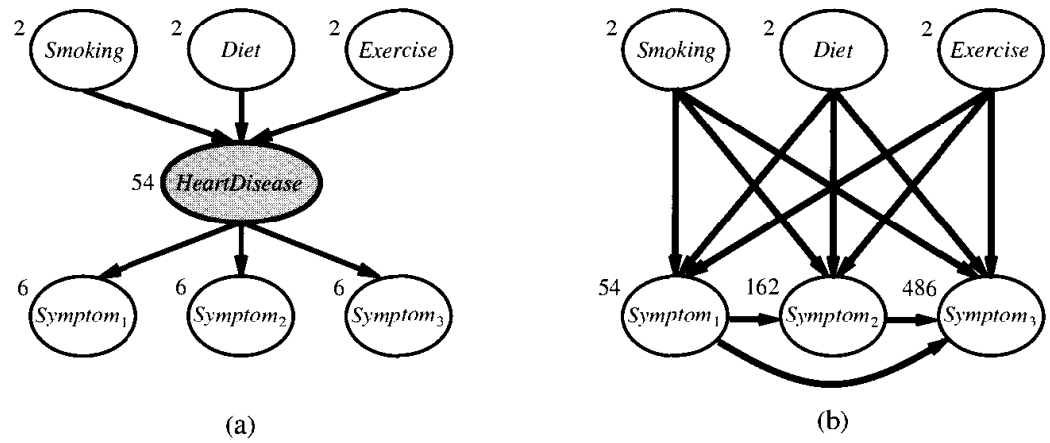


Figure 20.7 (a) A simple diagnostic network for heart disease, which is assumed to be a hidden variable. Each variable has three possible values and is labeled with the number of independent parameters in its conditional distribution; the total number is 78. (b) The equivalent network with *HeartDisease* removed. Note that the symptom variables are no longer conditionally independent given their parents. This network requires 708 parameters.

EXPECTATION-MAXIMIZATION

describes an algorithm called **expectation–maximization**, or EM, that solves this problem in a very general way. We will show three examples and then provide a general description. The algorithm seems like magic at first, but once the intuition has been developed, one can find applications for EM in a huge range of learning problems.

Unsupervised clustering: Learning mixtures of Gaussians

UNSUPERVISED CLUSTERING

Unsupervised clustering is the problem of discerning multiple categories in a collection of objects. The problem is unsupervised because the category labels are not given. For example, suppose we record the spectra of a hundred thousand stars; are there different *types* of stars revealed by the spectra, and, if so, how many and what are their characteristics? We are all familiar with terms such as "red giant" and "white dwarf," but the stars do not carry these labels on their hats—astronomers had to perform unsupervised clustering to identify these categories. Other examples include the identification of species, genera, orders, and so on in the Linnæan taxonomy of organisms and the creation of natural kinds to categorize ordinary objects (see Chapter 10).

Unsupervised clustering begins with data. Figure 20.8(a) shows 500 data points, each of which specifies the values of two continuous attributes. The data points might correspond to stars, and the attributes might correspond to spectral intensities at two particular frequencies. Next, we need to understand what kind of probability distribution might have generated the data. Clustering presumes that the data are generated from a **mixture distribution**, P. Such a distribution has k **components**, each of which is a distribution in its own right. A data point is generated by first choosing a component and then generating a sample from that component. Let the random variable C denote the component, with values $1, \ldots, k$; then the mixture

MIXTURE DISTRIBUTION

COMPONENT

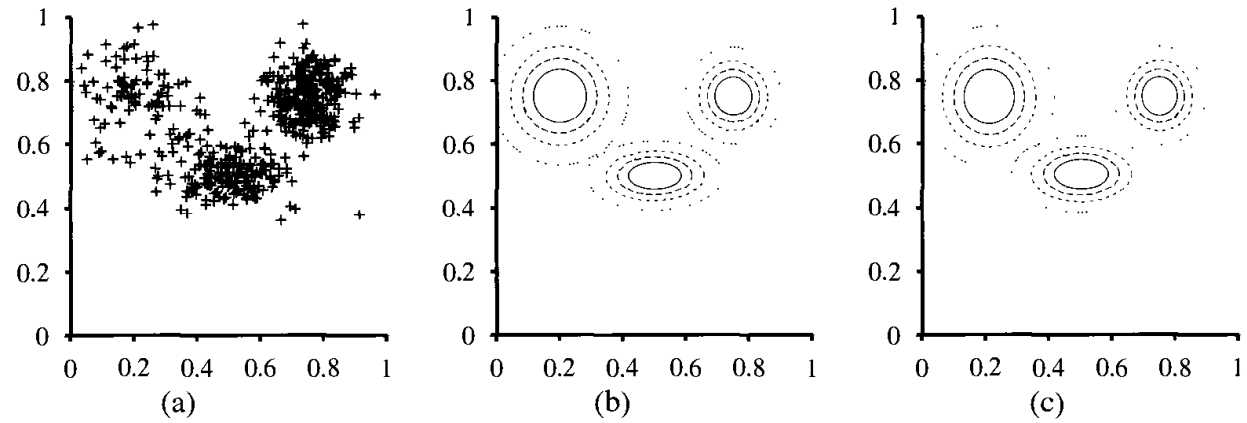


Figure 20.8 (a) 500 data points in two dimensions, suggesting the presence of three clusters. (b) A Gaussian mixture model with three components; the weights (left-to-right) are 0.2, 0.3, and 0.5. The data in (a) were generated from this model. (c) The model reconstructed by EM from the data in (b).

distribution is given by

$$P(\mathbf{x}) = \sum_{i=1}^{k} P(C=i)\, P(\mathbf{x}|C=i)\ ,$$

where $\mathbf{x}$ refers to the values of the attributes for a data point. For continuous data, a natural choice for the component distributions is the multivariate Gaussian, which gives the so-called **mixture of Gaussians** family of distributions. The parameters of a mixture of Gaussians are $w_i = P(C=i)$ (the weight of each component), $\boldsymbol{\mu}_i$ (the mean of each component), and $\boldsymbol{\Sigma}_i$ (the covariance of each component). Figure 20.8(b) shows a mixture of three Gaussians; this mixture is in fact the source of the data in (a).

MIXTURE OF GAUSSIANS

The unsupervised clustering problem, then, is to recover a mixture model like the one in Figure 20.8(b) from raw data like that in Figure 20.8(a). Clearly, if we *knew* which component generated each data point, then it would be easy to recover the component Gaussians: we could just select all the data points from a given component and then apply (a multivariate version of) Equation (20.4) for fitting the parameters of a Gaussian to a set of data. On the other hand, if we *knew* the parameters of each component, then we could, at least in a probabilistic sense, assign each data point to a component. The problem is that we know neither the assignments nor the parameters.

The basic idea of EM in this context is to *pretend* that we know the the parameters of the model and then to infer the probability that each data point belongs to each component. After that, we refit the components to the data, where each component is fitted to the entire data set with each point weighted by the probability that it belongs to that component. The process iterates until convergence. Essentially, we are "completing" the data by inferring probability distributions over the hidden variables—which component each data point belongs to—based on the current model. For the mixture of Gaussians, we initialize the mixture model parameters arbitrarily and then iterate the following two steps:

1. E-step: Compute the probabilities $p_{ij} = P(C = i|\mathbf{x}_j)$, the probability that datum $\mathbf{x}_j$ was generated by component i. By Bayes' rule, we have $p_{ij} = \alpha P(\mathbf{x}_j|C = i)P(C = i)$. The term $P(\mathbf{x}_j|C = i)$ is just the probability at $\mathbf{x}_j$ of the ith Gaussian, and the term $P(C = i)$ is just the weight parameter for the ith Gaussian. Define $p_i = \sum_j p_{ij}$.
2. M-step: Compute the new mean, covariance, and component weights as follows:

$$\begin{aligned}
\boldsymbol{\mu}_i &\leftarrow \sum_j p_{ij}\mathbf{x}_j/p_i \\
\boldsymbol{\Sigma}_i &\leftarrow \sum_j p_{ij}\mathbf{x}_j\mathbf{x}_j^\top/p_i \\
w_i &\leftarrow p_i\ .
\end{aligned}$$

INDICATOR VARIABLE

The E-step, or *expectation* step, can be viewed as computing the expected values p_{ij} of the hidden **indicator variables** Z_{ij}, where Z_{ij} is 1 if datum $\mathbf{x}_j$ was generated by the ith component and 0 otherwise. The M-step, or *maximization* step, finds the new values of the parameters that maximize the log likelihood of the data, given the expected values of the hidden indicator variables.

The final model that EM learns when it is applied to the data in Figure 20.8(a) is shown in Figure 20.8(c); it is virtually indistinguishable from the original model from which the data were generated. Figure 20.9(a) plots the log likelihood of the data according to the current model as EM progresses. There are two points to notice. First, the log likelihood for the final learned model slightly *exceeds* that of the original model, from which the data were generated. This might seem surprising, but it simply reflects the fact that the data were generated randomly and might not provide an exact reflection of the underlying model. The second point is that *EM increases the log likelihood of the data at every iteration.* This fact can be proved in general. Furthermore, under certain conditions, EM can be proven to reach a local maximum in likelihood. (In rare cases, it could reach a saddle point or even a local minimum.) In this sense, EM resembles a gradient-based hill-climbing algorithm, but notice that it has no "step size" parameter!

Things do not always go as well as Figure 20.9(a) might suggest. It can happen, for example, that one Gaussian component shrinks so that it covers just a single data point. Then its variance will go to zero and its likelihood will go to infinity! Another problem is that two components can "merge," acquiring identical means and variances and sharing their data points. These kinds of degenerate local maxima are serious problems, especially in high dimensions. One solution is to place priors on the model parameters and to apply the MAP version of EM. Another is to restart a component with new random parameters if it gets too small or too close to another component. It also helps to initialize the parameters with reasonable values.

Learning Bayesian networks with hidden variables

To learn a Bayesian network with hidden variables, we apply the same insights that worked for mixtures of Gaussians. Figure 20.10 represents a situation in which there are two bags of candies that have been mixed together. Candies are described by three features: in addition to the *Flavor* and the *Wrapper*, some candies have a *Hole* in the middle and some do not.

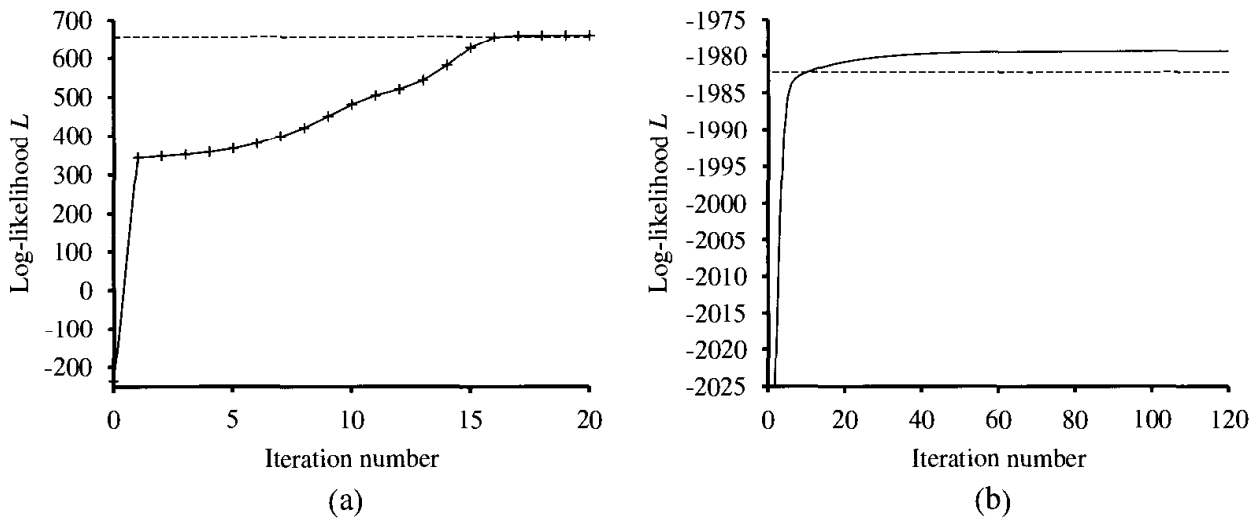


Figure 20.9 Graphs showing the log-likelihood of the data, L, as a function of the EM iteration. The horizontal line shows the log-likelihood according to the true model. (a) Graph for the Gaussian mixture model in Figure 20.8. (b) Graph for the Bayesian network in Figure 20.10(a).

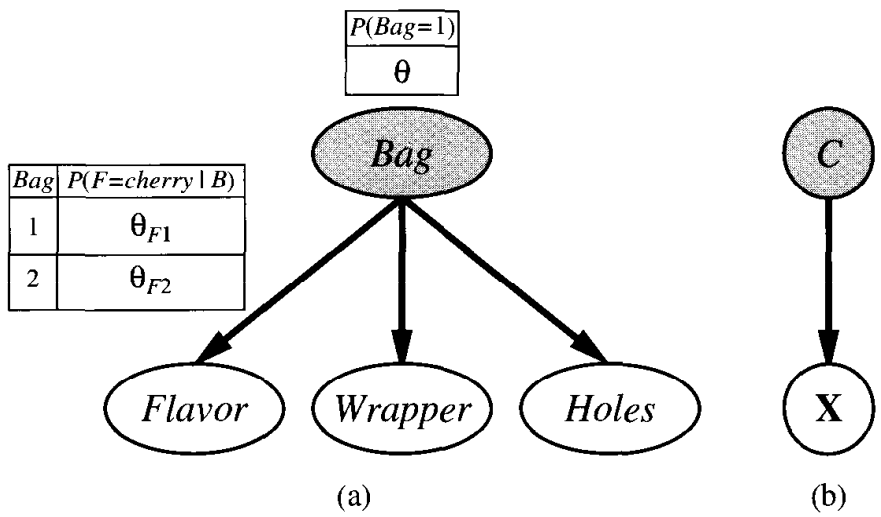


Figure 20.10 (a) A mixture model for candy. The proportions of different flavors, wrappers, and numbers of holes depend on the bag, which is not observed. (b) Bayesian network for a Gaussian mixture. The mean and covariance of the observable variables **X** depend on the component C.

The distribution of candies in each bag is described by a **naive Bayes** model: the features are independent, given the bag, but the conditional probability distribution for each feature depends on the bag. The parameters are as follows: θ is the prior probability that a candy comes from Bag 1; θ_{F1} and θ_{F2} are the probabilities that the flavor is cherry, given that the candy comes from Bag 1 and Bag 2 respectively; θ_{W1} and θ_{W2} give the probabilities that the wrapper is red; and θ_{H1} and θ_{H2} give the probabilities that the candy has a hole. Notice that

the overall model is a mixture model. (In fact, we can also model the mixture of Gaussians as a Bayesian network, as shown in Figure 20.10(b).) In the figure, the bag is is a hidden variable because, once the candies have been mixed together, we no longer know which bag each candy came from. In such a case, can we recover the descriptions of the two bags by observing candies from the mixture?

Let us work through an iteration of EM for this problem. First, let's look at the data. We generated 1000 samples from a model whose true parameters are

$$\theta = 0.5,\ \theta_{F1} = \theta_{W1} = \theta_{H1} = 0.8, \theta_{F2} = \theta_{W2} = \theta_{H2} = 0.3\ . \tag{20.7}$$

That is, the candies are equally likely to come from either bag; the first is mostly cherries with red wrappers and holes; the second is mostly limes with green wrappers and no holes. The counts for the eight possible kinds of candy are as follows:

	$W = red$		$W = green$	
	$H = 1$	$H = 0$	$H = 1$	$H = 0$
$F = cherry$	273	93	104	90
$F = lime$	79	100	94	167

We start by initializing the parameters. For numerical simplicity, we will choose[7]

$$\theta^{(0)} = 0.6,\ \theta^{(0)}_{F1} = \theta^{(0)}_{W1} = \theta^{(0)}_{H1} = 0.6, \theta^{(0)}_{F2} = \theta^{(0)}_{W2} = \theta^{(0)}_{H2} = 0.4\ . \tag{20.8}$$

First, let us work on the θ parameter. In the fully observable case, we would estimate this directly from the *observed* counts of candies from bags 1 and 2. Because the bag is a hidden variable, we calculate the *expected* counts instead. The expected count $\hat{N}(Bag = 1)$ is the sum, over all candies, of the probability that the candy came from bag 1:

$$\theta^{(1)} = \hat{N}(Bag = 1)/N = \sum_{j=1}^{N} P(Bag = 1 | flavor_j, wrapper_j, holes_j)/N\ .$$

These probabilities can be computed by any inference algorithm for Bayesian networks. For a naive Bayes model such as the one in our example, we can do the inference "by hand," using Bayes' rule and applying conditional independence:

$$\theta^{(1)} = \frac{1}{N} \sum_{j=1}^{N} \frac{P(flavor_j | Bag = 1) P(wrapper_j | Bag = 1) P(holes_j | Bag = 1) P(Bag = 1)}{\sum_i P(flavor_j | Bag = i) P(wrapper_j | Bag = i) P(holes_j | Bag = i) P(Bag = i)}\ .$$

(Notice that the normalizing constant also depends on the parameters.) Applying this formula to, say, the 273 red-wrapped cherry candies with holes, we get a contribution of

$$\frac{273}{1000} \cdot \frac{\theta^{(0)}_{F1}\theta^{(0)}_{W1}\theta^{(0)}_{H1}\theta^{(0)}}{\theta^{(0)}_{F1}\theta^{(0)}_{W1}\theta^{(0)}_{H1}\theta^{(0)} + \theta^{(0)}_{F2}\theta^{(0)}_{W2}\theta^{(0)}_{H2}(1 - \theta^{(0)})} \approx 0.22797\ .$$

Continuing with the other seven kinds of candy in the table of counts, we obtain $\theta^{(1)} = 0.6124$.

[7] It is better in practice to choose them randomly, to avoid local maxima due to symmetry.

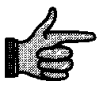
Now let us consider the other parameters, such as θ_{F1}. In the fully observable case, we would estimate this directly from the *observed* counts of cherry and lime candies from bag 1. The *expected* count of cherry candies from bag 1 is given by

$$\sum_{j:Flavor_j = cherry} P(Bag = 1 | Flavor_j = cherry, wrapper_j, holes_j) \ .$$

Again, these probabilities can be calculated by any Bayes net algorithm. Completing this process, we obtain the new values of all the parameters:

$$\begin{aligned} &\theta^{(1)} = 0.6124,\ \theta^{(1)}_{F1} = 0.6684,\ \theta^{(1)}_{W1} = 0.6483,\ \theta^{(1)}_{H1} = 0.6558, \\ &\theta^{(1)}_{F2} = 0.3887, \theta^{(1)}_{W2} = 0.3817,\ \theta^{(1)}_{H2} = 0.3827 \ . \end{aligned} \tag{20.9}$$

The log likelihood of the data increases from about -2044 initially to about -2021 after the first iteration, as shown in Figure 20.9(b). That is, the update improves the likelihood itself by a factor of about $e^{23} \approx 10^{10}$. By the tenth iteration, the learned model is a better fit than the original model ($L = -1982.214$). Thereafter, progress becomes very slow. This is not uncommon with EM, and many practical systems combine EM with a gradient-based algorithm such as Newton–Raphson (see Chapter 4) for the last phase of learning.



The general lesson from this example is that *the parameter updates for Bayesian network learning with hidden variables are directly available from the results of inference on each example. Moreover, only* local *posterior probabilities are needed for each parameter.* For the general case in which we are learning the conditional probability parameters for each variable X_i, given its parents —that is, $\theta_{ijk} = P(X_i = x_{ij} | Pa_i = pa_{ik})$—the update is given by the normalized expected counts as follows:

$$\theta_{ijk} \leftarrow \hat{N}(X_i = x_{ij}, Pa_i = pa_{ik}) / \hat{N}(Pa_i = pa_{ik}) \ .$$

The expected counts are obtained by summing over the examples, computing the probabilities $P(X_i = x_{ij}, Pa_i = pa_{ik})$ for each by using any Bayes net inference algorithm. For the exact algorithms—including variable elimination—all these probabilities are obtainable directly as a by-product of standard inference, with no need for extra computations specific to learning. Moreover, the information needed for learning is available *locally* for each parameter.

Learning hidden Markov models

Our final application of EM involves learning the transition probabilities in hidden Markov models (HMMs). Recall from Chapter 15 that a hidden Markov model can be represented by a dynamic Bayes net with a single discrete state variable, as illustrated in Figure 20.11. Each data point consists of an observation *sequence* of finite length, so the problem is to learn the transition probabilities from a set of observation sequences (or possibly from just one long sequence).

We have already worked out how to learn Bayes nets, but there is one complication: in Bayes nets, each parameter is distinct; in a hidden Markov model, on the other hand, the individual transition probabilities from state i to state j at time t, $\theta_{ijt} = P(X_{t+1} = j | X_t = i)$, are *repeated* across time—that is, $\theta_{ijt} = \theta_{ij}$ for all t. To estimate the transition probability

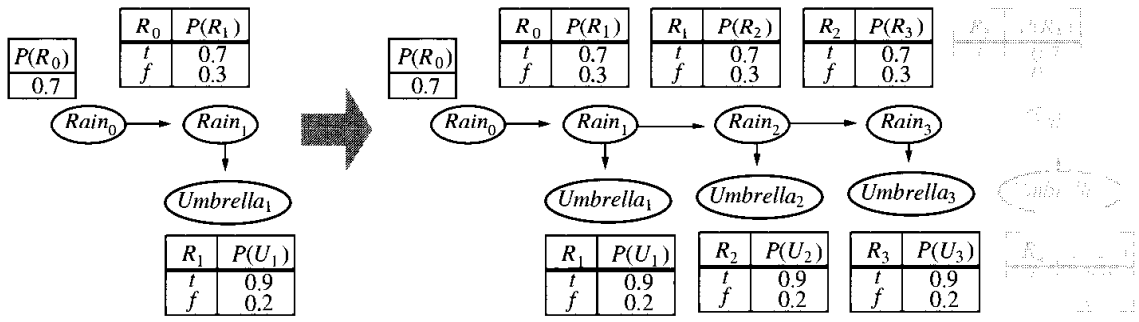


Figure 20.11 An unrolled dynamic Bayesian network that represents a hidden Markov model (repeat of Figure 15.14).

from state i to state j, we simply calculate the expected proportion of times that the system undergoes a transition to state j when in state i:

$$\theta_{ij} \leftarrow \sum_t \hat{N}(X_{t+1}=j, X_{t=i}) / \sum_t \hat{N}(X_{t=i}) \ .$$

Again, the expected counts are computed by any HMM inference algorithm. The **forward–backward** algorithm shown in Figure 15.4 can be modified very easily to compute the necessary probabilities. One important point is that the probabilities required are those obtained by **smoothing** rather than **filtering**; that is, we need to pay attention to subsequent evidence in estimating the probability that a particular transition occurred. As we said in Chapter 15, the evidence in a murder case is usually obtained *after* the crime (i.e., the transition from state i to state j) occurs.

The general form of the EM algorithm

We have seen several instances of the EM algorithm. Each involves computing expected values of hidden variables for each example and then recomputing the parameters, using the expected values as if they were observed values. Let $\mathbf{x}$ be all the observed values in all the examples, let $\mathbf{Z}$ denote all the hidden variables for all the examples, and let $\boldsymbol{\theta}$ be all the parameters for the probability model. Then the EM algorithm is

$$\boldsymbol{\theta}^{(i+1)} = \operatorname*{argmax}_{\boldsymbol{\theta}} \sum_{\mathbf{z}} P(\mathbf{Z}=\mathbf{z}|\mathbf{x}, \boldsymbol{\theta}^{(i)}) L(\mathbf{x}, \mathbf{Z}=\mathbf{z}|\boldsymbol{\theta}) \ .$$

This equation is the EM algorithm in a nutshell. The E-step is the computation of the summation, which is the expectation of the log likelihood of the "completed" data with respect to the distribution $P(\mathbf{Z}=\mathbf{z}|\mathbf{x}, \boldsymbol{\theta}^{(i)})$, which is the posterior over the hidden variables, given the data. The M-step is the maximization of this expected log likelihood with respect to the parameters. For mixtures of Gaussians, the hidden variables are the Z_{ij}s, where Z_{ij} is 1 if example j was generated by component i. For Bayes nets, the hidden variables are the values of the unobserved variables for each example. For HMMs, the hidden variables are the $i \rightarrow j$ transitions. Starting from the general form, it is possible to derive an EM algorithm for a specific application once the appropriate hidden variables have been identified.

As soon as we understand the general idea of EM, it becomes easy to derive all sorts of variants and improvements. For example, in many cases the E-step—the computation of

posteriors over the hidden variables—is intractable, as in large Bayes nets. It turns out that one can use an *approximate* E-step and still obtain an effective learning algorithm. With a sampling algorithm such as MCMC (see Section 14.5), the learning process is very intuitive: each state (configuration of hidden and observed variables) visited by MCMC is treated exactly as if it were a complete observation. Thus, the parameters can be updated directly after each MCMC transition. Other forms of approximate inference, such as variational and loopy methods, have also proven effective for learning very large networks.

Learning Bayes net structures with hidden variables

In Section 20.2, we discussed the problem of learning Bayes net structures with complete data. When hidden variables are taken into consideration, things get more difficult. In the simplest case, the hidden variables are listed along with the observed variables; although their values are not observed, the learning algorithm is told that they exist and must find a place for them in the network structure. For example, an algorithm might try to learn the structure shown in Figure 20.7(a), given the information that *HeartDisease* (a three-valued variable) should be included in the model. If the learning algorithm is not told this information, then there are two choices: either pretend that the data is really complete—which forces the algorithm to learn the parameter-intensive model in Figure 20.7(b)—or *invent* new hidden variables in order to simplify the model. The latter approach can be implemented by including new modification choices in the structure search: in addition to modifying links, the algorithm can add or delete a hidden variable or change its arity. Of course, the algorithm will not know that the new variable it has invented is called *HeartDisease*; nor will it have meaningful names for the values. Fortunately, newly invented hidden variables will usually be connected to pre-existing variables, so a human expert can often inspect the local conditional distributions involving the new variable and ascertain its meaning.

As in the complete-data case, pure maximum-likelihood structure learning will result in a completely connected network (moreover, one with no hidden variables), so some form of complexity penalty is required. We can also apply MCMC to approximate Bayesian learning. For example, we can learn mixtures of Gaussians with an unknown number of components by sampling over the number; the approximate posterior distribution for the number of Gaussians is given by the sampling frequencies of the MCMC process.

So far, the process we have discussed has an outer loop that is a structural search process and an inner loop that is a parametric optimization process. For the complete-data case, the inner loop is very fast—just a matter of extracting conditional frequencies from the data set. When there are hidden variables, the inner loop may involve many iterations of EM or a gradient-based algorithm, and each iteration involves the calculation of posteriors in a Bayes net, which is itself an NP-hard problem. To date, this approach has proved impractical for learning complex models. One possible improvement is the so-called **structural EM** algorithm, which operates in much the same way as ordinary (parametric) EM except that the algorithm can update the structure as well as the parameters. Just as ordinary EM uses the current parameters to compute the expected counts in the E-step and then applies those counts in the M-step to choose new parameters, structural EM uses the current structure to compute

STRUCTURAL EM

expected counts and then applies those counts in the M-step to evaluate the likelihood for potential new structures. (This contrasts with the outer-loop/inner-loop method, which computes new expected counts for each potential structure.) In this way, structural EM may make several structural alterations to the network without once recomputing the expected counts, and is capable of learning nontrivial Bayes net structures. Nonetheless, much work remains to be done before we can say that the structure learning problem is solved.

20.4 INSTANCE-BASED LEARNING

So far, our discussion of statistical learning has focused primarily on fitting the parameters of a *restricted* family of probability models to an *unrestricted* data set. For example, unsupervised clustering using mixtures of Gaussians assumes that the data are explained by the *sum* a *fixed* number of *Gaussian* distributions. We call such methods **parametric learning**. Parametric learning methods are often simple and effective, but assuming a particular restricted family of models often oversimplifies what's happening in the real world, from where the data come. Now, it is true when we have very little data, we cannot hope to learn a complex and detailed model, but it seems silly to keep the hypothesis complexity fixed even when the data set grows very large!

PARAMETRIC LEARNING

NONPARAMETRIC LEARNING

INSTANCE-BASED LEARNING

In contrast to parametric learning, **nonparametric learning** methods allow the hypothesis complexity to grow with the data. The more data we have, the wigglier the hypothesis can be. We will look at two very simple families of nonparametric **instance-based learning** (or **memory-based learning**) methods, so called because they construct hypotheses directly from the training instances themselves.

Nearest-neighbor models

NEAREST-NEIGHBOR

The key idea of **nearest-neighbor** models is that the properties of any particular input point $\mathbf{x}$ are likely to be similar to those of points in the neighborhood of $\mathbf{x}$. For example, if we want to do **density estimation**—that is, estimate the value of an unknown probability density at $\mathbf{x}$—then we can simply measure the density with which points are scattered in the neighborhood of $\mathbf{x}$. This sounds very simple, until we realize that we need to specify exactly what we mean by "neighborhood." If the neighborhood is too small, it won't contain any data points; too large, and it may include *all* the data points, resulting in a density estimate that is the same everywhere. One solution is to define the neighborhood to be just big enough to include k points, where k is large enough to ensure a meaningful estimate. For fixed k, the size of the neighborhood varies—where data are sparse, the neighborhood is large, but where data are dense, the neighborhood is small. Figure 20.12(a) shows an example for data scattered in two dimensions. Figure 20.13 shows the results of k-nearest-neighbor density estimation from these data with $k = 3$, 10, and 40 respectively. For $k = 3$, the density estimate at any point is based on only 3 neighboring points and is highly variable. For $k = 10$, the estimate provides a good reconstruction of the true density shown in Figure 20.12(b). For $k = 40$, the neighborhood becomes too large and structure of the data is altogether lost. In practice, using

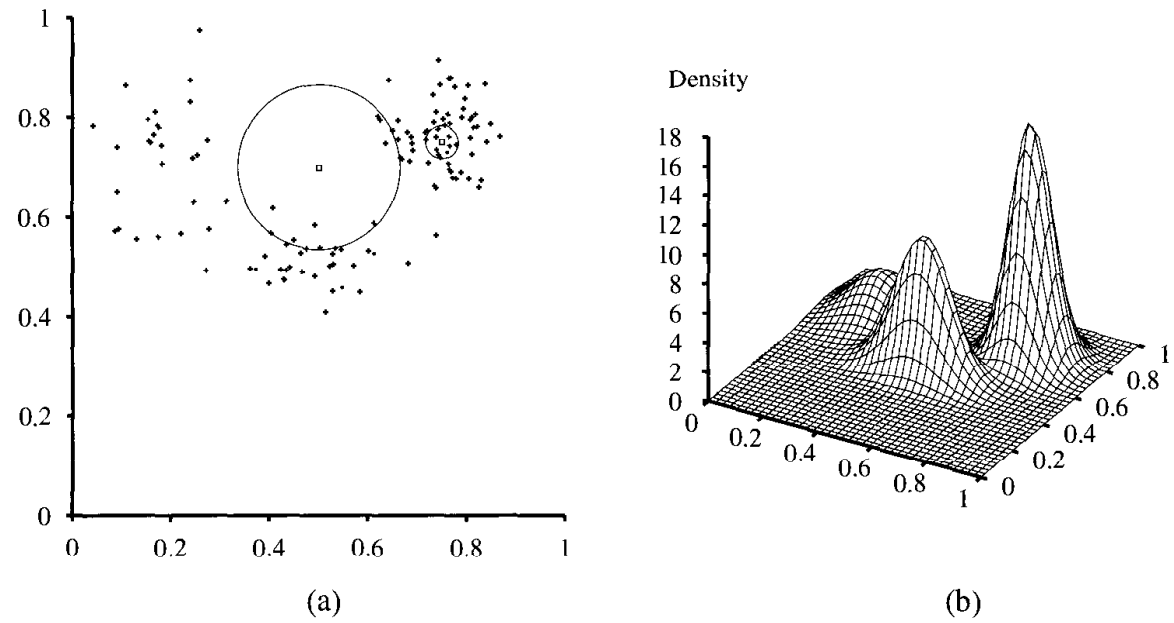


Figure 20.12 (a) A 128-point subsample of the data shown in Figure 20.8(a), together with two query points and their 10-nearest-neighborhoods. (b) A 3-D plot of the mixture of Gaussians from which the data were generated.

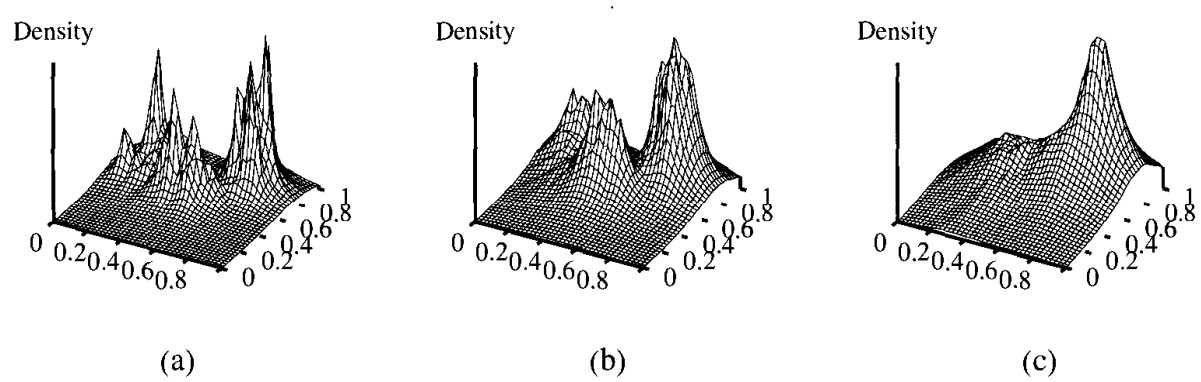


Figure 20.13 Density estimation using k-nearest-neighbors, applied to the data in Figure 20.12(a), for $k = 3$, 10, and 40 respectively.

a value of k somewhere between 5 and 10 gives good results for most low-dimensional data sets. A good value of k can also be chosen by using cross-validation.

To identify the nearest neighbors of a query point, we need a distance metric, $D(\mathbf{x}_1, \mathbf{x}_2)$. The two-dimensional example in Figure 20.12 uses Euclidean distance. This is inappropriate when each dimension of the space is measuring something different—for example, height and weight—because changing the scale of one dimension would change the set of nearest neighbors. One solution is to standardize the scale for each dimension. To do this, we measure

the standard deviation of each feature over the whole data set and express feature values as multiples of the standard deviation for that feature. (This is a special case of the **Mahalanobis distance**, which takes into account the covariance of the features as well.) Finally, for discrete features we can use the **Hamming distance**, which defines $D(\mathbf{x}_1, \mathbf{x}_2)$ to be the number of features on which $\mathbf{x}_1$ and $\mathbf{x}_2$ differ.

MAHALANOBIS DISTANCE

HAMMING DISTANCE

Density estimates like those shown in Figure 20.13 define joint distributions over the input space. Unlike a Bayesian network, however, an instance-based representation cannot contain hidden variables, which means that we cannot perform unsupervised clustering as we did with the mixture-of-Gaussians model. We can still use the density estimate to predict a target value y given input feature values $\mathbf{x}$ by calculating $P(y|\mathbf{x}) = P(y, \mathbf{x})/P(\mathbf{x})$, provided that the training data include values for the target feature.

It is also possible to use the nearest-neighbor idea for direct supervised learning. Given a test example with input $\mathbf{x}$, the output $y = h(\mathbf{x})$ is obtained from the y-values of the k nearest neighbors of $\mathbf{x}$. In the discrete case, we can obtain a single prediction by majority vote. In the continuous case, we can average the k values or do local linear regression, fitting a hyperplane to the k points and predicting the value at $\mathbf{x}$ according to the hyperplane.

The k-nearest-neighbor learning algorithm is very simple to implement, requires little in the way of tuning, and often performs quite well. It is a good thing to try first on a new learning problem. For large data sets, however, we require an efficient mechanism for finding the nearest neighbors of a query point $\mathbf{x}$—simply calculating the distance to every point would take far too long. A variety of ingenious methods have been proposed to make this step efficient by preprocessing the training data. Unfortunately, most of these methods do not scale well with the dimension of the space (i.e., the number of features).

High-dimensional spaces pose an additional problem, namely that nearest neighbors in such spaces are usually a long way away! Consider a data set of size N in the d-dimensional unit hypercube, and assume hypercubic neighborhoods of side b and volume b^d. (The same argument works with hyperspheres, but the formula for the volume of a hypersphere is more complicated.) To contain k points, the average neighborhood must occupy a fraction k/N of the entire volume, which is 1. Hence, $b^d = k/N$, or $b = (k/N)^{1/d}$. So far, so good. Now let the number of features d be 100 and let k be 10 and N be 1,000,000. Then we have $b \approx 0.89$—that is, the neighborhood has to span almost the entire input space! This suggests that nearest-neighbor methods cannot be trusted for high-dimensional data. In low dimensions there is no problem; with $d = 2$ we have $b = 0.003$.

Kernel models

KERNEL MODEL

KERNEL FUNCTION

In a **kernel model**, we view each training instance as generating a little density function—a **kernel function**—of its own. The density estimate as a whole is just the normalized sum of all the little kernel functions. A training instance at $\mathbf{x}_i$ will generate a kernel function $K(\mathbf{x}, \mathbf{x}_i)$ that assigns a probability to each point $\mathbf{x}$ in the space. Thus, the density estimate is

$$P(\mathbf{x}) = \frac{1}{N} \sum_{i=1}^{N} K(\mathbf{x}, \mathbf{x}_i) \ .$$

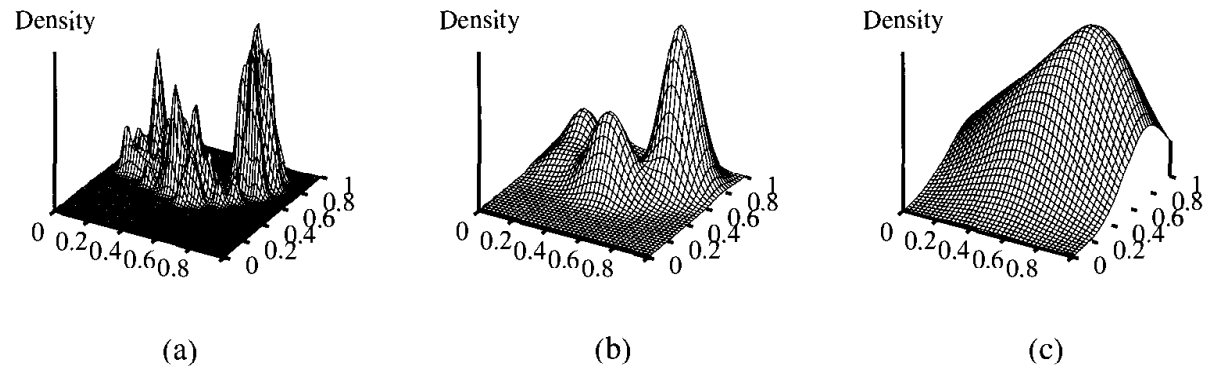


Figure 20.14 Kernel density estimation for the data in Figure 20.12(a), using Gaussian kernels with $w = 0.02$, 0.07, and 0.20 respectively.

The kernel function normally depends only on the *distance* $D(\mathbf{x}, \mathbf{x}_i)$ from $\mathbf{x}$ to the instance $\mathbf{x}_i$. The most popular kernel function is (of course) the Gaussian. For simplicity, we will assume spherical Gaussians with standard deviation w along each axis, i.e.,

$$K(\mathbf{x}, \mathbf{x}_i) = \frac{1}{(w^2\sqrt{2\pi})^d} e^{-\frac{D(\mathbf{x},\mathbf{x}_i)^2}{2w^2}} ,$$

where d is the number of dimensions in $\mathbf{x}$. We still have the problem of choosing a suitable value for w; as before, making the neighborhood too small gives a very spiky estimate—see Figure 20.14(a). In (b), a medium value of w gives a very good reconstruction. In (c), too large a neighborhood results in losing the structure altogether. A good value of w can be chosen by using cross-validation.

Supervised learning with kernels is done by taking a *weighted* combination of *all* the predictions from the training instances. (Compare this with k-nearest-neighbor prediction, which takes an unweighted combination of the nearest k instances.) The weight of the ith instance for a query point $\mathbf{x}$ is given by the value of the kernel $K(\mathbf{x}, \mathbf{x}_i)$. For a discrete prediction, we can take a weighted vote; for a continuous prediction, we can take weighted average or a weighted linear regression. Notice that making predictions with kernels requires looking at *every* training instance. It is possible to combine kernels with nearest-neighbor indexing schemes to make weighted predictions from just the nearby instances.

20.5 Neural Networks

A **neuron** is a cell in the brain whose principal function is the collection, processing, and dissemination of electrical signals. Figure 1.2 on page 11 showed a schematic diagram of a typical neuron. The brain's information-processing capacity is thought to emerge primarily from *networks* of such neurons. For this reason, some of the earliest AI work aimed to create artificial **neural networks**. (Other names for the field include **connectionism**, **parallel dis-**

NEURAL NETWORKS

tributed processing, and **neural computation**.) Figure 20.15 shows a simple mathematical model of the neuron devised by McCulloch and Pitts (1943). Roughly speaking, it "fires" when a linear combination of its inputs exceeds some threshold. Since 1943, much more detailed and realistic models have been developed, both for neurons and for larger systems in the brain, leading to the modern field of **computational neuroscience**. On the other hand, researchers in AI and statistics became interested in the more abstract properties of neural networks, such as their ability to perform distributed computation, to tolerate noisy inputs, and to learn. Although we understand now that other kinds of systems—including Bayesian networks—have these properties, neural networks remain one of the most popular and effective forms of learning system and are worthy of study in their own right.

COMPUTATIONAL NEUROSCIENCE

Units in neural networks

UNITS
LINKS
ACTIVATION
WEIGHT

Neural networks are composed of nodes or **units** (see Figure 20.15) connected by directed **links**. A link from unit j to unit i serves to propagate the **activation** a_j from j to i. Each link also has a numeric **weight** $W_{j,i}$ associated with it, which determines the strength and sign of the connection. Each unit i first computes a weighted sum of its inputs:

$$in_i = \sum_{j=0}^{n} W_{j,i} a_j \ .$$

ACTIVATION FUNCTION

Then it applies an **activation function** g to this sum to derive the output:

$$a_i = g(in_i) = g\left(\sum_{j=0}^{n} W_{j,i} a_j\right) \ . \tag{20.10}$$

BIAS WEIGHT

Notice that we have included a **bias weight** $W_{0,i}$ connected to a fixed input $a_0 = -1$. We will explain its role in a moment.

The activation function g is designed to meet two desiderata. First, we want the unit to be "active" (near +1) when the "right" inputs are given, and "inactive" (near 0) when the "wrong" inputs are given. Second, the activation needs to be *nonlinear*, otherwise the entire neural network collapses into a simple linear function (see Exercise 20.17). Two choices for g

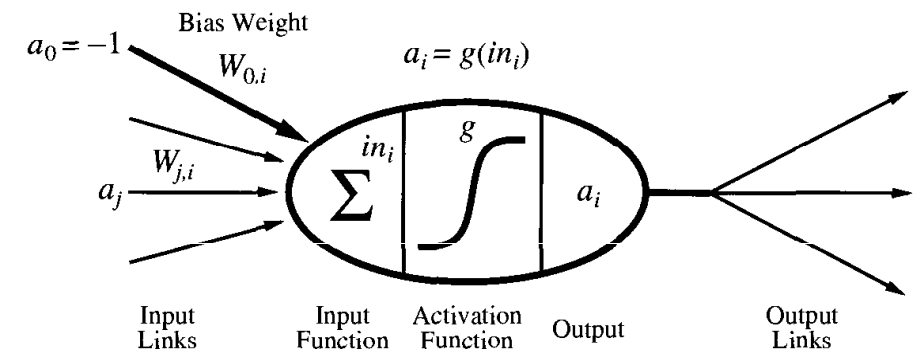


Figure 20.15 A simple mathematical model for a neuron. The unit's output activation is $a_i = g(\sum_{j=0}^{n} W_{j,i} a_j)$, where a_j is the output activation of unit j and $W_{j,i}$ is the weight on the link from unit j to this unit.

THRESHOLD
SIGMOID FUNCTION
LOGISTIC FUNCTION

are shown in Figure 20.16: the **threshold** function and the **sigmoid function** (also known as the **logistic function**). The sigmoid function has the advantage of being differentiable, which we will see later is important for the weight-learning algorithm. Notice that both functions have a threshold (either hard or soft) at zero; the bias weight $W_{0,i}$ sets the *actual* threshold for the unit, in the sense that the unit is activated when the weighted sum of "real" inputs $\sum_{j=1}^{n} W_{j,i}a_j$ (i.e., excluding the bias input) exceeds $W_{0,i}$.

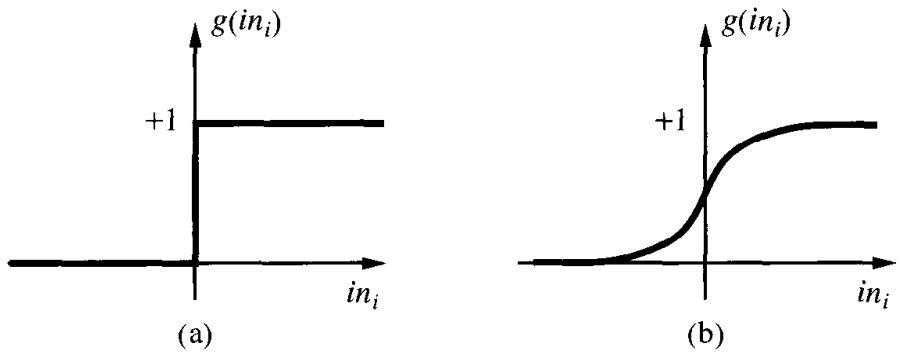


Figure 20.16 (a) The **threshold** activation function, which outputs 1 when the input is positive and 0 otherwise. (Sometimes the **sign** function is used instead, which outputs ± 1 depending on the sign of the input.) (b) The **sigmoid** function $1/(1+e^{-x})$.

We can get a feel for the operation of individual units by comparing them with logic gates. One of the original motivations for the design of individual units (McCulloch and Pitts, 1943) was their ability to represent basic Boolean functions. Figure 20.17 shows how the Boolean functions AND, OR, and NOT can be represented by threshold units with suitable weights. This is important because it means we can use these units to build a network to compute any Boolean function of the inputs.

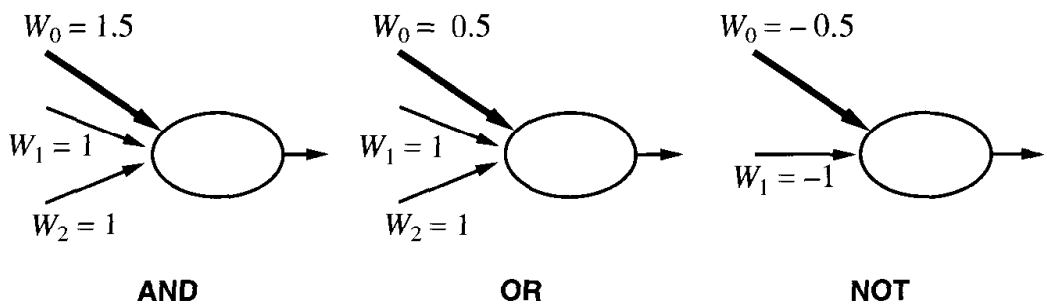


Figure 20.17 Units with a threshold activation function can act as logic gates, given appropriate input and bias weights.

Network structures

FEED-FORWARD NETWORKS
RECURRENT NETWORKS

There are two main categories of neural network structures: acyclic or **feed-forward networks** and cyclic or **recurrent networks**. A feed-forward network represents a function of its current input; thus, it has no internal state other than the weights themselves. A recurrent

network, on the other hand, feeds its outputs back into its own inputs. This means that the activation levels of the network form a dynamical system that may reach a stable state or exhibit oscillations or even chaotic behavior. Moreover, the response of the network to a given input depends on its initial state, which may depend on previous inputs. Hence, recurrent networks (unlike feed-forward networks) can support short-term memory. This makes them more interesting as models of the brain, but also more difficult to understand. This section will concentrate on feed-forward networks; some pointers for further reading on recurrent networks are given at the end of the chapter.

Let us look more closely into the assertion that a feed-forward network represents a function of its inputs. Consider the simple network shown in Figure 20.18, which has two input units, two **hidden units**, and an output unit. (To keep things simple, we have omitted the bias units in this example.) Given an input vector $\mathbf{x} = (x_1, x_2)$, the activations of the input units are set to $(a_1, a_2) = (x_1, x_2)$ and the network computes

HIDDEN UNITS

$$\begin{aligned} a_5 &= g(W_{3,5}a_3 + W_{4,5}a_4) \\ &= g(W_{3,5}g(W_{1,3}a_1 + W_{2,3}a_2) + W_{4,5}g(W_{1,4}a_1 + W_{2,4}a_2)) \,. \end{aligned} \tag{20.11}$$

That is, by expressing the output of each hidden unit as a function of *its* inputs, we have shown that output of the network as a whole, a_5, is a function of the network's inputs. Furthermore, we see that the weights in the network act as *parameters* of this function; writing $\mathbf{W}$ for the parameters, the network computes a function $h_{\mathbf{W}}(\mathbf{x})$. By adjusting the weights, we change the function that the network represents. This is how learning occurs in neural networks.

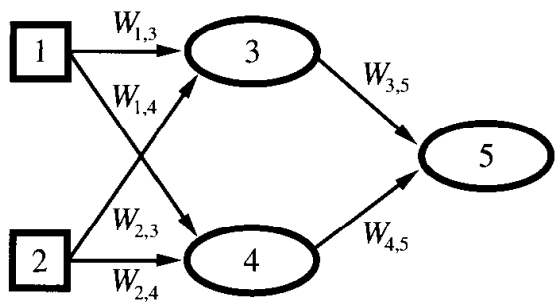


Figure 20.18 A very simple neural network with two inputs, one hidden layer of two units, and one output.

A neural network can be used for classification or regression. For Boolean classification with continuous outputs (e.g., with sigmoid units), it is traditional to have a single output unit, with a value over 0.5 interpreted as one class and a value below 0.5 as the other. For k-way classification, one could divide the single output unit's range into k portions, but it is more common to have k separate output units, with the value of each one representing the relative likelihood of that class given the current input.

LAYERS

Feed-forward networks are usually arranged in **layers**, such that each unit receives input only from units in the immediately preceding layer. In the next two subsections, we will look at single layer networks, which have no hidden units, and multilayer networks, which have one or more layers of hidden units.

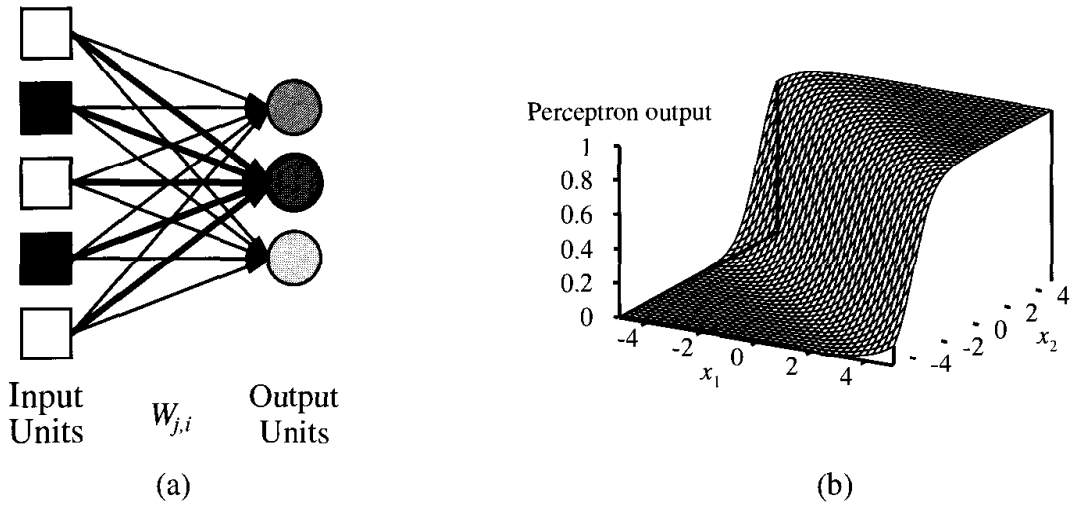


Figure 20.19 (a) A perceptron network consisting of three perceptron output units that share five inputs. Looking at a particular output unit (say the second one, outlined in bold), we see that the weights on its incoming links have no effect on the other output units. (b) A graph of the output of a two-input perceptron unit with a sigmoid activation function.

Single layer feed-forward neural networks (perceptrons)

SINGLE-LAYER NEURAL NETWORK

PERCEPTRON

A network with all the inputs connected directly to the outputs is called a **single-layer neural network**, or a **perceptron** network. Since each output unit is independent of the others—each weight affects only one of the outputs—we can limit our study to perceptrons with a single output unit, as explained in Figure 20.19(a).

Let us begin by examining the hypothesis space that a perceptron can represent. With a threshold activation function, we can view the perceptron as representing a Boolean function. In addition to the elementary Boolean functions AND, OR, and NOT (Figure 20.17), a perceptron can represent some quite "complex" Boolean functions very compactly. For example, the **majority function**, which outputs a 1 only if more than half of its n inputs are 1, can be represented by a perceptron with each $W_j = 1$ and threshold $W_0 = n/2$. A decision tree would need $O(2^n)$ nodes to represent this function.

Unfortunately, there are many Boolean functions that the threshold perceptron cannot represent. Looking at Equation (20.10), we see that the threshold perceptron returns 1 if and only if the weighted sum of its inputs (including the bias) is positive:

$$\sum_{j=0}^{n} W_j x_j > 0 \qquad \text{or} \qquad \mathbf{W} \cdot \mathbf{x} > 0 \,.$$

Now, the equation $\mathbf{W} \cdot \mathbf{x} = 0$ defines a *hyperplane* in the input space, so the perceptron returns 1 if and only if the input is on one side of that hyperplane. For this reason, the threshold perceptron is called a **linear separator**. Figure 20.20(a) and (b) show this hyperplane (a line, in two dimensions) for the perceptron representations of the AND and OR functions of two inputs. Black dots indicate a point in the input space where the value of the function is 1, and white dots indicate a point where the value is 0. The perceptron can represent these functions because there is some line that separates all the white dots from all the black

LINEAR SEPARATOR

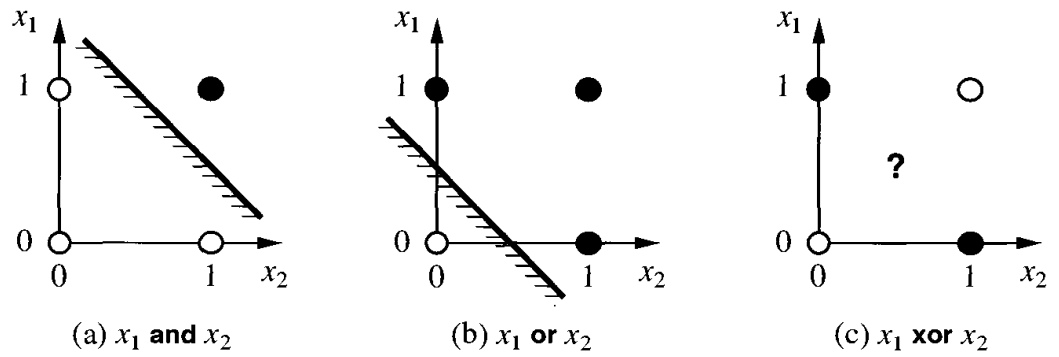


Figure 20.20 Linear separability in threshold perceptrons. Black dots indicate a point in the input space where the value of the function is 1, and white dots indicate a point where the value is 0. The perceptron returns 1 on the region on the non-shaded side of the line. In (c), no such line exists that correctly classifies the inputs.

LINEARLY SEPARABLE

dots. Such functions are called **linearly separable**. Figure 20.20(c) shows an example of a function that is *not* linearly separable—the XOR function. Clearly, there is no way for a threshold perceptron to learn this function. In general, *threshold perceptrons can represent only linearly separable functions.* These constitute just a small fraction of all functions; Exercise 20.14 asks you to quantify this fraction. Sigmoid perceptrons are similarly limited, in the sense that they represent only "soft" linear separators. (See Figure 20.19(b).)

Despite their limited expressive power, threshold perceptrons have some advantages. In particular, *there is a simple learning algorithm that will fit a threshold perceptron to any linearly separable training set.* Rather than present this algorithm, we will *derive* a closely related algorithm for learning in sigmoid perceptrons.

WEIGHT SPACE

The idea behind this algorithm, and indeed behind most algorithms for neural network learning, is to adjust the weights of the network to minimize some measure of the error on the training set. Thus, learning is formulated as an optimization search in **weight space**.[8] The "classical" measure of error is the **sum of squared errors**, which we used for linear regression on page 720. The squared error for a single training example with input $\mathbf{x}$ and true output y is written as

$$E = \frac{1}{2}Err^2 \equiv \frac{1}{2}(y - h_{\mathbf{W}}(\mathbf{x}))^2 ,$$

where $h_{\mathbf{W}}(\mathbf{x})$ is the output of the perceptron on the example.

We can use gradient descent to reduce the squared error by calculating the partial derivative of E with respect to each weight. We have

$$\begin{aligned}\frac{\partial E}{\partial W_j} &= Err \times \frac{\partial Err}{\partial W_j} \\ &= Err \times \frac{\partial}{\partial W_j}\left(y - g\left(\sum_{j=0}^{n} W_j x_j\right)\right) \\ &= -Err \times g'(in) \times x_j ,\end{aligned}$$

[8] See Section 4.4 for general optimization techniques applicable to continuous spaces.

where g' is the derivative of the activation function.[9] In the gradient descent algorithm, where we want to *reduce* E, we update the weight as follows:

$$W_j \leftarrow W_j + \alpha \times Err \times g'(in) \times x_j \,, \tag{20.12}$$

where α is the **learning rate**. Intuitively, this makes a lot of sense. If the error $Err = y - h_{\mathbf{W}}(\mathbf{x})$ is positive, then the network output is too small and so the weights are *increased* for the positive inputs and *decreased* for the negative inputs. The opposite happens when the error is negative.[10]

The complete algorithm is shown in Figure 20.21. It runs the training examples through the net one at a time, adjusting the weights slightly after each example to reduce the error. Each cycle through the examples is called an **epoch**. Epochs are repeated until some stopping criterion is reached—typically, that the weight changes have become very small. Other methods calculate the gradient for the whole training set by adding up all the gradient contributions in Equation (20.12) before updating the weights. The **stochastic gradient** method selects examples randomly from the training set rather than cycling through them.

EPOCH

STOCHASTIC GRADIENT

function PERCEPTRON-LEARNING(*examples*, *network*) **returns** a perceptron hypothesis
  inputs: *examples*, a set of examples, each with input $\mathbf{x} = x_1, \ldots, x_n$ and output y
    network, a perceptron with weights W_j, $j = 0 \ldots n$, and activation function g

  repeat
    for each e **in** *examples* **do**
      $in \leftarrow \sum_{j=0}^{n} W_j \, x_j[e]$
      $Err \leftarrow y[e] - g(in)$
      $W_j \leftarrow W_j + \alpha \times Err \times g'(in) \times x_j[e]$
  until some stopping criterion is satisfied
  return NEURAL-NET-HYPOTHESIS(*network*)

Figure 20.21 The gradient descent learning algorithm for perceptrons, assuming a differentiable activation function g. For threshold perceptrons, the factor $g'(in)$ is omitted from the weight update. NEURAL-NET-HYPOTHESIS returns a hypothesis that computes the network output for any given example.

Figure 20.22 shows the learning curve for a perceptron on two different problems. On the left, we show the curve for learning the majority function with 11 Boolean inputs (i.e., the function outputs a 1 if 6 or more inputs are 1). As we would expect, the perceptron learns the function quite quickly, because the majority function is linearly separable. On the other hand, the decision-tree learner makes no progress, because the majority function is very hard (although not impossible) to represent as a decision tree. On the right, we have the restaurant

[9] For the sigmoid, this derivative is given by $g' = g(1 - g)$.

[10] For threshold perceptrons, where $g'(in)$ is undefined, the original **perceptron learning rule** developed by Rosenblatt (1957) is identical to Equation (20.12) except that $g'(in)$ is omitted. Since $g'(in)$ is the same for all weights, its omission changes only the magnitude and not the direction of the overall weight update for each example.

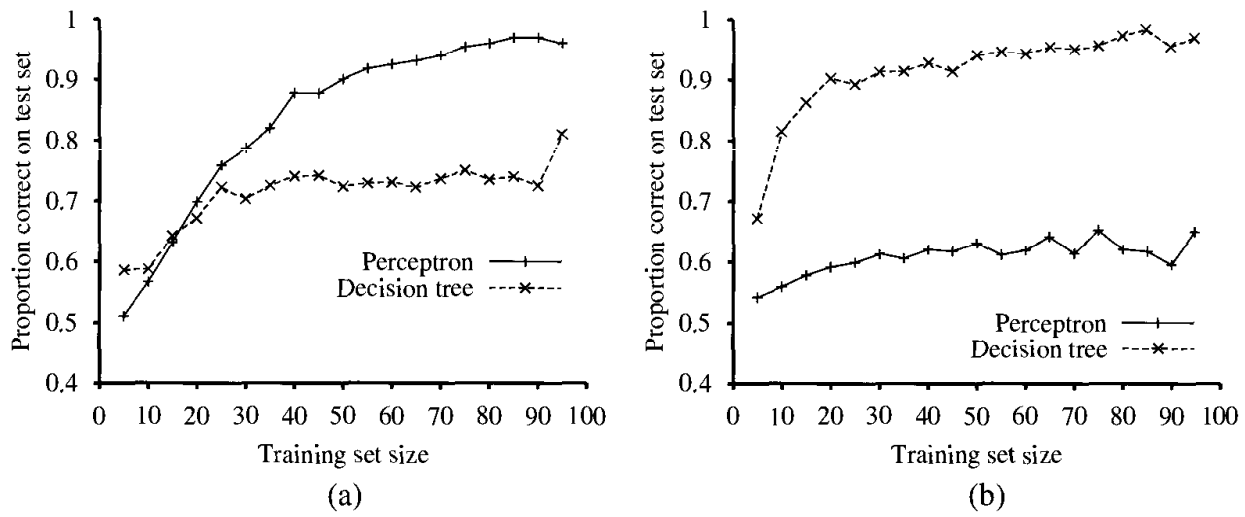


Figure 20.22 Comparing the performance of perceptrons and decision trees. (a) Perceptrons are better at learning the majority function of 11 inputs. (b) Decision trees are better at learning the *WillWait* predicate in the restaurant example.

example. The solution problem is easily represented as a decision tree, but is not linearly separable. The best plane through the data correctly classifies only 65%.

So far, we have treated perceptrons as deterministic functions with possibly erroneous outputs. It is also possible to interpret the output of a sigmoid perceptron as a *probability*—specifically, the probability that the true output is 1 given the inputs. With this interpretation, one can use the sigmoid as a canonical representation for conditional distributions in Bayesian networks (see Section 14.3). One can also derive a learning rule using the standard method of maximizing the (conditional) log likelihood of the data, as described earlier in this chapter. Let's see how this works.

Consider a single training example with true output value T, and let p be the probability returned by the perceptron for this example. If $T = 1$, the conditional probability of the datum is p, and if $T = 0$, the conditional probability of the datum is $(1 - p)$. Now we can use a simple trick to write the log likelihood in a form that is differentiable. The trick is that a 0/1 variable in the *exponent* of an expression acts as an **indicator variable**: p^T is p if $T = 1$ and 1 otherwise; similarly $(1 - p)^{(1-T)}$ is $(1 - p)$ if $T = 0$ and 1 otherwise. Hence, we can write the log likelihood of the datum as

INDICATOR VARIABLE

$$L = \log p^T (1 - p)^{(1-T)} = T \log p + (1 - T) \log(1 - p) \ . \tag{20.13}$$

Thanks to the properties of the sigmoid function, the gradient reduces to a very simple formula (Exercise 20.16):

$$\frac{\partial L}{\partial W_j} = Err \times a_j \ .$$

Notice that *the weight-update vector for maximum likelihood learning in sigmoid perceptrons is essentially identical to the update vector for squared error minimization.* Thus, we could say that perceptrons have a probabilistic interpretation even when the learning rule is derived from a deterministic viewpoint.

Multilayer feed-forward neural networks

Now we will consider networks with hidden units. The most common case involves a single hidden layer,[11] as in Figure 20.24. The advantage of adding hidden layers is that it enlarges the space of hypotheses that the network can represent. Think of each hidden unit as a perceptron that represents a soft threshold function in the input space, as shown in Figure 20.19(b). Then, think of an output unit as as a soft-thresholded linear combination of several such functions. For example, by adding two opposite-facing soft threshold functions and thresholding the result, we can obtain a "ridge" function as shown in Figure 20.23(a). Combining two such ridges at right angles to each other (i.e., combining the outputs from four hidden units), we obtain a "bump" as shown in Figure 20.23(b).

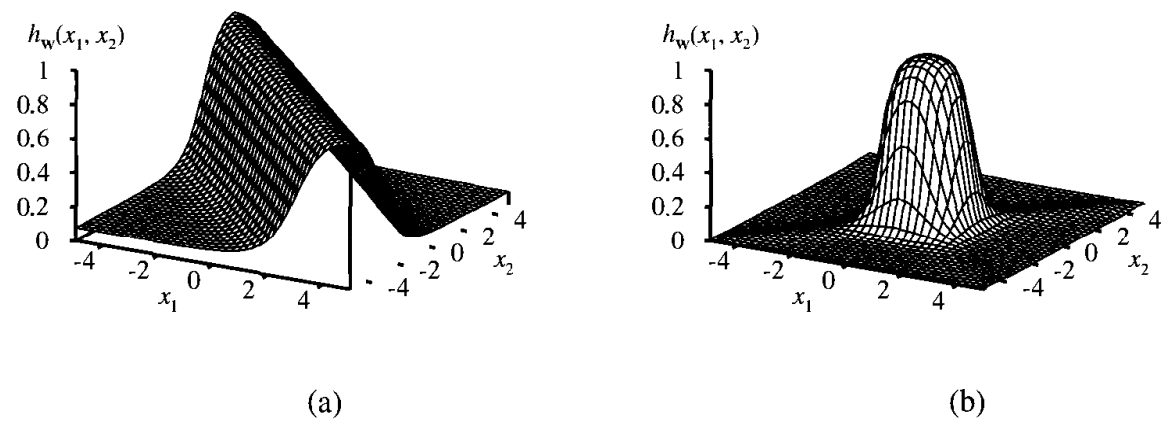


Figure 20.23 (a) The result of combining two opposite-facing soft threshold functions to produce a ridge. (b) The result of combining two ridges to produce a bump.

With more hidden units, we can produce more bumps of different sizes in more places. In fact, with a single, sufficiently large hidden layer, it is possible to represent any continuous function of the inputs with arbitrary accuracy; with two layers, even discontinuous functions can be represented.[12] Unfortunately, for any *particular* network structure, it is harder to characterize exactly which functions can be represented and which ones cannot.

Suppose we want to construct a hidden layer network for the restaurant problem. We have 10 attributes describing each example, so we will need 10 input units. How many hidden units are needed? In Figure 20.24, we show a network with four hidden units. This turns out to be about right for this problem. The problem of choosing the right number of hidden units in advance is still not well understood. (See page 748.)

Learning algorithms for multilayer networks are similar to the perceptron learning algorithm show in Figure 20.21. One minor difference is that we may have several outputs, so

[11] Some people call this a three-layer network, and some call it a two-layer network (because the inputs aren't "real" units). We will avoid confusion and call it a "single-hidden-layer network."

[12] The proof is complex, but the main point is that the required number of hidden units grows exponentially with the number of inputs. For example, $2^n/n$ hidden units are needed to encode all Boolean functions of n inputs.

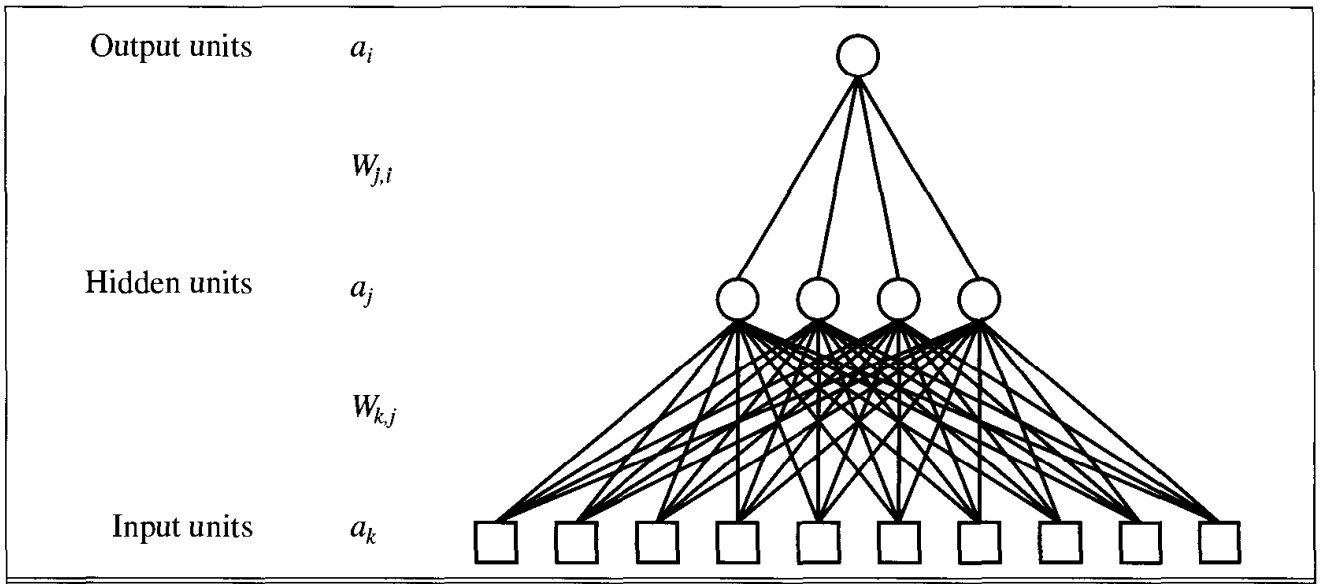


Figure 20.24 A multilayer neural network with one hidden layer and 10 inputs, suitable for the restaurant problem.

we have an output vector $\mathbf{h_W}(\mathbf{x})$ rather than a single value, and each example has an output vector $\mathbf{y}$. The major difference is that, whereas the error $\mathbf{y} - \mathbf{h_W}$ at the output layer is clear, the error at the hidden layers seems mysterious because the training data does not say what value the hidden nodes should have. It turns out that we can **back-propagate** the error from the output layer to the hidden layers. The back-propagation process emerges directly from a derivation of the overall error gradient. First, we will describe the process with an intuitive justification; then, we will show the derivation.

BACK-PROPAGATION

At the output layer, the weight-update rule is identical to Equation (20.12). We have multiple output units, so let Err_i be ith component of the error vector $\mathbf{y} - \mathbf{h_W}$. We will also find it convenient to define a modified error $\Delta_i = Err_i \times g'(in_i)$, so that the weight-update rule becomes

$$W_{j,i} \leftarrow W_{j,i} + \alpha \times a_j \times \Delta_i \ . \tag{20.14}$$

To update the connections between the input units and the hidden units, we need to define a quantity analogous to the error term for output nodes. Here is where we do the error back-propagation. The idea is that hidden node j is "responsible" for some fraction of the error Δ_i in each of the output nodes to which it connects. Thus, the Δ_i values are divided according to the strength of the connection between the hidden node and the output node and are propagated back to provide the Δ_j values for the hidden layer. The propagation rule for the Δ values is the following:

$$\Delta_j = g'(in_j) \sum_i W_{j,i} \Delta_i \ . \tag{20.15}$$

Now the weight-update rule for the weights between the inputs and the hidden layer is almost identical to the update rule for the output layer:

$$W_{k,j} \leftarrow W_{k,j} + \alpha \times a_k \times \Delta_j \ .$$

The back-propagation process can be summarized as follows:

```
function BACK-PROP-LEARNING(examples, network) returns a neural network
  inputs: examples, a set of examples, each with input vector x and output vector y
          network, a multilayer network with L layers, weights W_{j,i}, activation function g

  repeat
     for each e in examples do
        for each node j in the input layer do a_j ← x_j[e]
        for ℓ = 2 to L do
           in_i ← Σ_j W_{j,i} a_j
           a_i ← g(in_i)
        for each node i in the output layer do
           Δ_i ← g'(in_i) × (y_i[e] − a_i)
        for ℓ = L − 1 to 1 do
           for each node j in layer ℓ do
              Δ_j ← g'(in_j) Σ_i W_{j,i} Δ_i
              for each node i in layer ℓ + 1 do
                 W_{j,i} ← W_{j,i} + α × a_j × Δ_i
  until some stopping criterion is satisfied
  return NEURAL-NET-HYPOTHESIS(network)
```

Figure 20.25 The back-propagation algorithm for learning in multilayer networks.

- Compute the Δ values for the output units, using the observed error.
- Starting with output layer, repeat the following for each layer in the network, until the earliest hidden layer is reached:
  - Propagate the Δ values back to the previous layer.
  - Update the weights between the two layers.

The detailed algorithm is shown in Figure 20.25.

For the mathematically inclined, we will now derive the back-propagation equations from first principles. The squared error on a single example is defined as

$$E = \frac{1}{2}\sum_i (y_i - a_i)^2 ,$$

where the sum is over the nodes in the output layer. To obtain the gradient with respect to a specific weight $W_{j,i}$ in the output layer, we need only expand out the activation a_i as all other terms in the summation are unaffected by $W_{j,i}$:

$$\begin{aligned}
\frac{\partial E}{\partial W_{j,i}} &= -(y_i - a_i)\frac{\partial a_i}{\partial W_{j,i}} = -(y_i - a_i)\frac{\partial g(in_i)}{\partial W_{j,i}} \\
&= -(y_i - a_i)g'(in_i)\frac{\partial in_i}{\partial W_{j,i}} = -(y_i - a_i)g'(in_i)\frac{\partial}{\partial W_{j,i}}\left(\sum_j W_{j,i}a_j\right) \\
&= -(y_i - a_i)g'(in_i)a_j = -a_j\Delta_i ,
\end{aligned}$$

with Δ_i defined as before. To obtain the gradient with respect to the $W_{k,j}$ weights connecting the input layer to the hidden layer, we have to keep the entire summation over i because each output value a_i may be affected by changes in $W_{k,j}$. We also have to expand out the activations a_j. We will show the derivation in gory detail because it is interesting to see how the derivative operator propagates back through the network:

$$\begin{aligned}
\frac{\partial E}{\partial W_{k,j}} &= -\sum_i (y_i - a_i)\frac{\partial a_i}{\partial W_{k,j}} = -\sum_i (y_i - a_i)\frac{\partial g(in_i)}{\partial W_{k,j}} \\
&= -\sum_i (y_i - a_i) g'(in_i)\frac{\partial in_i}{\partial W_{k,j}} = -\sum_i \Delta_i \frac{\partial}{\partial W_{k,j}}\left(\sum_j W_{j,i} a_j\right) \\
&= -\sum_i \Delta_i W_{j,i}\frac{\partial a_j}{\partial W_{k,j}} = -\sum_i \Delta_i W_{j,i}\frac{\partial g(in_j)}{\partial W_{k,j}} \\
&= -\sum_i \Delta_i W_{j,i} g'(in_j)\frac{\partial in_j}{\partial W_{k,j}} \\
&= -\sum_i \Delta_i W_{j,i} g'(in_j)\frac{\partial}{\partial W_{k,j}}\left(\sum_k W_{k,j} a_k\right) \\
&= -\sum_i \Delta_i W_{j,i} g'(in_j) a_k = -a_k \Delta_j \ ,
\end{aligned}$$

where Δ_j is defined as before. Thus, we obtain the update rules obtained earlier from intuitive considerations. It is also clear that the process can be continued for networks with more than one hidden layer, which justifies the general algorithm given in Figure 20.25.

Having made it through (or skipped over) all the mathematics, let's see how a single-hidden-layer network performs on the restaurant problem. In Figure 20.26, we show two

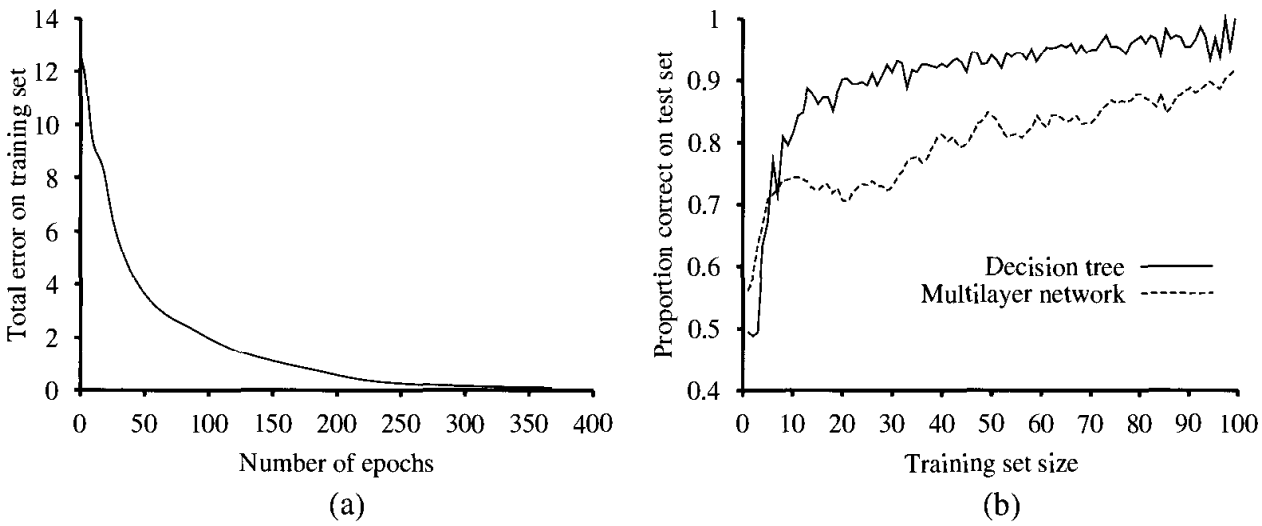


Figure 20.26 (a) Training curve showing the gradual reduction in error as weights are modified over several epochs, for a given set of examples in the restaurant domain. (b) Comparative learning curves showing that decision-tree learning does slightly better than back-propagation in a multilayer network.

TRAINING CURVE

curves. The first is a **training curve**, which shows the mean squared error on a given training set of 100 restaurant examples during the weight-updating process. This demonstrates that the network does indeed converge to a perfect fit to the training data. The second curve is the standard learning curve for the restaurant data. The neural network does learn well, although not quite as fast as decision-tree learning; this is perhaps not surprising, because the data were generated from a simple decision tree in the first place.

Neural networks are capable of far more complex learning tasks of course, although it must be said that a certain amount of twiddling is needed to get the network structure right and to achieve convergence to something close to the global optimum in weight space. There are literally tens of thousands of published applications of neural networks. Section 20.7 looks at one such application in more depth.

Learning neural network structures

So far, we have considered the problem of learning weights, given a fixed network structure; just as with Bayesian networks, we also need to understand how to find the best network structure. If we choose a network that is too big, it will be able to memorize all the examples by forming a large lookup table, but will not necessarily generalize well to inputs that have not been seen before.[13] In other words, like all statistical models, neural networks are subject to **overfitting** when there are too many parameters in the model. We saw this in Figure 18.1 (page 652), where the high-parameter models in (b) and (c) fit all the data, but might not generalize as well as the low-parameter models in (a) and (d).

If we stick to fully connected networks, the only choices to be made concern the number of hidden layers and their sizes. The usual approach is to try several and keep the best. The **cross-validation** techniques of Chapter 18 are needed if we are to avoid **peeking** at the test set. That is, we choose the network architecture that gives the highest prediction accuracy on the validation sets.

If we want to consider networks that are not fully connected, then we need to find some effective search method through the very large space of possible connection topologies.

OPTIMAL BRAIN DAMAGE

The **optimal brain damage** algorithm begins with a fully connected network and removes connections from it. After the network is trained for the first time, an information-theoretic approach identifies an optimal selection of connections that can be dropped. The network is then retrained, and if its performance has not decreased then the process is repeated. In addition to removing connections, it is also possible to remove units that are not contributing much to the result.

Several algorithms have been proposed for growing a larger network from a smaller one.

TILING

One, the **tiling** algorithm, resembles decision-list learning. The idea is to start with a single unit that does its best to produce the correct output on as many of the training examples as possible. Subsequent units are added to take care of the examples that the first unit got wrong. The algorithm adds only as many units as are needed to cover all the examples.

[13] It has been observed that very large networks *do* generalize well *as long as the weights are kept small.* This restriction keeps the activation values in the *linear* region of the sigmoid function $g(x)$ where x is close to zero. This, in turn, means that the network behaves like a linear function (Exercise 20.17) with far fewer parameters.

20.6 KERNEL MACHINES

Our discussion of neural networks left us with a dilemma. Single-layer networks have a simple and efficient learning algorithm, but have very limited expressive power—they can learn only linear decision boundaries in the input space. Multilayer networks, on the other hand, are much more expressive—they can represent general nonlinear functions—but are very hard to train because of the abundance of local minima and the high dimensionality of the weight space. In this section, we will explore a relatively new family of learning methods called **support vector machines** (SVMs) or, more generally, **kernel machines**. To some extent, kernel machines give us the best of both worlds. That is, these methods use an efficient training algorithm *and* can represent complex, nonlinear functions.

SUPPORT VECTOR MACHINE

KERNEL MACHINE

The full treatment of kernel machines is beyond the scope of the book, but we can illustrate the main idea through an example. Figure 20.27(a) shows a two-dimensional input space defined by attributes $\mathbf{x} = (x_1, x_2)$, with positive examples ($y = +1$) inside a circular region and negative examples ($y = -1$) outside. Clearly, there is no linear separator for this problem. Now, suppose we re-express the input data using some computed features—i.e., we map each input vector $\mathbf{x}$ to a new vector of feature values, $F(\mathbf{x})$. In particular, let us use the three features

$$f_1 = x_1^2 , \qquad f_2 = x_2^2 , \qquad f_3 = \sqrt{2}x_1x_2 . \tag{20.16}$$

We will see shortly where these came from, but, for now, just look at what happens. Figure 20.27(b) shows the data in the new, three-dimensional space defined by the three features; the data are *linearly separable* in this space! This phenomenon is actually fairly general: if data are mapped into a space of sufficiently high dimension, then they will always be linearly separable. Here, we used only three dimensions,[14] but if we have N data points then, except in special cases, they will always be separable in a space of $N - 1$ dimensions or more (Exercise 20.21).

So, is that it? Do we just produce loads of computed features and then find a linear separator in the corresponding high-dimensional space? Unfortunately, it's not that easy. Remember that a linear separator in a space of d dimensions is defined by an equation with d parameters, so we are in serious danger of overfitting the data if $d \approx N$, the number of data points. (This is like overfitting data with a high-degree polynomial, which we discussed in Chapter 18.) For this reason, kernel machines usually find the *optimal* linear separator—the one that has the largest **margin** between it and the positive examples on one side and the negative examples on the other. (See Figure 20.28.) It can be shown, using arguments from computational learning theory (Section 18.5), that this separator has desirable properties in terms of robust generalization to new examples.

MARGIN

Now, how do we find this separator? It turns out that this is a **quadratic programming** optimization problem. Suppose we have examples $\mathbf{x}_i$ with classifications $y_i = \pm 1$ and we want to find an optimal separator in the input space; then the quadratic programming problem

QUADRATIC PROGRAMMING

[14] The reader may notice that we could have used just f_1 and f_2, but the 3D mapping illustrates the idea better.

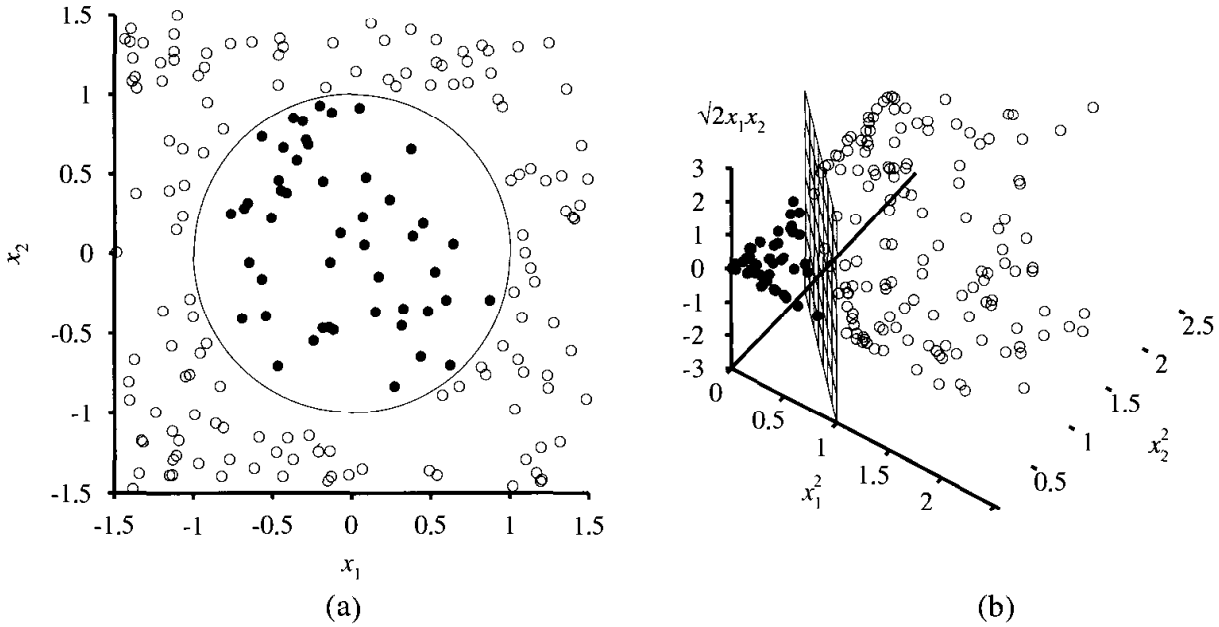


Figure 20.27 (a) A two-dimensional training with positive examples as black circles and negative examples as white circles. The true decision boundary, $x_1^2 + x_2^2 \leq 1$, is also shown. (b) The same data after mapping into a three-dimensional input space $(x_1^2, x_2^2, \sqrt{2}x_1x_2)$. The circular decision boundary in (a) becomes a linear decision boundary in three dimensions.

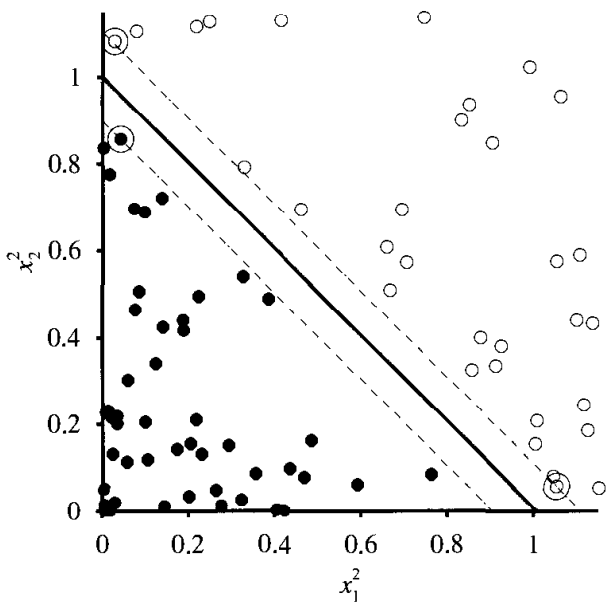


Figure 20.28 A close-up, projected onto the first two dimensions, of the optimal separator shown in Figure 20.27(b). The separator is shown as a heavy line, with the closest points—the **support vectors**—marked with circles. The **margin** is the separation between the positive and negative examples.

is to find values of the parameters α_i that maximize the expression

$$\sum_i \alpha_i - \frac{1}{2}\sum_{i,j} \alpha_i \alpha_j y_i y_j (\mathbf{x}_i \cdot \mathbf{x}_j) \tag{20.17}$$

subject to the constraints $\alpha_i \geq 0$ and $\sum_i \alpha_i y_i = 0$. Although the derivation of this expression is not crucial to the story, it does have two important properties. First, the expression has a single global maximum that can be found efficiently. Second, *the data enter the expression only in the form of dot products of pairs of points.* This second property is also true of the equation for the separator itself; once the optimal α_is have been calculated, it is

$$h(\mathbf{x}) = \text{sign}\left(\sum_i \alpha_i y_i (\mathbf{x} \cdot \mathbf{x}_i)\right) . \tag{20.18}$$

A final important property of the optimal separator defined by this equation is that the weights α_i associated with each data point are *zero* except for those points closest to the separator—the so-called **support vectors**. (They are called this because they "hold up" the separating plane.) Because there are usually many fewer support vectors than data points, the effective number of parameters defining the optimal separator is usually much less than N.

SUPPORT VECTOR

Now, we would not usually expect to find a linear separator in the input space $\mathbf{x}$, but it is easy to see that we can find linear separators in the high-dimensional feature space $F(\mathbf{x})$ simply by replacing $\mathbf{x}_i \cdot \mathbf{x}_j$ in Equation (20.17) with $F(\mathbf{x}_i) \cdot F(\mathbf{x}_j)$. This by itself is not remarkable—replacing $\mathbf{x}$ by $F(\mathbf{x})$ in *any* learning algorithm has the required effect—but the dot product has some special properties. It turns out that $F(\mathbf{x}_i) \cdot F(\mathbf{x}_j)$ can often be computed without first computing F for each point. In our three-dimensional feature space defined by Equation (20.16), a little bit of algebra shows that

$$F(\mathbf{x}_i) \cdot F(\mathbf{x}_j) = (\mathbf{x}_i \cdot \mathbf{x}_j)^2 .$$

The expression $(\mathbf{x}_i \cdot \mathbf{x}_j)^2$ is called a **kernel function**, usually written as $K(\mathbf{x}_i, \mathbf{x}_j)$. In the kernel machine context, this means a function that can be applied to pairs of input data to evaluate dot products in some corresponding feature space. So, we can restate the claim at the beginning of this paragraph as follows: we can find linear separators in the high-dimensional feature space $F(\mathbf{x})$ simply by replacing $\mathbf{x}_i \cdot \mathbf{x}_j$ in Equation (20.17) with a kernel function $K(\mathbf{x}_i, \mathbf{x}_j)$. Thus, we can learn in the high-dimensional space but we compute only kernel functions rather than the full list of features for each data point.

The next step, which should by now be obvious, is to see that there's nothing special about the kernel $K(\mathbf{x}_i, \mathbf{x}_j) = (\mathbf{x}_i \cdot \mathbf{x}_j)^2$. It corresponds to a particular higher-dimensional feature space, but other kernel functions correspond to other feature spaces. A venerable result in mathematics, **Mercer's theorem** (1909), tells us that any "reasonable" [15] kernel function corresponds to *some* feature space. These feature spaces can be very large, even for innocuous-looking kernels. For example, the **polynomial kernel**, $K(\mathbf{x}_i, \mathbf{x}_j) = (1 + \mathbf{x}_i \cdot \mathbf{x}_j)^d$, corresponds to a feature space whose dimension is exponential in d. Using such kernels in Equation (20.17), then, *optimal linear separators can be found efficiently in feature spaces with billions (or, in some cases, infinitely many) dimensions.* The resulting linear separators,

MERCER'S THEOREM

POLYNOMIAL KERNEL

[15] Here, "reasonable" means that the matrix $\mathbf{K}_{ij} = K(\mathbf{x}_i, \mathbf{x}_j)$ is positive definite; see Appendix A.

when mapped back to the original input space, can correspond to arbitrarily wiggly, nonlinear boundaries between the positive and negative examples.

We mentioned in the preceding section that kernel machines excel at handwritten digit recognition; they are rapidly being adopted for other applications—especially those with many input features. As part of this process, many new kernels have been designed that work with strings, trees, and other non-numerical data types. It has also been observed that the kernel method can be applied not only with learning algorithms that find optimal linear separators, but also with any other algorithm that can be reformulated to work only with dot products of pairs of data points, as in Equations 20.17 and 20.18. Once this is done, the dot product is replaced by a kernel function and we have a **kernelized** version of the algorithm. This can be done easily for k-nearest-neighbor and perceptron learning, among others.

KERNELIZATION

20.7 Case Study: Handwritten Digit Recognition

Recognizing handwritten digits is an important problem with many applications, including automated sorting of mail by postal code, automated reading of checks and tax returns, and data entry for hand-held computers. It is an area where rapid progress has been made, in part because of better learning algorithms and in part because of the availability of better training sets. The United States National Institute of Science and Technology (**NIST**) has archived a database of 60,000 labeled digits, each $20 \times 20 = 400$ pixels with 8-bit grayscale values. It has become one of the standard benchmark problems for comparing new learning algorithms. Some example digits are shown in Figure 20.29.

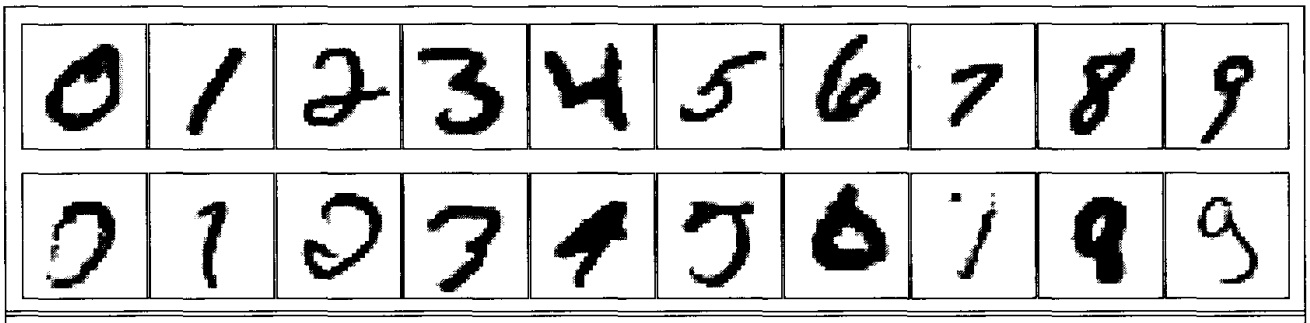


Figure 20.29 Examples from the NIST database of handwritten digits. Top row: examples of digits 0–9 that are easy to identify. Bottom row: more difficult examples of the same digits.

Many different learning approaches have been tried. One of the first, and probably the simplest, is the **3-nearest-neighbor** classifier, which also has the advantage of requiring no training time. As a memory-based algorithm, however, it must store all 60,000 images, and its runtime performance is slow. It achieved a test error rate of 2.4%.

A **single-hidden-layer neural network** was designed for this problem with 400 input units (one per pixel) and 10 output units (one per class). Using cross-validation, it was found that roughly 300 hidden units gave the best performance. With full interconnections between layers, there were a total of 123,300 weights. This network achieved a 1.6% error rate.

A series of **specialized neural networks** called LeNet were devised to take advantage of the structure of the problem—that the input consists of pixels in a two–dimensional array, and that small changes in the position or slant of an image are unimportant. Each network had an input layer of 32×32 units, onto which the 20×20 pixels were centered so that each input unit is presented with a local neighborhood of pixels. This was followed by three layers of hidden units. Each layer consisted of several planes of $n \times n$ arrays, where n is smaller than the previous layer so that the network is down-sampling the input, and where the weights of every unit in a plane are constrained to be identical, so that the plane is acting as a feature detector: it can pick out a feature such as a long vertical line or a short semi-circular arc. The output layer had 10 units. Many versions of this architecture were tried; a representative one had hidden layers with 768, 192, and 30 units, respectively. The training set was augmented by applying affine transformations to the actual inputs: shifting, slightly rotating, and scaling the images. (Of course, the transformations have to be small, or else a 6 will be transformed into a 9!) The best error rate achieved by LeNet was 0.9%.

A **boosted neural network** combined three copies of the LeNet architecture, with the second one trained on a mix of patterns that the first one got 50% wrong, and the third one trained on patterns for which the first two disagreed. During testing, the three nets voted with their weights for each of the ten digits, and the scores are added to determine the winner. The test error rate was 0.7%.

A **support vector machine** (see Section 20.6) with 25,000 support vectors achieved an error rate of 1.1%. This is remarkable because the SVM technique, like the simple nearest-neighbor approach, required almost no thought or iterated experimentation on the part of the developer, yet it still came close to the performance of LeNet, which had had years of development. Indeed, the support vector machine makes no use of the structure of the problem, and would perform just as well if the pixels were presented in a permuted order.

VIRTUAL SUPPORT VECTOR MACHINE

A **virtual support vector machine** starts with a regular SVM and then improves it with a technique that is designed to take advantage of the structure of the problem. Instead of allowing products of all pixel pairs, this approach concentrates on kernels formed from pairs of nearby pixels. It also augments the training set with transformations of the examples, just as LeNet did. A virtual SVM achieved the best error rate recorded to date, 0.56%.

Shape matching is a technique from computer vision used to align corresponding parts of two different images of objects. (See Chapter 24.) The idea is to pick out a set of points in each of the two images, and then compute, for each point in the first image, which point in the second image it corresponds to. From this alignment, we then compute a transformation between the images. The transformation gives us a measure of the distance between the images. This distance measure is better motivated than just counting the number of differing pixels, and it turns out that a 3–nearest neighbor algorithm using this distance measure performs very well. Training on only 20,000 of the 60,000 digits, and using 100 sample points per image extracted from a Canny edge detector, a shape matching classifier achieved 0.63% test error.

Humans are estimated to have an error rate of about 0.2% on this problem. This figure is somewhat suspect because humans have not been tested as extensively as have machine learning algorithms. On a similar data set of digits from the United States Postal Service, human errors were at 2.5%.

The following figure summarizes the error rates, runtime performance, memory requirements, and amount of training time for the seven algorithms we have discussed. It also adds another measure, the percentage of digits that must be rejected to achieve 0.5% error. For example, if the SVM is allowed to reject 1.8% of the inputs—that is, pass them on for someone else to make the final judgment—then its error rate on the remaining 98.2% of the inputs is reduced from 1.1% to 0.5%.

The following table summarizes the error rate and some of the other characteristics of the seven techniques we have discussed.

	3 NN	300 Hidden	LeNet	Boosted LeNet	SVM	Virtual SVM	Shape Match
Error rate (pct.)	2.4	1.6	0.9	0.7	1.1	0.56	0.63
Runtime (millisec/digit)	1000	10	30	50	2000	200	
Memory requirements (Mbyte)	12	.49	.012	.21	11		
Training time (days)	0	7	14	30	10		
% rejected to reach 0.5% error	8.1	3.2	1.8	0.5	1.8		

20.8 SUMMARY

Statistical learning methods range from simple calculation of averages to the construction of complex models such as Bayesian networks and neural networks. They have applications throughout computer science, engineering, neurobiology, psychology, and physics. This chapter has presented some of the basic ideas and given a flavor of the mathematical underpinnings. The main points are as follows:

- **Bayesian learning** methods formulate learning as a form of probabilistic inference, using the observations to update a prior distribution over hypotheses. This approach provides a good way to implement Ockham's razor, but quickly becomes intractable for complex hypothesis spaces.
- **Maximum a posteriori** (MAP) learning selects a single most likely hypothesis given the data. The hypothesis prior is still used and the method is often more tractable than full Bayesian learning.
- **Maximum likelihood** learning simply selects the hypothesis that maximizes the likelihood of the data; it is equivalent to MAP learning with a uniform prior. In simple cases such as linear regression and fully observable Bayesian networks, maximum likelihood solutions can be found easily in closed form. **Naive Bayes** learning is a particularly effective technique that scales well.
- When some variables are hidden, local maximum likelihood solutions can be found using the EM algorithm. Applications include clustering using mixtures of Gaussians, learning Bayesian networks, and learning hidden Markov models.

- Learning the structure of Bayesian networks is an example of **model selection**. This usually involves a discrete search in the space of structures. Some method is required for trading off model complexity against degree of fit.
- **Instance-based models** represent a distribution using the collection of training instances. Thus, the number of parameters grows with the training set. **Nearest-neighbor** methods look at the instances nearest to the point in question, whereas **kernel** methods form a distance-weighted combination of all the instances.
- **Neural networks** are complex nonlinear functions with many parameters. Their parameters can be learned from noisy data and they have been used for thousands of applications.
- A **perceptron** is a feed-forward neural network with no hidden units that can represent only **linearly separable** functions. If the data are linearly separable, a simple weight-update rule can be used to fit the data exactly.
- **Multilayer feed-forward** neural networks can represent any function, given enough units. The **back-propagation** algorithm implements a gradient descent in parameter space to minimize the output error.

Statistical learning continues to be a very active area of research. Enormous strides have been made in both theory and practice, to the point where it is possible to learn almost any model for which exact or approximate inference is feasible.

BIBLIOGRAPHICAL AND HISTORICAL NOTES

The application of statistical learning techniques in AI was an active area of research in the early years (see Duda and Hart, 1973) but became separated from mainstream AI as the latter field concentrated on symbolic methods. It continued in various forms—some explicitly probabilistic, others not—in areas such as **pattern recognition** (Devroye *et al.*, 1996) and **information retrieval** (Salton and McGill, 1983). A resurgence of interest occurred shortly after the introduction of Bayesian network models in the late 1980s; at roughly the same time, a statistical view of neural network learning began to emerge. In the late 1990s, there was a noticeable convergence of interests in machine learning, statistics, and neural networks, centered on methods for creating large probabilistic models from data.

The naive Bayes model is one of the oldest and simplest forms of Bayesian network, dating back to the 1950s. Its origins were mentioned in the notes at the end of Chapter 13. is partially explained by Domingos and Pazzani (1997). A boosted form of naive Bayes learning won the first KDD Cup data mining competition (Elkan, 1997). Heckerman (1998) gives an excellent introduction to the general problem of Bayes net learning. Bayesian parameter learning with Dirichlet priors for Bayesian networks was discussed by Spiegelhalter *et al.* (1993). The BUGS software package (Gilks *et al.*, 1994) incorporates many of these ideas and provides a very powerful tool for formulating and learning complex probability models. The first algorithms for learning Bayes net structures used conditional independence tests (Pearl, 1988; Pearl and Verma, 1991). Spirtes *et al.* (1993) developed a comprehensive approach

and the TETRAD package for Bayes net learning using similar ideas. Algorithmic improvements since then led to a clear victory in the 2001 KDD Cup data mining competition for a Bayes net learning method (Cheng *et al.*, 2002). (The specific task here was a bioinformatics problem with 139,351 features!) A structure-learning approach based on maximizing likelihood was developed by Cooper and Herskovits (1992) and improved by Heckerman *et al.* (1994). Friedman and Goldszmidt (1996) pointed out the influence of the representation of local conditional distributions on the learned structure.

The general problem of learning probability models with hidden variables and missing data was addressed by the EM algorithm (Dempster *et al.*, 1977), which was abstracted from several existing methods including the Baum–Welch algorithm for HMM learning (Baum and Petrie, 1966). (Dempster himself views EM as a schema rather than an algorithm, since a good deal of mathematical work may be required before it can be applied to a new family of distributions.) EM is now one of the most widely used algorithms in science, and McLachlan and Krishnan (1997) devote an entire book to the algorithm and its properties. The specific problem of learning mixture models, including mixtures of Gaussians, is covered by Titterington *et al.* (1985). Within AI, the first successful system that used EM for mixture modeling was AUTOCLASS (Cheeseman *et al.*, 1988; Cheeseman and Stutz, 1996). AUTOCLASS has been applied to a number of real-world scientific classification tasks, including the discovery of new types of stars from spectral data (Goebel *et al.*, 1989) and new classes of proteins and introns in DNA/protein sequence databases (Hunter and States, 1992).

An EM algorithm for learning Bayes nets with hidden variables was developed by Lauritzen (1995). Gradient-based techniques have also proved effective for Bayes nets as well as dynamic Bayes nets (Russell *et al.*, 1995; Binder *et al.*, 1997a). The structural EM algorithm was developed by (Friedman, 1998). The ability to learn the structure of Bayesian networks is closely connected to the issue of recovering *causal* information from data. That is, is it possible to learn Bayes nets in such a way that the recovered network structure indicates real causal influences? For many years, statisticians avoided this question, believing that observational data (as opposed to data generated from experimental trials) could yield only correlational information—after all, any two variables that appear related might in fact be influenced by third, unknown causal factor rather than influencing each other directly. Pearl (2000) has presented convincing arguments to the contrary, showing that there are in fact many cases where causality can be ascertained and developing the **causal network** formalism to express causes and the effects of intervention as well as ordinary conditional probabilities.

CAUSAL NETWORK

Nearest-neighbor models date back at least to (Fix and Hodges, 1951) and have been a standard tool in statistics and pattern recognition ever since. Within AI, they were popularized by (Stanfill and Waltz, 1986), who investigated methods for adapting the distance metric to the data. Hastie and Tibshirani (1996) developed a way to localize the metric to each point in the space, depending on the distribution of data around that point. Efficient indexing schemes for finding nearest neighbors are studied within the algorithms community (see, e.g., Indyk, 2000). Kernel density estimation, also called **Parzen window** density estimation, was investigated initially by Rosenblatt (1956) and Parzen (1962). Since that time, a huge literature has developed investigating the properties of various estimators. Devroye (1987) gives a thorough introduction.

The literature on neural networks is rather too large (approximately 100,000 papers to date) to cover in detail. Cowan and Sharp (1988b, 1988a) survey the early history, beginning with the work of McCulloch and Pitts (1943). Norbert Wiener, a pioneer of cybernetics and control theory (Wiener, 1948), worked with McCulloch and Pitts and influenced a number of young researchers including Marvin Minsky, who may have been the first to develop a working neural network in hardware in 1951 (see Minsky and Papert, 1988, pp. ix–x). Meanwhile, in Britain, W. Ross Ashby (also a pioneer of cybernetics; see Ashby, 1940), Alan Turing, Grey Walter, and others formed the Ratio Club for "those who had Wiener's ideas before Wiener's book appeared." Ashby's *Design for a Brain* (1948, 1952) put forth the idea that intelligence could be created by the use of **homeostatic** devices containing appropriate feedback loops to achieve stable adaptive behavior. Turing (1948) wrote a research report titled *Intelligent Machinery* that begins with the sentence "I propose to investigate the question as to whether it is possible for machinery to show intelligent behaviour" and goes on to describe a recurrent neural network architecture he called "B-type unorganized machines" and an approach to training them. Unfortunately, the report went unpublished until 1969, and was all but ignored until recently.

Frank Rosenblatt (1957) invented the modern "perceptron" and proved the perceptron convergence theorem (1960), although it had been foreshadowed by purely mathematical work outside the context of neural networks (Agmon, 1954; Motzkin and Schoenberg, 1954). Some early work was also done on multilayer networks, including **Gamba perceptrons** (Gamba *et al.*, 1961) and **madalines** (Widrow, 1962). *Learning Machines* (Nilsson, 1965) covers much of this early work and more. The subsequent demise of early perceptron research efforts was hastened (or, the authors later claimed, merely explained) by the book *Perceptrons* (Minsky and Papert, 1969), which lamented the field's lack of mathematical rigor. The book pointed out that single-layer perceptrons could represent only linearly separable concepts and noted the lack of effective learning algorithms for multilayer networks.

The papers in (Hinton and Anderson, 1981), based on a conference in San Diego in 1979, can be regarded as marking the renaissance of connectionism. The two-volume "PDP" (Parallel Distributed Processing) anthology (Rumelhart *et al.*, 1986a) and a short article in *Nature* (Rumelhart *et al.*, 1986b) attracted a great deal of attention—indeed, the number of papers on "neural networks" multiplied by a factor of 200 between 1980–84 and 1990–94. The analysis of neural networks using the physical theory of magnetic spin glasses (Amit *et al.*, 1985) tightened the links between statistical mechanics and neural network theory—providing not only useful mathematical insights but also *respectability*. The back-propagation technique had been invented quite early (Bryson and Ho, 1969) but it was rediscovered several times (Werbos, 1974; Parker, 1985).

Support vector machines were originated in the 1990s (Cortes and Vapnik, 1995) and are now the subject of a fast-growing literature, including textbooks such as Cristianini and Shawe-Taylor (2000). They have proven to be very popular and effective for tasks such as text categorization (Joachims, 2001), bioinformatics research (Brown *et al.*, 2000), and natural language processing, such as the handwritten digit recognition of DeCoste and Scholkopf (2002). A related technique that also uses the "kernel trick" to implicitly represent an exponential feature space is the voted perceptron (Collins and Duffy, 2002).

The probabilistic interpretation of neural networks has several sources, including Baum and Wilczek (1988) and Bridle (1990). The role of the sigmoid function is discussed by Jordan (1995). Bayesian parameter learning for neural networks was proposed by MacKay (1992) and is explored further by Neal (1996). The capacity of neural networks to represent functions was investigated by Cybenko (1988, 1989), who showed that two hidden layers are enough to represent any function and a single layer is enough to represent any *continuous* function. The "optimal brain damage" method for removing useless connections is by LeCun et al. (1989), and Sietsma and Dow (1988) show how to remove useless units. The tiling algorithm for growing larger structures is due to Mézard and Nadal (1989). LeCun *et al.* (1995) survey a number of algorithms for handwritten digit recognition. Improved error rates since then were reported by Belongie *et al.* (2002) for shape matching and DeCoste and Schölkopf (2002) for virtual support vectors.

The complexity of neural network learning has been investigated by researchers in computational learning theory. Early computational results were obtained by Judd (1990), who showed that the general problem of finding a set of weights consistent with a set of examples is NP-complete, even under very restrictive assumptions. Some of the first sample complexity results were obtained by Baum and Haussler (1989), who showed that the number of examples required for effective learning grows as roughly $W \log W$, where W is the number of weights.[16] Since then, a much more sophisticated theory has been developed (Anthony and Bartlett, 1999), including the important result that the representational capacity of a network depends on the *size* of the weights as well as on their number.

RADIAL BASIS FUNCTION

The most popular kind of neural network that we did not cover is the **radial basis function**, or RBF, network. A radial basis function combines a weighted collection of kernels (usually Gaussians, of course) to do function approximation. RBF networks can be trained in two phases: first, an unsupervised clustering approach is used to train the parameters of the Gaussians—the means and variances—are trained, as in Section 20.3. In the second phase, the relative weights of the Gaussians are determined. This is a system of linear equations, which we know how to solve directly. Thus, both phases of RBF training have a nice benefit: the first phase is unsupervised, and thus does not require labelled training data, and the second phase, although supervised, is efficient. See Bishop (1995) for more details.

HOPFIELD NETWORKS

ASSOCIATIVE MEMORY

Recurrent networks, in which units are linked in cycles, were mentioned in the chapter but not explored in depth. **Hopfield networks** (Hopfield, 1982) are probably the best-understood class of recurrent networks. They use *bidirectional* connections with *symmetric* weights (i.e., $W_{i,j} = W_{j,i}$), all of the units are both input and output units, the activation function g is the sign function, and the activation levels can only be ± 1. A Hopfield network functions as an **associative memory**: after the network trains on a set of examples, a new stimulus will cause it to settle into an activation pattern corresponding to the example in the training set that *most closely resembles* the new stimulus. For example, if the training set consists of a set of photographs, and the new stimulus is a small piece of one of the photographs, then the network activation levels will reproduce the photograph from which the piece was

[16] This approximately confirmed "Uncle Bernie's rule." The rule was named after Bernie Widrow, who recommended using roughly ten times as many examples as weights.

taken. Notice that the original photographs are not stored separately in the network; each weight is a partial encoding of all the photographs. One of the most interesting theoretical results is that Hopfield networks can reliably store up to $0.138N$ training examples, where N is the number of units in the network.

BOLTZMANN MACHINES

Boltzmann machines (Hinton and Sejnowski, 1983, 1986) also use symmetric weights, but include hidden units. In addition, they use a *stochastic* activation function, such that the probability of the output being 1 is some function of the total weighted input. Boltzmann machines therefore undergo state transitions that resemble a simulated annealing search (see Chapter 4) for the configuration that best approximates the training set. It turns out that Boltzmann machines are very closely related to a special case of Bayesian networks evaluated with a stochastic simulation algorithm. (See Section 14.5.)

The first application of the ideas underlying kernel machines was by Aizerman *et al.* (1964), but the full development of the theory, under the heading of support vector machines, is due to Vladimir Vapnik and colleagues (Boser *et al.*, 1992; Vapnik, 1998). Cristianini and Shawe-Taylor (2000) and Schölkopf and Smola (2002) provide rigorous introductions; a friendlier exposition appears in the *AI Magazine* article by Cristianini and Schölkopf (2002).

The material in this chapter brings together work from the fields of statistics, pattern recognition, and neural networks, so the story has been told many times in many ways. Good texts on Bayesian statistics include those by DeGroot (1970), Berger (1985), and Gelman *et al.* (1995). Hastie *et al.* (2001) provide an excellent introduction to statistical learning methods. For pattern classification, the classic text for many years has been Duda and Hart (1973), now updated (Duda *et al.*, 2001). For neural nets, Bishop (1995) and Ripley (1996) are the leading texts. The field of computational neuroscience is covered by Dayan and Abbott (2001). The most important conference on neural networks and related topics is the annual NIPS (Neural Information Processing Conference) conference, whose proceedings are published as the series *Advances in Neural Information Processing Systems*. Papers on learning Bayesian networks also appear in the *Uncertainty in AI* and *Machine Learning* conferences and in several statistics conferences. Journals specific to neural networks include *Neural Computation*, *Neural Networks*, and the *IEEE Transactions on Neural Networks*.

EXERCISES

20.1 The data used for Figure 20.1 can be viewed as being generated by h_5. For each of the other four hypotheses, generate a data set of length 100 and plot the corresponding graphs for $P(h_i|d_1,\ldots,d_m)$ and $P(D_{m+1}=lime|d_1,\ldots,d_m)$. Comment on your results.

20.2 Repeat Exercise 20.1, this time plotting the values of $P(D_{m+1}=lime|h_{\text{MAP}})$ and $P(D_{m+1}=lime|h_{\text{ML}})$.

20.3 Suppose that Ann's utilities for cherry and lime candies are c_A and ℓ_A, whereas Bob's utilities are c_B and ℓ_B. (But once Ann has unwrapped a piece of candy, Bob won't buy it.) Presumably, if Bob likes lime candies much more than Ann, it would be wise to sell for Ann

to sell her bag of candies once she is sufficiently sure of its lime content. On the other hand, if Ann unwraps too many candies in the process, the bag will be worth less. Discuss the problem of determining the optimal point at which to sell the bag. Determine the expected utility of the optimal procedure, given the prior distribution from Section 20.1.

20.4 Two statisticians go to the doctor and are both given the same prognosis: A 40% chance that the problem is the deadly disease A, and a 60% chance of the fatal disease B. Fortunately, there are anti-A and anti-B drugs that are inexpensive, 100% effective, and free of side-effects. The statisticians have the choice of taking one drug, both, or neither. What will the first statistician (an avid Bayesian) do? How about the second statistician, who always uses the maximum likelihood hypothesis?

The doctor does some research and discovers that disease B actually comes in two versions, dextro-B and levo-B, which are equally likely and equally treatable by the anti-B drug. Now that there are three hypotheses, what will the two statisticians do?

20.5 Explain how to apply the boosting method of Chapter 18 to naive Bayes learning. Test the performance of the resulting algorithm on the restaurant learning problem.

20.6 Consider m data points (x_j, y_j), where the y_js are generated from the x_js according to the linear Gaussian model in Equation (20.5). Find the values of θ_1, θ_2, and σ that maximize the conditional log likelihood of the data.

20.7 Consider the noisy-OR model for fever described in Section 14.3. Explain how to apply maximum-likelihood learning to fit the parameters of such a model to a set of complete data. (*Hint*: use the chain rule for partial derivatives.)

20.8 This exercise investigates properties of the Beta distribution defined in Equation (20.6).

GAMMA FUNCTION

a. By integrating over the range $[0, 1]$, show that the normalization constant for the distribution $\text{beta}[a, b]$ is given by $\alpha = \Gamma(a + b)/\Gamma(a)\Gamma(b)$ where $\Gamma(x)$ is the **Gamma function**, defined by $\Gamma(x + 1) = x \cdot \Gamma(x)$ and $\Gamma(1) = 1$. (For integer x, $\Gamma(x + 1) = x!$.)

b. Show that the mean is $a/(a + b)$.

c. Find the mode(s) (the most likely value(s) of θ).

d. Describe the distribution $\text{beta}[\epsilon, \epsilon]$ for very small ϵ. What happens as such a distribution is updated?

20.9 Consider an arbitrary Bayesian network, a complete data set for that network, and the likelihood for the data set according to the network. Give a simple proof that the likelihood of the data cannot decrease if we add a new link to the network and recompute the maximum-likelihood parameter values.

20.10 Consider the application of EM to learn the parameters for the network in Figure 20.10(a), given the true parameters in Equation (20.7).

a. Explain why the EM algorithm would not work if there were just two attributes in the model rather than three.

b. Show the calculations for the first iteration of EM starting from Equation (20.8).

c. What happens if we start with all the parameters set to the same value p? (*Hint*: you may find it helpful to investigate this empirically before deriving the general result.)

d. Write out an expression for the log likelihood of the tabulated candy data on page 729 in terms of the parameters, calculate the partial derivatives with respect to each parameter, and investigate the nature of the fixed point reached in part (c).

20.11 Construct by hand a neural network that computes the XOR function of two inputs. Make sure to specify what sort of units you are using.

20.12 Construct a support vector machine that computes the XOR function. It will be convenient to use values of 1 and –1 instead of 1 and 0 for the inputs and for the outputs. So an example looks like $([-1, 1], 1)$ or $([-1, -1], -1)$. It is typical to map an input $\mathbf{x}$ into a space consisting of five dimensions, the two original dimensions x_1 and x_2, and the three combination x_1^2, x_2^2 and $x_1\, x_2$. But for this exercise we will consider only the two dimensions x_1 and $x_1\, x_2$. Draw the four input points in this space, and the maximal margin separator. What is the margin? Now draw the separating line back in the original Euclidean input space.

20.13 A simple perceptron cannot represent XOR (or, generally, the parity function of its inputs). Describe what happens to the weights of a four-input, step-function perceptron, beginning with all weights set to 0.1, as examples of the parity function arrive.

20.14 Recall from Chapter 18 that there are 2^{2^n} distinct Boolean functions of n inputs. How many of these are representable by a threshold perceptron?

20.15 Consider the following set of examples, each with six inputs and one target output:

I_1	1	1	1	1	1	1	1	0	0	0	0	0	0	0
I_2	0	0	0	1	1	0	0	1	1	0	1	0	1	1
I_3	1	1	1	0	1	0	0	1	1	0	0	0	1	1
I_4	0	1	0	0	1	0	0	1	0	1	1	1	0	1
I_5	0	0	1	1	0	1	1	0	1	1	0	0	1	0
I_6	0	0	0	1	0	1	0	1	1	0	1	1	1	0
T	1	1	1	1	1	1	0	1	0	0	0	0	0	0

a. Run the perceptron learning rule on these data and show the final weights.

b. Run the decision tree learning rule, and show the resulting decision tree.

c. Comment on your results.

20.16 Starting from Equation (20.13), show that $\partial L/\partial W_j = Err \times a_j$.

20.17 Suppose you had a neural network with linear activation functions. That is, for each unit the output is some constant c times the weighted sum of the inputs.

a. Assume that the network has one hidden layer. For a given assignment to the weights $\mathbf{W}$, write down equations for the value of the units in the output layer as a function of $\mathbf{W}$ and the input layer $\mathbf{I}$, without any explicit mention to the output of the hidden layer. Show that there is a network with no hidden units that computes the same function.

b. Repeat the calculation in part (a), this time for a network with any number of hidden layers. What can you conclude about linear activation functions?

20.18 Implement a data structure for layered, feed-forward neural networks, remembering to provide the information needed for both forward evaluation and backward propagation. Using this data structure, write a function NEURAL-NETWORK-OUTPUT that takes an example and a network and computes the appropriate output values.

20.19 Suppose that a training set contains only a single example, repeated 100 times. In 80 of the 100 cases, the single output value is 1; in the other 20, it is 0. What will a back-propagation network predict for this example, assuming that it has been trained and reaches a global optimum? (*Hint:* to find the global optimum, differentiate the error function and set to zero.)

20.20 The network in Figure 20.24 has four hidden nodes. This number was chosen somewhat arbitrarily. Run systematic experiments to measure the learning curves for networks with different numbers of hidden nodes. What is the optimal number? Would it be possible to use a cross-validation method to find the best network before the fact?

20.21 Consider the problem of separating N data points into positive and negative examples using a linear separator. Clearly, this can always be done for $N = 2$ points on a line of dimension $d = 1$, regardless of how the points are labelled or where they are located (unless the points are in the same place).

a. Show that it can always be done for $N = 3$ points on a plane of dimension $d = 2$, unless they are collinear.

b. Show that it cannot always be done for $N = 4$ points on a plane of dimension $d = 2$.

c. Show that it can always be done for $N = 4$ points in a space of dimension $d = 3$, unless they are coplanar.

d. Show that it cannot always be done for $N = 5$ points in a space of dimension $d = 3$.

e. The ambitious student may wish to prove that N points in general position (but not $N + 1$ are linearly separable in a space of dimension $N - 1$. From this it follows that the **VC dimension** (see Chapter 18) of linear halfspaces in dimension $N - 1$ is N.

21 REINFORCEMENT LEARNING

In which we examine how an agent can learn from success and failure, from reward and punishment.

21.1 INTRODUCTION

Chapters 18 and 20 covered learning methods that learn functions and probability models from example. In this chapter, we will study how agents can learn *what to do*, particularly when there is no teacher telling the agent what action to take in each circumstance.

For example, we know an agent can learn to play chess by supervised learning—by being given examples of game situations along with the best moves for those situations. But if there is no friendly teacher providing examples, what can the agent do? By trying random moves, the agent can eventually build a predictive model of its environment: what the board will be like after it makes a given move and even how the opponent is likely to reply in a given situation. The problem is this: *without some feedback about what is good and what is bad, the agent will have no grounds for deciding which move to make.* The agent needs to know that something good has happened when it wins and that something bad has happened when it loses. This kind of feedback is called a **reward**, or **reinforcement**. In games like chess, the reinforcement is received only at the end of the game. In other environments, the rewards come more frequently. In ping-pong, each point scored can be considered a reward; when learning to crawl, any forward motion is an achievement. Our framework for agents regards the reward as *part* of the input percept, but the agent must be "hardwired" to recognize that part as a reward rather than as just another sensory input. Thus, animals seem to be hardwired to recognize pain and hunger as negative rewards and pleasure and food intake as positive rewards. Reinforcement has been carefully studied by animal psychologists for over 60 years.

Rewards were introduced in Chapter 17, where they served to define optimal policies in **Markov decision processes** (MDPs). An optimal policy is a policy that maximizes the expected total reward. The task of **reinforcement learning** is to use observed rewards to learn an optimal (or nearly optimal) policy for the environment. Whereas in Chapter 17 the agent

has a complete model of the environment and knows the reward function, here we assume no prior knowledge of either. Imagine playing a new game whose rules you don't know; after a hundred or so moves, your opponent announces, "You lose." This is reinforcement learning in a nutshell.

In many complex domains, reinforcement learning is the only feasible way to train a program to perform at high levels. For example, in game playing, it is very hard for a human to provide accurate and consistent evaluations of large numbers of positions, which would be needed to train an evaluation function directly from examples. Instead, the program can be told when it has won or lost, and it can use this information to learn an evaluation function that gives reasonably accurate estimates of the probability of winning from any given position. Similarly, it is extremely difficult to program an agent to fly a helicopter; yet given appropriate negative rewards for crashing, wobbling, or deviating from a set course, an agent can learn to fly by itself.

Reinforcement learning might be considered to encompass all of AI: an agent is placed in an environment and must learn to behave successfully therein. To keep the chapter manageable, we will concentrate on simple settings and simple agent designs. For the most part, we will assume a fully observable environment, so that the current state is supplied by each percept. On the other hand, we will assume that the agent does not know how the environment works or what its actions do, and we will allow for probabilistic action outcomes. We will consider three of the agent designs first introduced in Chapter 2:

- A **utility-based agent** learns a utility function on states and uses it to select actions that maximize the expected outcome utility.
- Q-LEARNING ACTION-VALUE A **Q-learning** agent learns an **action-value** function, or Q-function, giving the expected utility of taking a given action in a given state.
- A **reflex agent** learns a policy that maps directly from states to actions.

A utility-based agent must also have a model of the environment in order to make decisions, because it must know the states to which its actions will lead. For example, in order to make use of a backgammon evaluation function, a backgammon program must know what its legal moves are *and how they affect the board position*. Only in this way can it apply the utility function to the outcome states. A Q-learning agent, on the other hand, can compare the values of its available choices without needing to know their outcomes, so it does not need a model of the environment. On the other hand, because they do not know where their actions lead, Q-learning agents cannot look ahead; this can seriously restrict their ability to learn, as we shall see.

PASSIVE LEARNING We begin in Section 21.2 with **passive learning**, where the agent's policy is fixed and the task is to learn the utilities of states (or state–action pairs); this could also involve learning a model of the environment. Section 21.3 covers **active learning**, where the agent must also learn what to do. The principal issue is **exploration**: an agent must experience as much as possible of its environment in order to learn how to behave in it. Section 21.4 discusses how an agent can use inductive learning to learn much faster from its experiences. Section 21.5 covers methods for learning direct policy representations in reflex agents. An understanding of Markov decision processes (Chapter 17) is essential for this chapter.

ACTIVE LEARNING

EXPLORATION

21.2 PASSIVE REINFORCEMENT LEARNING

To keep things simple, we start with the case of a passive learning agent using a state-based representation in a fully observable environment. In passive learning, the agent's policy π is fixed: in state s, it always executes the action $\pi(s)$. Its goal is simply to learn how good the policy is—that is, to learn the utility function $U^\pi(s)$. We will use as our example the 4×3 world introduced in Chapter 17. Figure 21.1 shows a policy for that world and the corresponding utilities. Clearly, the passive learning task is similar to the **policy evaluation** task, part of the **policy iteration** algorithm described in Section 17.3. The main difference is that the passive learning agent does not know the **transition model** $T(s, a, s')$, which specifies the probability of reaching state s' from state s after doing action a; nor does it know the **reward function** $R(s)$, which specifies the reward for each state.

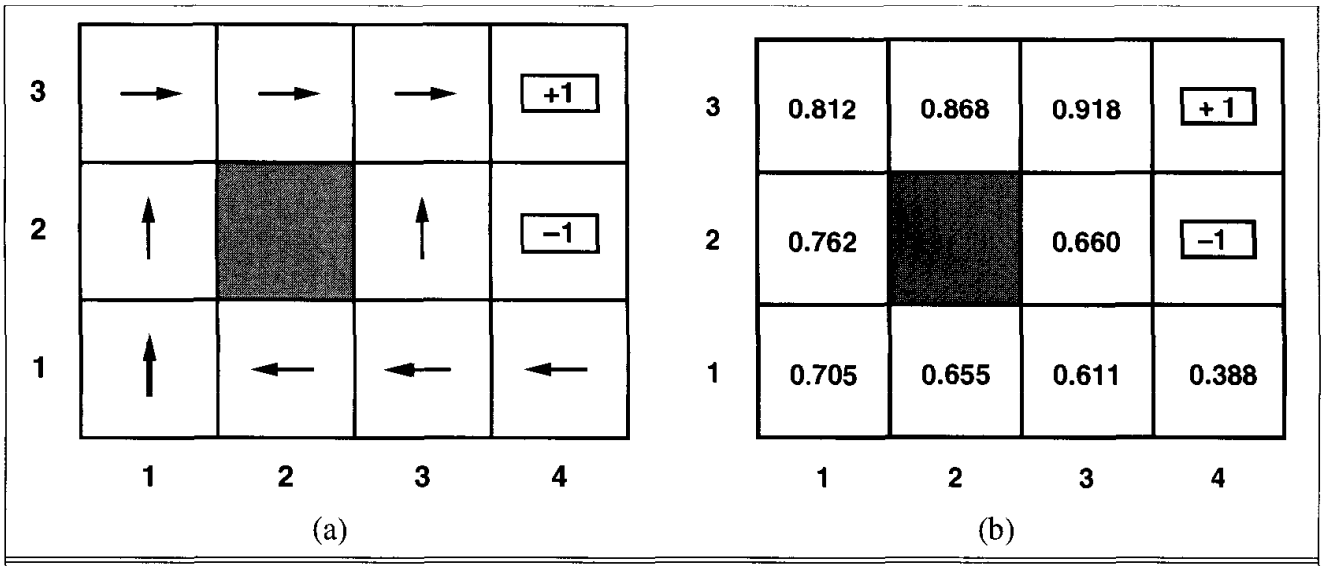


Figure 21.1 (a) A policy π for the 4×3 world; this policy happens to be optimal with rewards of $R(s) = -0.04$ in the nonterminal states and no discounting. (b) The utilities of the states in the 4×3 world, given policy π.

TRIAL

The agent executes a set of **trials** in the environment using its policy π. In each trial, the agent starts in state (1,1) and experiences a sequence of state transitions until it reaches one of the terminal states, (4,2) or (4,3). Its percepts supply both the current state and the reward received in that state. Typical trials might look like this:

$$(1,1)_{-.04} \rightarrow (1,2)_{-.04} \rightarrow (1,3)_{-.04} \rightarrow (1,2)_{-.04} \rightarrow (1,3)_{-.04} \rightarrow (2,3)_{-.04} \rightarrow (3,3)_{-.04} \rightarrow (4,3)_{+1}$$
$$(1,1)_{-.04} \rightarrow (1,2)_{-.04} \rightarrow (1,3)_{-.04} \rightarrow (2,3)_{-.04} \rightarrow (3,3)_{-.04} \rightarrow (3,2)_{-.04} \rightarrow (3,3)_{-.04} \rightarrow (4,3)_{+1}$$
$$(1,1)_{-.04} \rightarrow (2,1)_{-.04} \rightarrow (3,1)_{-.04} \rightarrow (3,2)_{-.04} \rightarrow (4,2)_{-1} \,.$$

Note that each state percept is subscripted with the reward received. The object is to use the information about rewards to learn the expected utility $U^\pi(s)$ associated with each nonterminal state s. The utility is defined to be the expected sum of (discounted) rewards obtained if

policy π is followed. As in Equation (17.3) on page 619, this is written as

$$U^{\pi}(s) = E\left[\sum_{t=0}^{\infty} \gamma^t R(s_t) \mid \pi, s_0 = s\right] . \tag{21.1}$$

We will include a **discount factor** γ in all of our equations, but for the 4×3 world we will set $\gamma = 1$.

Direct utility estimation

DIRECT UTILITY ESTIMATION
ADAPTIVE CONTROL THEORY

A simple method for **direct utility estimation** was invented in the late 1950s in the area of **adaptive control theory** by Widrow and Hoff (1960). The idea is that the utility of a state is the expected total reward from that state onward, and each trial provides a *sample* of this value for each state visited. For example, the first trial in the set of three given earlier provides a sample total reward of 0.72 for state (1,1), two samples of 0.76 and 0.84 for (1,2), two samples of 0.80 and 0.88 for (1,3), and so on. Thus, at the end of each sequence, the algorithm calculates the observed reward-to-go for each state and updates the estimated utility for that state accordingly, just by keeping a running average for each state in a table. In the limit of infinitely many trials, the sample average will converge to the true expectation in Equation (21.1).

It is clear that direct utility estimation is just an instance of supervised learning where each example has the state as input and the observed reward-to-go as output. This means that we have reduced reinforcement learning to a standard inductive learning problem, as discussed in Chapter 18. Section 21.4 discusses the use of more powerful kinds of representations for the utility function, such as neural networks. Learning techniques for those representations can be applied directly to the observed data.

Direct utility estimation succeeds in reducing the reinforcement learning problem to an inductive learning problem, about which much is known. Unfortunately, it misses a very important source of information, namely, the fact that the utilities of states are not independent! *The utility of each state equals its own reward plus the expected utility of its successor states.* That is, the utility values obey the Bellman equations for a fixed policy (see also Equation (17.10)):

$$U^{\pi}(s) = R(s) + \gamma \sum_{s'} T(s, \pi(s), s') U^{\pi}(s') . \tag{21.2}$$

By ignoring the connections between states, direct utility estimation misses opportunities for learning. For example, the second of the three trials given earlier reaches the state (3,2), which has not previously been visited. The next transition reaches (3,3), which is known from the first trial to have a high utility. The Bellman equation suggests immediately that (3,2) is also likely to have a high utility, because it leads to (3,3), but direct utility estimation learns nothing until the end of the trial. More broadly, we can view direct utility estimation as searching in a hypothesis space for U that is much larger than it needs to be, in that it includes many functions that violate the Bellman equations. For this reason, the algorithm often converges very slowly.

Adaptive dynamic programming

ADAPTIVE DYNAMIC PROGRAMMING

In order to take advantage of the constraints between states, an agent must learn how states are connected. An **adaptive dynamic programming** (or **ADP**) agent works by learning the transition model of the environment as it goes along and solving the corresponding Markov decision process using a dynamic programming method. For a passive learning agent, this means plugging the learned transition model $T(s, \pi(s), s')$ and the observed rewards $R(s)$ into the Bellman equations (21.2) to calculate the utilities of the states. As we remarked in our discussion of policy iteration in Chapter 17, these equations are linear (no maximization involved) so they can be solved using any linear algebra package. Alternatively, we can adopt the approach of **modified policy iteration** (see page 625), using a simplified value iteration process to update the utility estimates after each change to the learned model. Because the model usually changes only slightly with each observation, the value iteration process can use the previous utility estimates as initial values and should converge quite quickly.

The process of learning the model itself is easy, because the environment is fully observable. This means that we have a supervised learning task where the input is a state–action pair and the output is the resulting state. In the simplest case, we can represent the transition model as a table of probabilities. We keep track of how often each action outcome occurs and estimate the transition probability $T(s, a, s')$ from the frequency with which s' is reached when executing a in s.[1] For example, in the three traces given on page 765, *Right* is executed three times in (1,3) and two out of three times the resulting state is (2,3), so $T((1,3), \mathit{Right}, (2,3))$ is estimated to be 2/3.

The full agent program for a passive ADP agent is shown in Figure 21.2. Its performance on the 4×3 world is shown in Figure 21.3. In terms of how quickly its value estimates improve, the ADP agent does as well as possible, subject to its ability to learn the transition model. In this sense, it provides a standard against which to measure other reinforcement learning algorithms. It is, however, somewhat intractable for large state spaces. In backgammon, for example, it would involve solving roughly 10^{50} equations in 10^{50} unknowns.

Temporal difference learning

It is possible to have (almost) the best of both worlds; that is, one can approximate the constraint equations shown earlier without solving them for all possible states. *The key is to use the observed transitions to adjust the values of the observed states so that they agree with the constraint equations.* Consider, for example, the transition from (1,3) to (2,3) in the second trial on page 765. Suppose that, as a result of the first trial, the utility estimates are $U^\pi(1,3) = 0.84$ and $U^\pi(2,3) = 0.92$. Now, if this transition occurred all the time, we would expect the utilities to obey

$$U^\pi(1,3) = -0.04 + U^\pi(2,3) \ ,$$

so $U^\pi(1,3)$ would be 0.88. Thus, its current estimate of 0.84 might be a little low and should be increased. More generally, when a transition occurs from state s to state s', we apply the

[1] This is the maximum likelihood estimate, as discussed in Chapter 20. A Bayesian update with a Dirichlet prior might work better.

```
function PASSIVE-ADP-AGENT(percept) returns an action
  inputs: percept, a percept indicating the current state s' and reward signal r'
  static: π, a fixed policy
          mdp, an MDP with model T, rewards R, discount γ
          U, a table of utilities, initially empty
          N_sa, a table of frequencies for state-action pairs, initially zero
          N_sas', a table of frequencies for state-action-state triples, initially zero
          s, a, the previous state and action, initially null

  if s' is new then do U[s'] ← r'; R[s'] ← r'
  if s is not null then do
     increment N_sa[s, a] and N_sas'[s, a, s']
     for each t such that N_sas'[s, a, t] is nonzero do
        T[s, a, t] ← N_sas'[s, a, t] / N_sa[s, a]
  U ← VALUE-DETERMINATION(π, U, mdp)
  if TERMINAL?[s'] then s, a ← null else s, a ← s', π[s']
  return a
```

Figure 21.2 A passive reinforcement learning agent based on adaptive dynamic programming. To simplify the code, we have assumed that each percept can be divided into a perceived state and a reward signal.

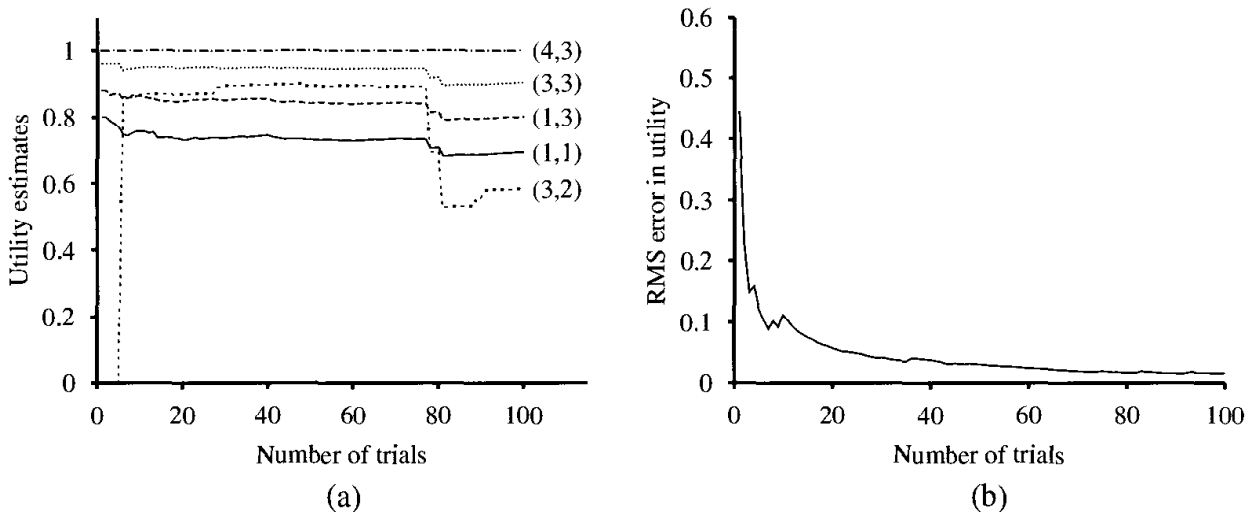


Figure 21.3 The passive ADP learning curves for the 4×3 world, given the optimal policy shown in Figure 21.1. (a) The utility estimates for a selected subset of states, as a function of the number of trials. Notice the large changes occurring around the 78th trial—this is the first time that the agent falls into the -1 terminal state at (4,2). (b) The root-mean-square error in the estimate for $U(1,1)$, averaged over 20 runs of 100 trials each.

```
function PASSIVE-TD-AGENT(percept) returns an action
  inputs: percept, a percept indicating the current state s' and reward signal r'
  static: π, a fixed policy
          U, a table of utilities, initially empty
          N_s, a table of frequencies for states, initially zero
          s, a, r, the previous state, action, and reward, initially null

  if s' is new then U[s'] ← r'
  if s is not null then do
     increment N_s[s]
     U[s] ← U[s] + α(N_s[s])(r + γ U[s'] − U[s])
  if TERMINAL?[s'] then s, a, r ← null else s, a, r ← s', π[s'], r'
  return a
```

Figure 21.4 A passive reinforcement learning agent that learns utility estimates using temporal differences.

following update to $U^\pi(s)$:

$$U^\pi(s) \leftarrow U^\pi(s) + \alpha(R(s) + \gamma U^\pi(s') - U^\pi(s)) \ . \tag{21.3}$$

Here, α is the **learning rate** parameter. Because this update rule uses the difference in utilities between successive states, it is often called the **temporal-difference**, or **TD**, equation.

TEMPORAL-DIFFERENCE

The basic idea of all temporal-difference methods is, first to define the conditions that hold locally when the utility estimates are correct, and then, to write an update equation that moves the estimates toward this ideal "equilibrium" equation. In the case of passive learning, the equilibrium is given by Equation (21.2). Now Equation (21.3) does in fact cause the agent to reach the equilibrium given by Equation (21.2), but there is some subtlety involved. First, notice that the update involves only the observed successor s', whereas the actual equilibrium conditions involve all possible next states. One might think that this causes an improperly large change in $U^\pi(s)$ when a very rare transition occurs; but, in fact, because rare transitions occur only rarely, the *average value* of $U^\pi(s)$ will converge to the correct value. Furthermore, if we change α from a fixed parameter to a function that decreases as the number of times a state has been visited increases, then $U(s)$ itself will converge to the correct value.[2] This gives us the agent program shown in Figure 21.4. Figure 21.5 illustrates the performance of the passive TD agent on the 4×3 world. It does not learn quite as fast as the ADP agent and shows much higher variability, but it is much simpler and requires much less computation per observation. Notice that *TD does not need a model to perform its updates.* The environment supplies the connection between neighboring states in the form of observed transitions.

The ADP approach and the TD approach are actually closely related. Both try to make local adjustments to the utility estimates in order to make each state "agree" with its successors. One difference is that TD adjusts a state to agree with its *observed* successor (Equa-

[2] Technically, we require that $\sum_{n=1}^{\infty} \alpha(n) = \infty$ and $\sum_{n=1}^{\infty} \alpha^2(n) < \infty$. The decay $\alpha(n) = 1/n$ satisfies these conditions. In Figure 21.5 we have used $\alpha(n) = 60/(59 + n)$.

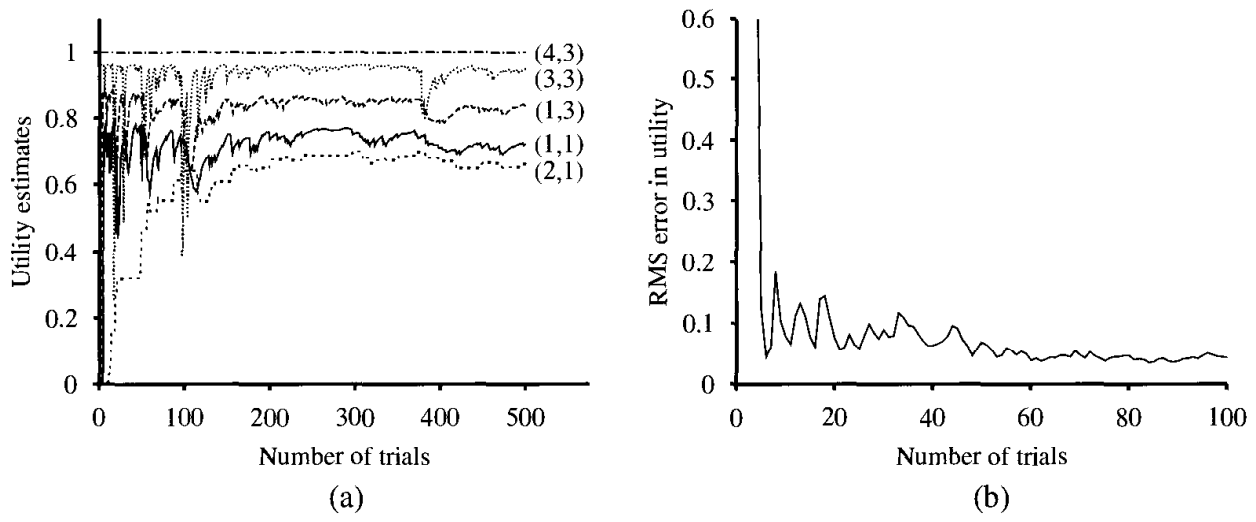


Figure 21.5 The TD learning curves for the 4×3 world. (a) The utility estimates for a selected subset of states, as a function of the number of trials. (b) The root-mean-square error in the estimate for $U(1,1)$, averaged over 20 runs of 500 trials each. Only the first 100 trials are shown to enable comparison with Figure 21.3.

tion (21.3)), whereas ADP adjusts the state to agree with *all* of the successors that might occur, weighted by their probabilities (Equation (21.2)). This difference disappears when the effects of TD adjustments are averaged over a large number of transitions, because the frequency of each successor in the set of transitions is approximately proportional to its probability. A more important difference is that whereas TD makes a single adjustment per observed transition, ADP makes as many as it needs to restore consistency between the utility estimates U and the environment model T. Although the observed transition makes only a local change in T, its effects might need to be propagated throughout U. Thus, TD can be viewed as a crude but efficient first approximation to ADP.

Each adjustment made by ADP could be seen, from the TD point of view, as a result of a "pseudo-experience" generated by simulating the current environment model. It is possible to extend the TD approach to use an environment model to generate several pseudo-experiences—transitions that the TD agent can imagine *might* happen, given its current model. For each observed transition, the TD agent can generate a large number of imaginary transitions. In this way, the resulting utility estimates will approximate more and more closely those of ADP—of course, at the expense of increased computation time.

In a similar vein, we can generate more efficient versions of ADP by directly approximating the algorithms for value iteration or policy iteration. Recall that full value iteration can be intractable when the number of states is large. Many of the adjustment steps, however, are extremely tiny. One possible approach to generating reasonably good answers quickly is to bound the number of adjustments made after each observed transition. One can also use a heuristic to rank the possible adjustments so as to carry out only the most significant ones.

PRIORITIZED SWEEPING

The **prioritized sweeping** heuristic prefers to make adjustments to states whose *likely* successors have just undergone a *large* adjustment in their own utility estimates. Using heuristics like this, approximate ADP algorithms usually can learn roughly as fast as full ADP, in terms

of the number of training sequences, but can be several orders of magnitude more efficient in terms of computation. (See Exercise 21.3.) This enables them to handle state spaces that are far too large for full ADP. Approximate ADP algorithms have an additional advantage: in the early stages of learning a new environment, the environment model T often will be far from correct, so there is little point in calculating an exact utility function to match it. An approximation algorithm can use a minimum adjustment size that decreases as the environment model becomes more accurate. This eliminates the very long value iterations that can occur early in learning due to large changes in the model.

21.3 Active Reinforcement Learning

A passive learning agent has a fixed policy that determines its behavior. An active agent must decide what actions to take. Let us begin with the adaptive dynamic programming agent and consider how it must be modified to handle this new freedom.

First, the agent will need to learn a complete model with outcome probabilities for all actions, rather than just the model for the fixed policy. The simple learning mechanism used by PASSIVE-ADP-AGENT will do just fine for this. Next, we need to take into account the fact that the agent has a choice of actions. The utilities it needs to learn are those defined by the *optimal* policy; they obey the Bellman equations given on page 619, which we repeat here:

$$U(s) = R(s) + \gamma \max_a \sum_{s'} T(s,a,s')U(s') \,. \tag{21.4}$$

These equations can be solved to obtain the utility function U using the value iteration or policy iteration algorithms from Chapter 17. The final issue is what to do at each step. Having obtained a utility function U that is optimal for the learned model, the agent can extract an optimal action by one-step look-ahead to maximize the expected utility; alternatively, if it uses policy iteration, the optimal policy is already available, so it should simply execute the action the optimal policy recommends. Or should it?

Exploration

Figure 21.6 shows the results of one sequence of trials for an ADP agent that follows the recommendation of the optimal policy for the learned model at each step. The agent *does not* learn the true utilities or the true optimal policy! What happens instead is that, in the 39th trial, it finds a policy that reaches the +1 reward along the lower route via (2,1), (3,1), (3,2), and (3,3). (See Figure 21.6.) After experimenting with minor variations, from the 276th trial onward it sticks to that policy, never learning the utilities of the other states and never finding the optimal route via (1,2), (1,3), and (2,3). We call this agent the **greedy agent**. Repeated experiments show that the greedy agent *very seldom* converges to the optimal policy for this environment and sometimes converges to really horrendous policies.

GREEDY AGENT

How can it be that choosing the optimal action leads to suboptimal results? The answer is that the learned model is not the same as the true environment; what is optimal in the

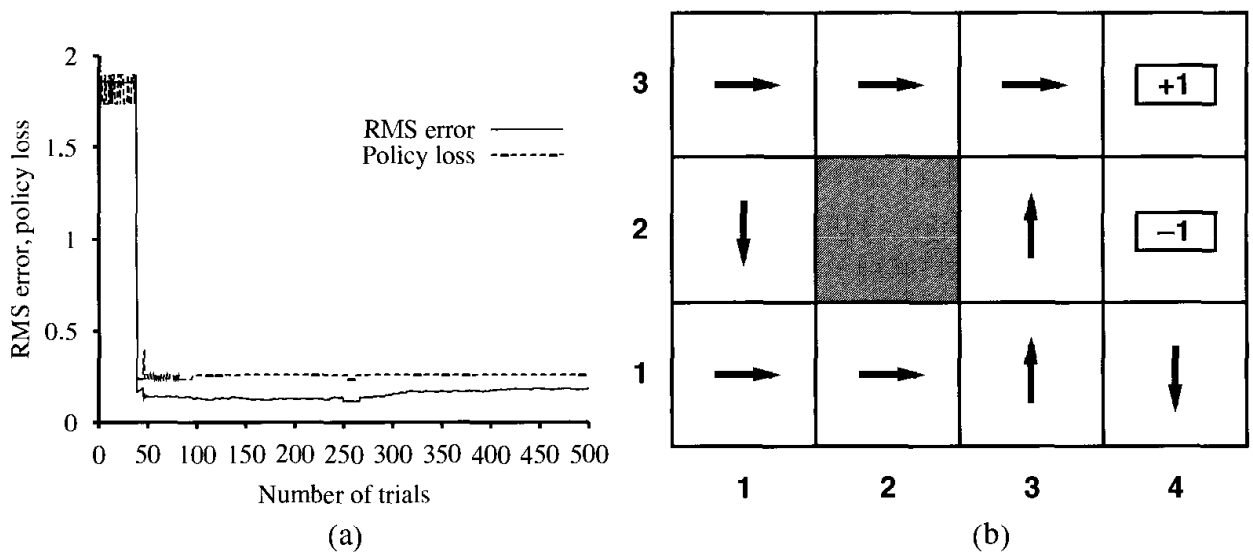


Figure 21.6 Performance of a greedy ADP agent that executes the action recommended by the optimal policy for the learned model. (a) RMS error in the utility estimates averaged over the nine nonterminal squares. (b) The suboptimal policy to which the greedy agent converges in this particular sequence of trials.

learned model can therefore be suboptimal in the true environment. Unfortunately, the agent does not know what the true environment is, so it cannot compute the optimal action for the true environment. What, then, is to be done?

What the greedy agent has overlooked is that actions do more than provide rewards according to the current learned model; they also contribute to learning the true model by affecting the percepts that are received. By improving the model, the agent will receive greater rewards in the future.[3] An agent therefore must make a trade-off between **exploitation** to maximize its reward—as reflected in its current utility estimates—and **exploration** to maximize its long-term well-being. Pure exploitation risks getting stuck in a rut. Pure exploration to improve one's knowledge is of no use if one never puts that knowledge into practice. In the real world, one constantly has to decide between continuing in a comfortable existence and striking out into the unknown in the hopes of discovering a new and better life. With greater understanding, less exploration is necessary.

EXPLOITATION

EXPLORATION

Can we be a little more precise than this? Is there an *optimal* exploration policy? It turns out that this question has been studied in depth in the subfield of statistical decision theory that deals with so-called **bandit problems**. (See sidebar.)

BANDIT PROBLEMS

Although bandit problems are extremely difficult to solve exactly to obtain an *optimal* exploration method, it is nonetheless possible to come up with a *reasonable* scheme that will eventually lead to optimal behavior by the agent. Technically, any such scheme needs to be greedy in the limit of infinite exploration, or **GLIE**. A GLIE scheme must try each action in each state an unbounded number of times to avoid having a finite probability that an optimal action is missed because of an unusually bad series of outcomes. An ADP agent

GLIE

[3] Notice the direct analogy to the theory of information value in Chapter 16.

EXPLORATION AND BANDITS

In Las Vegas, a *one-armed bandit* is a slot machine. A gambler can insert a coin, pull the lever, and collect the winnings (if any). An n-**armed bandit** has n levers. The gambler must choose which lever to play on each successive coin—the one that has paid off best, or maybe one that has not been tried?

The n-armed bandit problem is a formal model for real problems in many vitally important areas, such as deciding on the annual budget for AI research and development. Each arm corresponds to an action (such as allocating $20 million for the development of new AI textbooks), and the payoff from pulling the arm corresponds to the benefits obtained from taking the action (immense). Exploration, whether it is exploration of a new research field or exploration of a new shopping mall, is risky, is expensive, and has uncertain payoffs; on the other hand, failure to explore at all means that one never discovers *any* actions that are worthwhile.

To formulate a bandit problem properly, one must define exactly what is meant by optimal behavior. Most definitions in the literature assume that the aim is to maximize the expected total reward obtained over the agent's lifetime. These definitions require that the expectation be taken over the possible worlds that the agent could be in, as well as over the possible results of each action sequence in any given world. Here, a "world" is defined by the transition model $T(s, a, s')$. Thus, in order to act optimally, the agent needs a prior distribution over the possible models. The resulting optimization problems are usually wildly intractable.

In some cases—for example, when the payoff of each machine is independent and discounted rewards are used—it is possible to calculate a **Gittins index** for each slot machine (Gittins, 1989). The index is a function only of the number of times the slot machine has been played and how much it has paid off. The index for each machine indicates how worthwhile it is to invest more, based on a combination of expected return and expected value of information. Choosing the machine with the highest index value gives an optimal exploration policy. Unfortunately, no way has been found to extend Gittins indices to sequential decision problems.

One can use the theory of n-armed bandits to argue for the reasonableness of the selection strategy in genetic algorithms. (See Chapter 4.) If you consider each arm in an n-armed bandit problem to be a possible string of genes, and the investment of a coin in one arm to be the reproduction of those genes, then genetic algorithms allocate coins optimally, given an appropriate set of independence assumptions.

using such a scheme will eventually learn the true environment model. A GLIE scheme must also eventually become greedy, so that the agent's actions become optimal with respect to the learned (and hence the true) model.

There are several GLIE schemes; one of the simplest is to have the agent choose a random action a fraction $1/t$ of the time and to follow the greedy policy otherwise. While this does eventually converge to an optimal policy, it can be extremely slow. A more sensible approach would give some weight to actions that the agent has not tried very often, while tending to avoid actions that are believed to be of low utility. This can be implemented by altering the constraint equation (21.4) so that it assigns a higher utility estimate to relatively unexplored state–action pairs. Essentially, this amounts to an optimistic prior over the possible environments and causes the agent to behave initially as if there were wonderful rewards scattered all over the place. Let us use $U^+(s)$ to denote the optimistic estimate of the utility (i.e., the expected reward-to-go) of the state s, and let $N(a, s)$ be the number of times action a has been tried in state s. Suppose we are using value iteration in an ADP learning agent; then we need to rewrite the update equation (i.e., Equation (17.6)) to incorporate the optimistic estimate. The following equation does this:

$$U^+(s) \leftarrow R(s) + \gamma \max_a f\left(\sum_{s'} T(s,a,s')U^+(s'),\ N(a,s)\right) . \tag{21.5}$$

EXPLORATION FUNCTION

Here, $f(u, n)$ is called the **exploration function**. It determines how greed (preference for high values of u) is traded off against curiosity (preference for low values of n—actions that have not been tried often). The function $f(u, n)$ should be increasing in u and decreasing in n. Obviously, there are many possible functions that fit these conditions. One particularly simple definition is

$$f(u,n) = \begin{cases} R^+ & \text{if } n < N_e \\ u & \text{otherwise} \end{cases}$$

where R^+ is an optimistic estimate of the best possible reward obtainable in any state and N_e is a fixed parameter. This will have the effect of making the agent try each action–state pair at least N_e times.

The fact that U^+ rather than U appears on the right-hand side of Equation (21.5) is very important. As exploration proceeds, the states and actions near the start state might well be tried a large number of times. If we used U, the more pessimistic utility estimate, then the agent would soon become disinclined to explore further afield. The use of U^+ means that the benefits of exploration are propagated back from the edges of unexplored regions, so that actions that lead *toward* unexplored regions are weighted more highly, rather than just actions that are themselves unfamiliar. The effect of this exploration policy can be seen clearly in Figure 21.7, which shows a rapid convergence toward optimal performance, unlike that of the greedy approach. A very nearly optimal policy is found after just 18 trials. Notice that the utility estimates themselves do not converge as quickly. This is because the agent stops exploring the unrewarding parts of the state space fairly soon, visiting them only "by accident" thereafter. However, it makes perfect sense for the agent not to care about the exact utilities of states that it knows are undesirable and can be avoided.

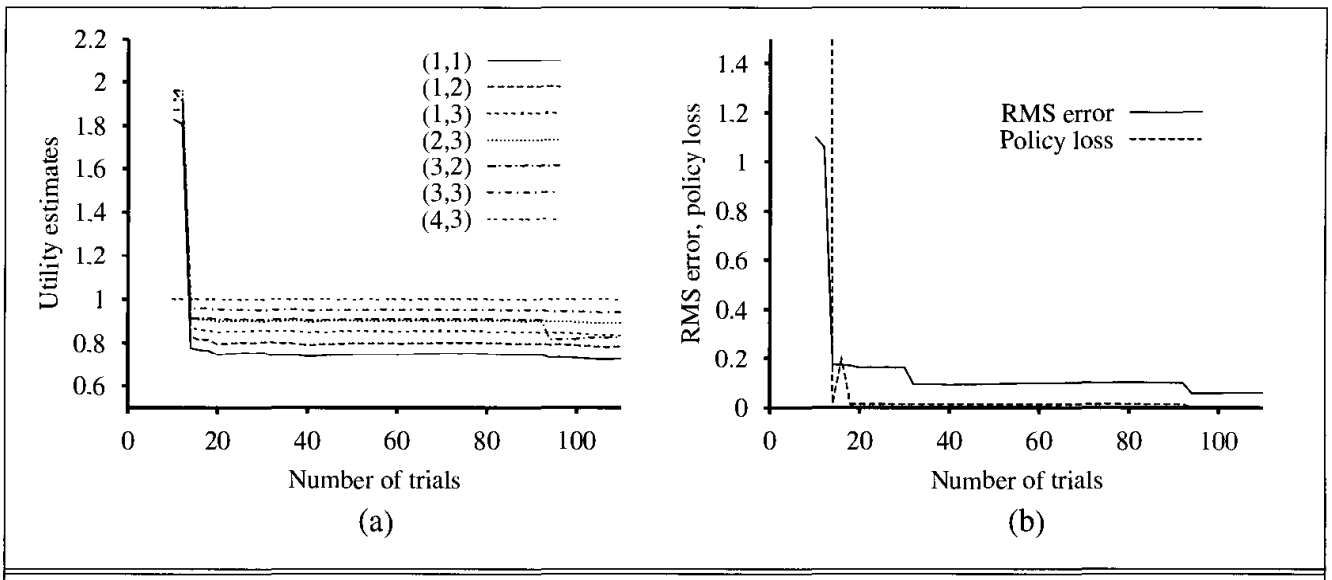


Figure 21.7 Performance of the exploratory ADP agent. using $R^+ = 2$ and $N_e = 5$. (a) Utility estimates for selected states over time. (b) The RMS error in utility values and the associated policy loss.

Learning an Action-Value Function

Now that we have an active ADP agent, let us consider how to construct an active temporal-difference learning agent. The most obvious change from the passive case is that the agent is no longer equipped with a fixed policy, so, if it learns a utility function U, it will need to learn a model in order to be able to choose an action based on U via one-step look-ahead. The model acquisition problem for the TD agent is identical to that for the ADP agent. What of the TD update rule itself? Perhaps surprisingly, the update rule (21.3) remains unchanged. This might seem odd, for the following reason: Suppose the agent takes a step that normally leads to a good destination, but because of nondeterminism in the environment the agent ends up in a catastrophic state. The TD update rule will take this as seriously as if the outcome had been the normal result of the action, whereas one might suppose that, because the outcome was a fluke, the agent should not worry about it too much. In fact, of course, the unlikely outcome will occur only infrequently in a large set of training sequences; hence in the long run its effects will be weighted proportionally to its probability, as we would hope. Once again, it can be shown that the TD algorithm will converge to the same values as ADP as the number of training sequences tends to infinity.

There is an alternative TD method called **Q-learning** that learns an action-value representation instead of learning utilities. We will use the notation $Q(a, s)$ to denote the value of doing action a in state s. Q-values are directly related to utility values as follows:

$$U(s) = \max_a Q(a, s) \,. \tag{21.6}$$

MODEL-FREE

Q-functions may seem like just another way of storing utility information, but they have a very important property: *a TD agent that learns a Q-function does not need a model for either learning or action selection.* For this reason, Q-learning is called a **model-free** method. As with utilities, we can write a constraint equation that must hold at equilibrium when the

```
function Q-LEARNING-AGENT(percept) returns an action
  inputs: percept, a percept indicating the current state s' and reward signal r'
  static: Q, a table of action values index by state and action
          N_sa, a table of frequencies for state-action pairs
          s, a, r, the previous state, action, and reward, initially null

  if s is not null then do
     increment N_sa[s, a]
     Q[a, s] ← Q[a, s] + α(N_sa[s, a])(r + γ max_a' Q[a', s'] − Q[a, s])
  if TERMINAL?[s'] then s, a, r ← null
  else s, a, r ← s', argmax_a' f(Q[a', s'], N_sa[a', s']), r'
  return a
```

Figure 21.8 An exploratory Q-learning agent. It is an active learner that learns the value $Q(a, s)$ of each action in each situation. It uses the same exploration function f as the exploratory ADP agent, but avoids having to learn the transition model because the Q-value of a state can be related directly to those of its neighbors.

Q-values are correct:

$$Q(a, s) = R(s) + \gamma \sum_{s'} T(s, a, s') \max_{a'} Q(a', s') \ . \tag{21.7}$$

As in the ADP learning agent, we can use this equation directly as an update equation for an iteration process that calculates exact Q-values, given an estimated model. This does, however, require that a model also be learned because the equation uses $T(s, a, s')$. The temporal-difference approach, on the other hand, requires no model. The update equation for TD Q-learning is

$$Q(a, s) \leftarrow Q(a, s) + \alpha(R(s) + \gamma \max_{a'} Q(a', s') - Q(a, s)) \ , \tag{21.8}$$

which is calculated whenever action a is executed in state s leading to state s'.

The complete agent design for an exploratory Q-learning agent using TD is shown in Figure 21.8. Notice that it uses exactly the same exploration function f as that used by the exploratory ADP agent—hence the need to keep statistics on actions taken (the table N). If a simpler exploration policy is used—say, acting randomly on some fraction of steps, where the fraction decreases over time—then we can dispense with the statistics.

The Q-learning agent learns the optimal policy for the 4×3 world, but does so at a much slower rate than the ADP agent. This is because TD does not enforce consistency among values via the model. The comparison raises a general question: is it better to learn a model and a utility function or to learn an action-value function with no model? In other words, what is the best way to represent the agent function? This is an issue at the foundations of artificial intelligence. As we stated in Chapter 1, one of the key historical characteristics of much of AI research is its (often unstated) adherence to the **knowledge-based** approach. This amounts to an assumption that the best way to represent the agent function is to build a representation of some aspects of the environment in which the agent is situated.

Some researchers, both inside and outside AI, have claimed that the availability of model-free methods such as Q-learning means that the knowledge-based approach is unnecessary. There is, however, little to go on but intuition. Our intuition, for what it's worth, is that as the environment becomes more complex, the advantages of a knowledge-based approach become more apparent. This is borne out even in games such as chess, checkers (draughts), and backgammon (see next section), where efforts to learn an evaluation function by means of a model have met with more success than Q-learning methods.

21.4 Generalization in Reinforcement Learning

So far, we have assumed that the utility functions and Q-functions learned by the agents are represented in tabular form with one output value for each input tuple. Such an approach works reasonably well for small state spaces, but the time to convergence and (for ADP) the time per iteration increase rapidly as the space gets larger. With carefully controlled, approximate ADP methods, it might be possible to handle 10,000 states or more. This suffices for two-dimensional maze-like environments, but more realistic worlds are out of the question. Chess and backgammon are tiny subsets of the real world, yet their state spaces contain on the order of 10^{50} to 10^{120} states. It would be absurd to suppose that one must visit all these states in order to learn how to play the game!

FUNCTION APPROXIMATION

One way to handle such problems is to use **function approximation**, which simply means using any sort of representation for the function other than a table. The representation is viewed as approximate because it might not be the case that the *true* utility function or Q-function can be represented in the chosen form. For example, in Chapter 6 we described an **evaluation function** for chess that is represented as a weighted linear function of a set of **features** (or **basis functions**) $f_1, \ldots, f_n$:

BASIS FUNCTIONS

$$\hat{U}_\theta(s) = \theta_1 f_1(s) + \theta_2 f_2(s) + \cdots + \theta_n f_n(s) .$$

A reinforcement learning algorithm can learn values for the parameters $\theta = \theta_1, \ldots, \theta_n$ such that the evaluation function $\hat{U}_\theta$ approximates the true utility function. Instead of, say, 10^{120} values in a table, this function approximator is characterized by, say, $n = 20$ parameters—an *enormous* compression. Although no one knows the true utility function for chess, no one believes that it can be represented exactly in 20 numbers. If the approximation is good enough, however, the agent might still play excellent chess.[4]

Function approximation makes it practical to represent utility functions for very large state spaces, but that is not its principal benefit. *The compression achieved by a function approximator allows the learning agent to generalize from states it has visited to states it has not visited.* That is, the most important aspect of function approximation is not that it

[4] We do know that the exact utility function can be represented in a page or two of Lisp, Java, or C++. That is, it can be represented by a program that solves the game exactly every time it is called. We are interested only in function approximators that use a *reasonable* amount of computation. It might in fact be better to learn a very simple function approximator and combine it with a certain amount of look-ahead search. The trade-offs involved are currently not well understood.

requires less space, but that it allows for inductive generalization over input states. To give you some idea of the power of this effect: by examining only one in every 10^{44} of the possible backgammon states, it is possible to learn a utility function that allows a program to play as well as any human (Tesauro, 1992).

On the flip side, of course, there is the problem that there could fail to be any function in the chosen hypothesis space that approximates the true utility function sufficiently well. As in all inductive learning, there is a trade-off between the size of the hypothesis space and the time it takes to learn the function. A larger hypothesis space increases the likelihood that a good approximation can be found, but also means that convergence is likely to be delayed.

Let us begin with the simplest case, which is direct utility estimation. (See Section 21.2.) With function approximation, this is an instance of **supervised learning**. For example, suppose we represent the utilities for the 4×3 world using a simple linear function. The features of the squares are just their x and y coordinates, so we have

$$\hat{U}_\theta(x, y) = \theta_0 + \theta_1 x + \theta_2 y \ . \tag{21.9}$$

Thus, if $(\theta_0, \theta_1, \theta_2) = (0.5, 0.2, 0.1)$, then $\hat{U}_\theta(1,1) = 0.8$. Given a collection of trials, we obtain a set of sample values of $\hat{U}_\theta(x, y)$, and we can find the best fit, in the sense of minimizing the squared error, using standard linear regression. (See Chapter 20.)

For reinforcement learning, it makes more sense to use an *online* learning algorithm that updates the parameters after each trial. Suppose we run a trial and the total reward obtained starting at (1,1) is 0.4. This suggests that $\hat{U}_\theta(1,1)$, currently 0.8, is too large and must be reduced. How should the parameters be adjusted to achieve this? As with neural network learning, we write an error function and compute its gradient with respect to the parameters. If $u_j(s)$ is the observed total reward from state s onward in the jth trial, then the error is defined as (half) the squared difference of the predicted total and the actual total: $E_j(s) = (\hat{U}_\theta(s) - u_j(s))^2/2$. The rate of change of the error with respect to each parameter θ_i is $\partial E_j / \partial \theta_i$, so to move the parameter in the direction of decreasing the error, we want

$$\theta_i \leftarrow \theta_i - \alpha \frac{\partial E_j(s)}{\partial \theta_i} = \theta_i + \alpha \left(u_j(s) - \hat{U}_\theta(s)\right) \frac{\partial \hat{U}_\theta(s)}{\partial \theta_i} \ . \tag{21.10}$$

WIDROW–HOFF RULE
DELTA RULE

This is called the **Widrow–Hoff rule**, or the **delta rule**, for online least-squares. For the linear function approximator $\hat{U}_\theta(s)$ in Equation (21.9), we get three simple update rules:

$$\begin{aligned}
\theta_0 &\leftarrow \theta_0 + \alpha \left(u_j(s) - \hat{U}_\theta(s)\right) , \\
\theta_1 &\leftarrow \theta_1 + \alpha \left(u_j(s) - \hat{U}_\theta(s)\right) x , \\
\theta_2 &\leftarrow \theta_2 + \alpha \left(u_j(s) - \hat{U}_\theta(s)\right) y .
\end{aligned}$$

We can apply these rules to the example where $\hat{U}_\theta(1,1)$ is 0.8 and $u_j(1,1)$ is 0.4. θ_0, θ_1, and θ_2 are all decreased by 0.4α, which reduces the error for (1,1). Notice that changing the θ_is *also changes the values of* $\hat{U}_\theta$ *for every other state*! This is what we mean by saying that function approximation allows a reinforcement learner to generalize from its experiences.

We expect that the agent will learn faster if it uses a function approximator, provided that the hypothesis space is not too large, but includes some functions that are a reasonably good fit to the true utility function. Exercise 21.7 asks you to evaluate the performance of direct utility estimation, both with and without function approximation. The improvement

in the 4×3 world is noticeable but not dramatic, because this is a very small state space to begin with. The improvement is much greater in a 10×10 world with a +1 reward at (10,10). This world is well suited for a linear utility function because the true utility function is smooth and nearly linear. (See Exercise 21.10.) If we put the +1 reward at (5,5), the true utility is more like a pyramid and the function approximator in Equation (21.9) will fail miserably. All is not lost, however! Remember that what matters for linear function approximation is that the function be linear in the *parameters*—the features themselves can be arbitrary nonlinear functions of the state variables. Hence, we can include a term such as $\theta_3 = \sqrt{(x - x_g)^2 + (y - y_g)^2}$ that measures the distance to the goal.

We can apply these ideas equally well to temporal-difference learners. All we need do is adjust the parameters to try to reduce the temporal difference between successive states. The new versions of the TD and Q-learning equations (21.3 and 21.8) are

$$\theta_i \leftarrow \theta_i + \alpha\,[R(s) + \gamma\,\hat{U}_\theta(s') - \hat{U}_\theta(s)]\frac{\partial \hat{U}_\theta(s)}{\partial \theta_i} \tag{21.11}$$

for utilities and

$$\theta_i \leftarrow \theta_i + \alpha\,[R(s) + \gamma \max_{a'} \hat{Q}_\theta(a', s') - \hat{Q}_\theta(a, s)]\frac{\partial \hat{Q}_\theta(a, s)}{\partial \theta_i} \tag{21.12}$$

for Q-values. These update rules can be shown to converge to the closest possible[5] approximation to the true function when the function approximator is *linear* in the parameters. Unfortunately, all bets are off when *nonlinear* functions—such as neural networks—are used. There are some very simple cases in which the parameters can go off to infinity even though there are good solutions in the hypothesis space. There are more sophisticated algorithms that can avoid these problems, but at present reinforcement learning with general function approximators remains a delicate art.

Function approximation can also be very helpful for learning a model of the environment. Remember that learning a model for an *observable* environment is a *supervised* learning problem, because the next percept gives the outcome state. Any of the supervised learning methods in Chapter 18 can be used, with suitable adjustments for the fact that we need to predict a complete state description rather than just a Boolean classification or a single real value. For example, if the state is defined by n Boolean variables, we will need to learn n Boolean functions to predict all the variables. For a *partially observable* environment, the learning problem is much more difficult. If we know what the hidden variables are and how they are causally related to each other and to the observable variables, then we can fix the structure of a dynamic Bayesian network and use the EM algorithm to learn the parameters, as was described in Chapter 20. Inventing the hidden variables and learning the model structure are still open problems.

We now turn to examples of large-scale applications of reinforcement learning. We will see that, in cases where a utility function (and hence a model) is used, the model is usually taken as given. For example, in learning an evaluation function for backgammon, it is normally assumed that the legal moves and their effects are known in advance.

[5] The definition of distance between utility functions is rather technical; see Tsitsiklis and Van Roy (1997).

Applications to game-playing

The first significant application of reinforcement learning was also the first significant learning program of any kind—the checker-playing program written by Arthur Samuel (1959, 1967). Samuel first used a weighted linear function for the evaluation of positions, using up to 16 terms at any one time. He applied a version of Equation (21.11) to update the weights. There were some significant differences, however, between his program and current methods. First, he updated the weights using the difference between the current state and the backed-up value generated by full look-ahead in the search tree. This works fine, because it amounts to viewing the state space at a different granularity. A second difference was that the program did *not* use any observed rewards! That is, the values of terminal states were ignored. This means that it is quite possible for Samuel's program not to converge, or to converge on a strategy designed to lose rather than to win. He managed to avoid this fate by insisting that the weight for material advantage should always be positive. Remarkably, this was sufficient to direct the program into areas of weight space corresponding to good checker play.

Gerry Tesauro's TD-Gammon system (1992) forcefully illustrates the potential of reinforcement learning techniques. In earlier work (Tesauro and Sejnowski, 1989), Tesauro tried learning a neural network representation of $Q(a, s)$ directly from examples of moves labeled with relative values by a human expert. This approach proved extremely tedious for the expert. It resulted in a program, called NEUROGAMMON, that was strong by computer standards, but not competitive with human experts. The TD-Gammon project was an attempt to learn from self-play alone. The only reward signal was given at the end of each game. The evaluation function was represented by a fully connected neural network with a single hidden layer containing 40 nodes. Simply by repeated application of Equation (21.11), TD-Gammon learned to play considerably better than Neurogammon, even though the input representation contained just the raw board position with no computed features. This took about 200,000 training games and two weeks of computer time. Although that may seem like a lot of games, it is only a vanishingly small fraction of the state space. When precomputed features were added to the input representation, a network with 80 hidden units was able, after 300,000 training games, to reach a standard of play comparable to that of the top three human players worldwide. Kit Woolsey, a top player and analyst, said that "There is no question in my mind that its positional judgment is far better than mine."

Application to robot control

CART–POLE

INVERTED PENDULUM

The setup for the famous **cart–pole** balancing problem, also known as the **inverted pendulum**, is shown in Figure 21.9. The problem is to control the position x of the cart so that the pole stays roughly upright ($\theta \approx \pi/2$), while staying within the limits of the cart track as shown. Over two thousand papers in reinforcement learning and control theory have been published on this seemingly simple problem. The cart–pole problem differs from the problems described earlier in that the state variables x, θ, $\dot{x}$, and $\dot{\theta}$ are continuous. The actions are usually discrete: jerk left or jerk right, the so-called **bang-bang control** regime.

BANG-BANG CONTROL

The earliest work on learning for this problem was carried out by Michie and Chambers (1968). Their BOXES algorithm was able to balance the pole for over an hour after only

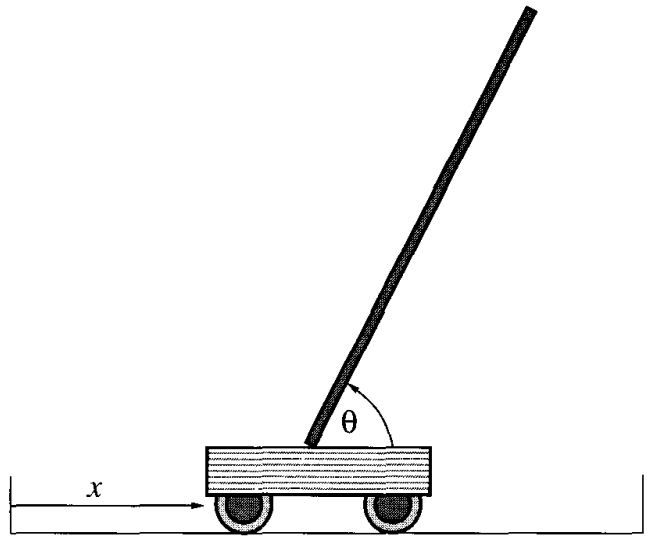


Figure 21.9 Setup for the problem of balancing a long pole on top of a moving cart. The cart can be jerked left or right by a controller that observes x, θ, $\dot{x}$, and $\dot{\theta}$.

about 30 trials. Moreover, unlike many subsequent systems, BOXES was implemented with a real cart and pole, not a simulation. The algorithm first discretized the four-dimensional state space into boxes—hence the name. It then ran trials until the pole fell over or the cart hit the end of the track. Negative reinforcement was associated with the final action in the final box and then propagated back through the sequence. It was found that the discretization caused some problems when the apparatus was initialized in a position different from those used in training, suggesting that generalization was not perfect. Improved generalization and faster learning can be obtained using an algorithm that *adaptively* partitions the state space according to the observed variation in the reward. Nowadays, balancing a *triple* inverted pendulum is a common exercise—a feat far beyond the capabilities of most humans.

21.5 POLICY SEARCH

POLICY SEARCH

The final approach we will consider for reinforcement learning problems is called **policy search**. In some ways, policy search is the simplest of all the methods in this chapter: the idea is to keep twiddling the policy as long as its performance improves, then stop.

Let us begin with the policies themselves. Remember that a policy π is a function that maps states to actions. We are interested primarily in *parameterized* representations of π that have far fewer parameters than there are states in the state space (just as in the preceding section). For example, we could represent π by a collection of parameterized Q-functions, one for each action, and take the action with the highest predicted value:

$$\pi(s) = \max_a \hat{Q}_\theta(a, s) \ . \tag{21.13}$$

Each Q-function could be a linear function of the parameters θ, as in Equation (21.9), or it could be a nonlinear function such as a neural network. Policy search will then adjust the parameters θ to improve the policy. Notice that if the policy is represented by Q-functions,

then policy search results in a process that learns Q-functions. *This process is not the same as Q-learning!* In Q-learning with function approximation, the algorithm finds a value of θ such that $\hat{Q}_\theta$ is "close" to Q^*, the optimal Q-function. Policy search, on the other hand, finds a value of θ that results in good performance; the values found may differ very substantially.[6] Another clear example of the difference is the case where $\pi(s)$ is calculated using, say, depth-10 look-ahead search with an approximate utility function $\hat{U}_\theta$. The value of θ that gives good play may be a long way from making $\hat{U}_\theta$ resemble the true utility function.

One problem with policy representations of the kind given in Equation (21.13) is that the policy is a *discontinuous* function of the parameters when the actions are discrete.[7] That is, there will be values of θ such that an infinitesimal change in θ causes the policy to switch from one action to another. This means that the value of the policy may also change discontinuously, which makes gradient-based search difficult. For this reason, policy search methods often use a **stochastic policy** representation $\pi_\theta(s, a)$, which specifies the *probability* of selecting action a in state s. One popular representation is the **softmax function**:

STOCHASTIC POLICY

SOFTMAX FUNCTION

$$\pi_\theta(s, a) = \exp(\hat{Q}_\theta(a, s)) / \sum_{a'} \exp(\hat{Q}_\theta(a', s)) \ .$$

Softmax becomes nearly deterministic if one action is much better than the others, but it always gives a differentiable function of θ; hence, the value of the policy (which depends in a continuous fashion on the action selection probabilities) is a differentiable function of θ.

Now let us look at methods for improving the policy. We start with the simplest case: a deterministic policy and a deterministic environment. In this case, evaluating the policy is trivial: we simply execute it and observe the accumulated reward; this gives us the **policy value** $\rho(\theta)$. Improving the policy is just a standard optimization problem, as described in Chapter 4. We can follow the **policy gradient** vector $\nabla_\theta \rho(\theta)$ provided $\rho(\theta)$ is differentiable. Alternatively, we can follow the **empirical gradient** by hillclimbing—i.e., evaluating the change in policy for small increments in each parameter value. With the usual caveats, this process will converge to a local optimum in policy space.

POLICY VALUE

POLICY GRADIENT

When the environment (or the policy) is stochastic, things get more difficult. Suppose we are trying to do hillclimbing, which requires comparing $\rho(\theta)$ and $\rho(\theta + \Delta\theta)$ for some small $\Delta\theta$. The problem is that the total reward on each trial may vary widely, so estimates of the policy value from a small number of trials will be quite unreliable; trying to compare two such estimates will be even more unreliable. One solution is simply to run lots of trials, measuring the sample variance and using it to determine that enough trials have been run to get a reliable indication of the direction of improvement for $\rho(\theta)$. Unfortunately, this is impractical for many real problems where each trial may be expensive, time-consuming, and perhaps even dangerous.

For the case of a stochastic policy $\pi_\theta(s, a)$, it is possible to obtain an unbiased estimate of the gradient at θ, $\nabla_\theta \rho(\theta)$, directly from the results of trials executed at θ. For simplicity, we will derive this estimate for the simple case of a nonsequential environment in which the

[6] Trivially, the approximate Q-function defined by $\hat{Q}_\theta(a, s) = Q^*(a, s)/10$ gives optimal performance, even though it is not at all close to Q^*.

[7] For a continuous action space, the policy can be a smooth function of the parameters.

reward is obtained immediately after acting in the start state s_0. In this case, the policy value is just the expected value of the reward, and we have

$$\nabla_\theta \rho(\theta) = \nabla_\theta \sum_a \pi_\theta(s_0, a) R(a) = \sum_a (\nabla_\theta \pi_\theta(s_0, a)) R(a) .$$

Now we perform a simple trick so that this summation can be approximated by samples generated from the probability distribution defined by $\pi_\theta(s_0, a)$. Suppose that we have N trials in all and the action taken on the jth trial is a_j. Then

$$\nabla_\theta \rho(\theta) = \sum_a \pi_\theta(s_0, a) \cdot \frac{(\nabla_\theta \pi_\theta(s_0, a)) R(a)}{\pi_\theta(s_0, a)} \approx \frac{1}{N} \sum_{j=1}^{N} \frac{(\nabla_\theta \pi_\theta(s_0, a_j)) R(a_j)}{\pi_\theta(s_0, a_j)} .$$

Thus, the true gradient of the policy value is approximated by a sum of terms involving the gradient of the action selection probability in each trial. For the sequential case, this generalizes to

$$\nabla_\theta \rho(\theta) \approx \frac{1}{N} \sum_{j=1}^{N} \frac{(\nabla_\theta \pi_\theta(s, a_j)) R_j(s)}{\pi_\theta(s, a_j)}$$

for each state s visited, where a_j is executed in s on the jth trial and $R_j(s)$ is the total reward received from state s onwards in the jth trial. The resulting algorithm is called REINFORCE (Williams, 1992); it is usually much more effective than hillclimbing using lots of trials at each value of θ. It is still much slower than necessary, however.

Consider the following task: given two blackjack[8] programs, determine which is best. One way to do this is to have each play against a standard "dealer" for a certain number of hands and then to measure their respective winnings. The problem with this, as we have seen, is that the winnings of each program fluctuate widely depending on whether it receives good or bad cards. An obvious solution is to generate a certain number of hands in advance and *set of hands*. In this way, we eliminate the measurement error due to differences in the cards received. This is the idea behind the PEGASUS algorithm (Ng and Jordan, 2000). The algorithm is applicable to domains for which a simulator is available so that the "random" outcomes of actions can be repeated. The algorithm works by generating in advance N sequences of random numbers, each of which can be used to run a trial of any policy. Policy search is carried out by evaluating each candidate policy using the *same* set of random sequences to determine the action outcomes. It can be shown that the number of random sequences required to ensure that the value of *every* policy is well-estimated depends only on the complexity of the policy space, and not at all on the complexity of the underlying domain. The PEGASUS algorithm has been used to develop effective policies for several domains, including autonomous helicopter flight (see Figure 21.10).

[8] Also known as twenty-one or pontoon.

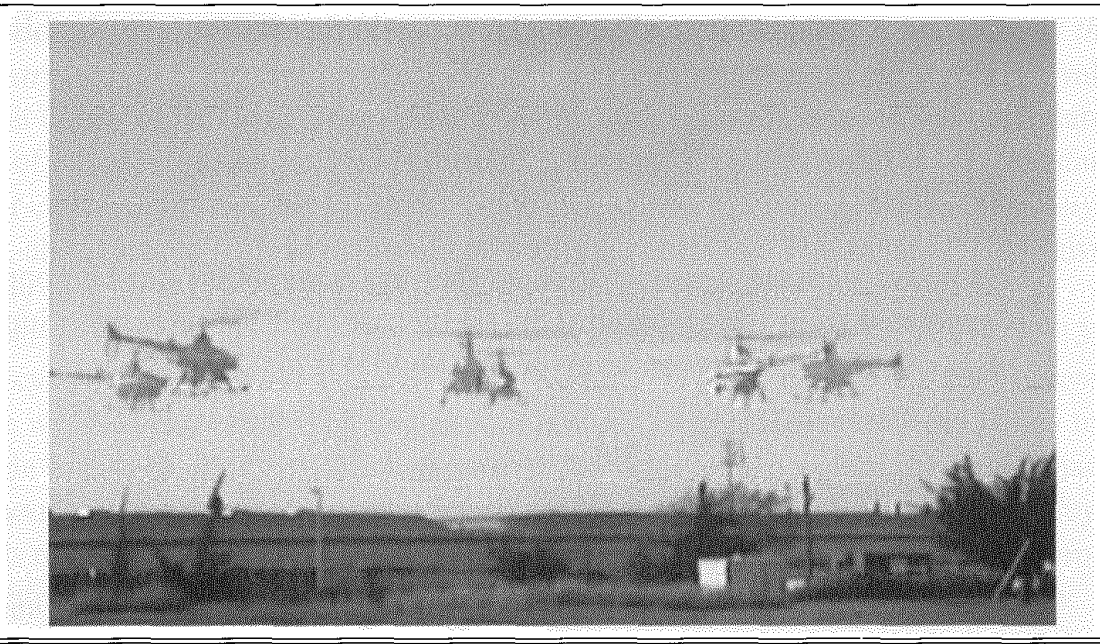


Figure 21.10 Superimposed time-lapse images of an autonomous helicopter performing a very difficult "nose-in circle" maneuver. The helicopter is under the control of a policy developed by the PEGASUS policy search algorithm. A simulator model was developed by observing the effects of various control manipulations on the real helicopter; then the algorithm was run on the simulator model overnight. A variety of controllers were developed for different maneuvers. In all cases, performance far exceeded that of an expert human pilot using remote control. (Image courtesy of Andrew Ng.)

21.6 SUMMARY

This chapter has examined the reinforcement learning problem: how an agent can become proficient in an unknown environment, given only its percepts and occasional rewards. Reinforcement learning can be viewed as a microcosm for the entire AI problem, but it is studied in a number of simplified settings to facilitate progress. The major points are:

- The overall agent design dictates the kind of information that must be learned. The three main designs we covered were the model-based design, using a model T and a utility function U; the model-free design, using an action-value function Q; and the reflex design, using a policy π.
- Utilities can be learned using three approaches:
  1. **Direct utility estimation** uses the total observed reward-to-go for a given state as direct evidence for learning its utility.
  2. **Adaptive dynamic programming** (ADP) learns a model and a reward function from observations and then uses value or policy iteration to obtain the utilities or an optimal policy. ADP makes optimal use of the local constraints on utilities of states imposed through the neighborhood structure of the environment.
  3. **Temporal-difference** (TD) methods update utility estimates to match those of successor states. They can be viewed as simple approximations to the ADP approach

that require no model for the learning process. Using a learned model to generate pseudoexperiences can, however, result in faster learning.

- Action-value functions, or Q-functions, can be learned by an ADP approach or a TD approach. With TD, Q-learning requires no model in either the learning or action-selection phase. This simplifies the learning problem but potentially restricts the ability to learn in complex environments, because the agent cannot simulate the results of possible courses of action.
- When the learning agent is responsible for selecting actions while it learns, it must trade off the estimated value of those actions against the potential for learning useful new information. An exact solution of the exploration problem is infeasible, but some simple heuristics do a reasonable job.
- In large state spaces, reinforcement learning algorithms must use an approximate functional representation in order to generalize over states. The temporal-difference signal can be used directly to update parameters in representations such as neural networks.
- **Policy search** methods operate directly on a representation of the policy, attempting to improve it based on observed performance. The variance in the performance in a stochastic domain is a serious problem; for simulated domains this can be overcome by fixing the randomness in advance.

Because of its potential for eliminating hand coding of control strategies, reinforcement learning continues to be one of the most active areas of machine learning research. Applications in robotics promise to be particularly valuable; these will require methods for handling *continuous, high-dimensional, partially observable* environments in which successful behaviors may consist of thousands or even millions of primitive actions.

BIBLIOGRAPHICAL AND HISTORICAL NOTES

Turing (1948, 1950) proposed the reinforcement learning approach, although he was not convinced of its effectiveness, writing, "the use of punishments and rewards can at best be a part of the teaching process." Arthur Samuel's work (1959) was probably the earliest successful machine learning research. Although this work was informal and had a number of flaws, it contained most of the modern ideas in reinforcement learning, including temporal differencing and function approximation. Around the same time, researchers in adaptive control theory (Widrow and Hoff, 1960), building on work by Hebb (1949), were training simple networks using the delta rule. (This early connection between neural networks and reinforcement learning may have led to the persistent misperception that the latter is a subfield of the former.) The cart–pole work of Michie and Chambers (1968) can also be seen as a reinforcement learning method with a function approximator. The psychological literature on reinforcement learning is much older; Hilgard and Bower (1975) provide a good survey. Direct evidence for the operation of reinforcement learning in animals has been provided by investigations into the foraging behavior of bees; there is a clear neural correlate of the reward signal in the form of a large neuron mapping from the nectar intake sensors directly

to the motor cortex (Montague *et al.*, 1995). Research using single-cell recording suggests that the dopamine system in primate brains implements something resembling value function learning (Schultz *et al.*, 1997).

The connection between reinforcement learning and Markov decision processes was first made by Werbos (1977), but the development of reinforcement learning in AI stems from work at the University of Massachusetts in the early 1980s (Barto *et al.*, 1981). The paper by Sutton (1988) provides a good historical overview. Equation (21.3) in this chapter is a special case for $\lambda = 0$ of Sutton's general TD(λ) algorithm. TD(λ) updates the values of all states in a sequence leading up to each transition by an amount that drops off as λ^t for states t steps in the past. TD(1) is identical to the Widrow–Hoff or delta rule. Boyan (2002), building on work by Bradtke and Barto (1996), argues that TD(λ) and related algorithms make inefficient use of experiences; essentially, they are online regression algorithms that converge much more slowly than offline regression. His LSTD(λ) is an online algorithm that gives the same results as offline regression.

The combination of temporal difference learning with the model-based generation of simulated experiences was proposed in Sutton's DYNA architecture (Sutton, 1990). The idea of prioritized sweeping was introduced independently by Moore and Atkeson (1993) and Peng and Williams (1993). Q-learning was developed in Watkins's Ph.D. thesis (1989).

Bandit problems, which model the problem of exploration for nonsequential decisions, are analyzed in depth by Berry and Fristedt (1985). Optimal exploration strategies for several settings are obtainable using the technique called **Gittins indices** (Gittins, 1989). A variety of exploration methods for sequential decision problems are discussed by Barto *et al.* (1995). Kearns and Singh (1998) and Brafman and Tennenholtz (2000) describe algorithms that explore unknown environments and are guaranteed to converge on near-optimal policies in polynomial time.

Function approximation in reinforcement learning goes back to the work of Samuel, who used both linear and nonlinear evaluation functions and also used feature selection methods to reduce the feature space. Later methods include the **CMAC** (Cerebellar Model Articulation Controller) (Albus, 1975), which is essentially a sum of overlapping local kernel functions, and the associative neural networks of Barto *et al.* (1983). Neural networks are currently the most popular form of function approximator. The best known application is TD-Gammon (Tesauro, 1992, 1995), which was discussed in the chapter. One significant problem exhibited by neural-network-based TD learners is that they tend to forget earlier experiences, especially those in parts of the state space that are avoided once competence is achieved. This can result in catastrophic failure if such circumstances reappear. Function approximation based on **instance-based learning** can avoid this problem (Ormoneit and Sen, 2002; Forbes, 2002).

CMAC

The convergence of reinforcement learning algorithms using function approximation is an extremely technical subject. Results for TD learning have been progressively strengthened for the case of linear function approximators (Sutton, 1988; Dayan, 1992; Tsitsiklis and Van Roy, 1997), but several examples of divergence have been presented for nonlinear functions (see Tsitsiklis and Van Roy, 1997, for a discussion). Papavassiliou and Russell (1999) describe a new type of reinforcement learning that converges with any form of function ap-

proximator, provided that a best-fit approximation can be found for the observed data.

Policy search methods were brought to the fore by Williams (1992), who developed the REINFORCE family of algorithms. Later work by Marbach and Tsitsiklis (1998), Sutton *et al.* (2000), and Baxter and Bartlett (2000) strengthened and generalized the convergence results for policy search. The PEGASUS algorithm is due to Ng and Jordan (2000) although similar techniques appear in Van Roy's PhD thesis (1998). As we mentioned in the chapter, the performance of a *stochastic* policy is a continuous function of its parameters, which helps with gradient-based search methods. This is not the only benefit: Jaakkola *et al.* (1995) argue that stochastic policies actually work better than deterministic policies in partially observable environments, if both are limited to acting based on the current percept. (One reason is that the stochastic policy is less likely to get "stuck" because of some unseen hindrance.) Now, in Chapter 17 we pointed out that optimal policies in partially observable MDPs are deterministic functions of the *belief state* rather than the current percept, so we would expect still better results by keeping track of the belief state using the **filtering** methods of Chapter 15. Unfortunately, belief state space is high-dimensional and continuous, and effective algorithms have not yet been developed for reinforcement learning with belief states.

Real-world environments also exhibit enormous complexity in terms of the number of primitive actions required to achieve significant reward. For example, a robot playing soccer might make a hundred thousand individual leg motions before scoring a goal. One common method, used originally in animal training, is called **reward shaping**. This involves supplying the agent with additional rewards for "making progress." For soccer, these might be given for making contact with the ball or for kicking it toward the goal. Such rewards can speed up learning enormously, and are very simple to provide, but there is a risk that the agent will learn to maximize the pseudorewards rather than the true rewards; for example, standing next to the ball and "vibrating" causes many contacts with the ball. Ng *et al.* (1999) show that the agent will still learn the optimal policy provided that the pseudoreward $F(s, a, s')$ satisfies $F(s, a, s') = \gamma\Phi(s') - \Phi(s)$, where Φ is an arbitrary function of state. Φ can be constructed to reflect any desirable aspects of the state, such as achievement of subgoals or distance to a goal state.

REWARD SHAPING

The generation of complex behaviors can also be facilitated by **hierarchical reinforcement learning** methods, which attempt to solve problems at multiple levels of abstraction—much like the **HTN planning** methods of Chapter 12. For example, "scoring a goal" can be broken down into "obtain possession," "dribble towards the goal," and "shoot;" and each of these can be broken down further into lower-level motor behaviors. The fundamental result in this area is due to Forestier and Varaiya (1978), who proved that lower-level behaviors of arbitrary complexity can be treated just like primitive actions (albeit ones that can take varying amounts of time) from the point of view of the higher-level behavior that invokes them. Current approaches (Parr and Russell, 1998; Dietterich, 2000; Sutton *et al.*, 2000; Andre and Russell, 2002) build on this result to develop methods for supplying an agent with a **partial program** that constrains the agent's behavior to have a particular hierarchical structure. Reinforcement learning is then applied to learn the best behavior consistent with the partial program. The combination of function approximation, shaping, and hierarchical reinforcement learning may enable large-scale problems to be tackled successfully.

HIERARCHICAL REINFORCEMENT LEARNING

PARTIAL PROGRAM

The survey by Kaelbling *et al.* (1996) provides a good entry point to the literature. The text by Sutton and Barto (1998), two of the field's pioneers, focuses on architectures and algorithms, showing how reinforcement learning weaves together the ideas of learning, planning, and acting. The somewhat more technical work by Bertsekas and Tsitsiklis (1996) gives a rigorous grounding in the theory of dynamic programming and stochastic convergence. Reinforcement learning papers are published frequently in *Machine Learning*, in the *Journal of Machine Learning Research*, and in the International Conferences on Machine Learning and the Neural Information Processing Systems meetings.

EXERCISES

21.1 Implement a passive learning agent in a simple environment, such as the 4×3 world. For the case of an initially unknown environment model, compare the learning performance of the direct utility estimation, TD, and ADP algorithms. Do the comparison for the optimal policy and for several random policies. For which do the utility estimates converge faster? What happens when the size of the environment is increased? (Try environments with and without obstacles.)

21.2 Chapter 17 defined a **proper policy** for an MDP as one that is guaranteed to reach a terminal state. Show that it is possible for a passive ADP agent to learn a transition model for which its policy π is improper even if π is proper for the true MDP; with such models, the value determination step may fail if $\gamma = 1$. Show that this problem cannot arise if value determination is applied to the learned model only at the end of a trial.

21.3 Starting with the passive ADP agent, modify it to use an approximate ADP algorithm as discussed in the text. Do this in two steps:

a. Implement a priority queue for adjustments to the utility estimates. Whenever a state is adjusted, all of its predecessors also become candidates for adjustment and should be added to the queue. The queue is initialized with the state from which the most recent transition took place. Allow only a fixed number of adjustments.

b. Experiment with various heuristics for ordering the priority queue, examining their effect on learning rates and computation time.

21.4 The direct utility estimation method in Section 21.2 uses distinguished terminal states to indicate the end of a trial. How could it be modified for environments with discounted rewards and no terminal states?

21.5 How can the value determination algorithm be used to calculate the expected loss experienced by an agent using a given set of utility estimates U and an estimated model M, compared with an agent using correct values?

21.6 Adapt the vacuum world (Chapter 2) for reinforcement learning by including rewards for picking up each piece of dirt and for getting home and switching off. Make the world accessible by providing suitable percepts. Now experiment with different reinforcement learn-

ing agents. Is function approximation necessary for success? What sort of approximator works for this application?

21.7 Implement an exploring reinforcement learning agent that uses direct utility estimation. Make two versions—one with a tabular representation and one using the function approximator in Equation (21.9). Compare their performance in three environments:

a. The 4×3 world described in the chapter.

b. A 10×10 world with no obstacles and a +1 reward at (10,10).

c. A 10×10 world with no obstacles and a +1 reward at (5,5).

21.8 Write out the parameter update equations for TD learning with

$$\hat{U}(x, y) = \theta_0 + \theta_1 x + \theta_2 y + \theta_3 \sqrt{(x - x_g)^2 + (y - y_g)^2} \, .$$

21.9 Devise suitable features for stochastic grid worlds (generalizations of the 4×3 world) that contain multiple obstacles and multiple terminal states with $+1$ or -1 rewards.

21.10 Compute the true utility function and the best linear approximation in x and y (as in Equation (21.9)) for the following environments:

a. A 10×10 world with a single $+1$ terminal state at (10,10).

b. As in (a), but add a -1 terminal state at (10,1).

c. As in (b), but add obstacles in 10 randomly selected squares.

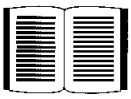
d. As in (b), but place a wall stretching from (5,2) to (5,9).

e. As in (a), but with the terminal state at (5,5).

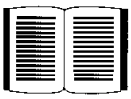
The actions are deterministic moves in the four directions. In each case, compare the results using three-dimensional plots. For each environment, propose additional features (besides x and y) that would improve the approximation and show the results.

21.11 Extend the standard game-playing environment (Chapter 6) to incorporate a reward signal. Put two reinforcement learning agents into the environment (they may, of course, share the agent program) and have them play against each other. Apply the generalized TD update rule (Equation (21.11)) to update the evaluation function. You might wish to start with a simple linear weighted evaluation function and a simple game, such as tic-tac-toe.

21.12 Implement the REINFORCE and PEGASUS algorithms and apply them to the 4×3 world, using a policy family of your own choosing. Comment on the results.



21.13 Investigate the application of reinforcement learning ideas to the modeling of human and animal behavior.



21.14 Is reinforcement learning an appropriate abstract model for evolution? What connection exists, if any, between hardwired reward signals and evolutionary fitness?

22 COMMUNICATION

In which we see why agents might want to exchange information-carrying messages with each other and how they can do so.

It is dusk in the savanna woodlands of Amboseli National Park near the base of Kilimanjaro. A group of vervet monkeys are foraging for food when one lets out a loud barking call. The others in the group recognize this as the leopard warning call (distinct from the short cough used to warn of eagles, or the chutter for snakes) and scramble for the trees. The vervet has successfully communicated with the group.

COMMUNICATION

SIGNS

Communication is the intentional exchange of information brought about by the production and perception of **signs** drawn from a shared system of conventional signs. Most animals use signs to represent important messages: food here, predator nearby, approach, withdraw, let's mate. In a partially observable world, communication can help agents be successful because they can learn information that is observed or inferred by others.

LANGUAGE

What sets humans apart from other animals is the complex system of structured messages known as **language** that enables us to communicate most of what we know about the world. Although chimpanzees, dolphins, and other mammals have shown vocabularies of hundreds of signs and some aptitude for stringing them together, only humans can reliably communicate an unbounded number of qualitatively different messages.

Of course, there are other attributes that are uniquely human: no other species wears clothes, creates representational art, or watches three hours of television a day. But when Turing proposed his test (see Section 1.1), he based it on language, because language is intimately tied to thinking. In this chapter, we will both explain how a communicating agent works and describe a fragment of English.

22.1 COMMUNICATION AS ACTION

SPEECH ACT

One of the actions available to an agent is to produce language. This is called a **speech act**. "Speech" is used in the same sense as in "free speech," not "talking," so e-mailing, skywriting, and using sign language all count as speech acts. English has no neutral word for an agent that produces language by any means, so we will use **speaker**, **hearer**, and **utterance** as generic

SPEAKER

HEARER

UTTERANCE

WORD

terms referring to any mode of communication. We will also use the term **word** to refer to any kind of conventional communicative sign.

Why would an agent bother to perform a speech act when it could be doing a "regular" action? We saw in Chapter 12 that agents in a multiagent environment can use communication to help arrive at joint plans. For example, a group of agents exploring the wumpus world together gains an advantage (collectively and individually) by being able to do the following:

- **Query** other agents about particular aspects of the world. This is typically done by asking questions: *Have you smelled the wumpus anywhere?*
- **Inform** each other about the world. This is done by making representative statements: *There's a breeze here in 3 4.* Answering a question is another kind of informing.
- **Request** other agents to perform actions: *Please help me carry the gold.* Sometimes an **indirect speech act** (a request in the form of a statement or question) is considered more polite: *I could use some help carrying this.* An agent with authority can give commands (*Alpha go right; Bravo and Charlie go left*), and an agent with power can make a threat (*Give me the gold, or else*). Together, these kinds of speech acts are called **directives**.

INDIRECT SPEECH ACT

- **Acknowledge** requests: *OK.*
- **Promise** or commit to a plan: *I'll shoot the wumpus; you grab the gold.*

All speech acts affect the world by making air molecules vibrate (or the equivalent effect in some other medium) and thereby changing the mental state and eventually the future actions of other agents. Some kinds of speech acts transfer information to the hearer, assuming that the hearer's decision making will be suitably affected by that information. Others are aimed more directly at making the hearer take some action. Another class of speech act, the **declarative**, appears to have a more direct effect on the world, as in *I now pronounce you man and wife* or *Strike three, you're out.* Of course, the effect is achieved by creating or confirming a complex web of mental states among the agents involved: being married and being out are states characterized primarily by convention rather than by "physical" properties of the world.

DECLARATIVE

The communicating agent's task is to decide *when* a speech act of some kind is called for and *which* speech act, out of all the possibilities, is the right one. The problem of understanding speech acts is much like other **understanding** problems, such as understanding images or diagnosing illnesses. We are given a set of ambiguous inputs, and from them we have to work backwards to decide what state of the world could have created these inputs. However, because speech is a planned action, understanding it also involves plan recognition.

UNDERSTANDING

Fundamentals of language

FORMAL LANGUAGE

STRINGS

TERMINAL SYMBOLS

A **formal language** is defined as a (possibly infinite) set of **strings**. Each string is a concatenation of **terminal symbols**, sometimes called words. For example, in the language of first-order logic, the terminal symbols include $\wedge$ and P, and a typical string is "$P \wedge Q$." The string "$P\ Q \wedge$" is not a member of the language. Formal languages such as first-order logic and Java have strict mathematical definitions. This is in contrast to **natural languages**, such as Chinese, Danish, and English, that have no strict definition but are used by a community

NATURAL LANGUAGES

of speakers. For this chapter we will attempt to treat natural languages as if they were formal languages, although we recognize the match will not be perfect.

GRAMMAR

A **grammar** is a finite set of rules that specifies a language. Formal languages always have an official grammar, specified in manuals or books. Natural languages have no official grammar, but linguists strive to discover properties of the language by a process of scientific inquiry and then to codify their discoveries in a grammar. To date, no linguist has succeeded completely. Note that linguists are scientists, attempting to define a language as it *is*. There are also prescriptive grammarians who try to dictate how a language *should be*. They create rules such as "Don't split infinitives" which are sometimes printed in style guides, but have little relevance to actual language usage.

Both formal and natural languages associate a meaning or **semantics** to each valid string. For example, in the language of arithmetic, we would have a rule saying that if "X" and "Y" are expressions, then "$X + Y$" is also an expression, and its semantics is the sum of X and Y. In natural languages, it is also important to understand the **pragmatics** of a string: the actual meaning of the string as it is spoken in a given situation. The meaning is not just in the words themselves, but in the interpretation of the words *in situ*.

PRAGMATICS

PHRASE STRUCTURE

Most grammar rule formalisms are based on the idea of **phrase structure**—that strings are composed of substrings called **phrases**, which come in different categories. For example, the phrases "the wumpus," "the king," and "the agent in the corner" are all examples of the category **noun phrase**, or NP. There are two reasons for identifying phrases in this way. First, phrases usually correspond to natural semantic elements from which the meaning of an utterance can be constructed; for example, noun phrases refer to objects in the world. Second, categorizing phrases helps us to describe the allowable strings of the language. We can say that any of the noun phrases can combine with a **verb phrase** (or VP) such as "is dead" to form a phrase of category **sentence** (or S). Without the intermediate notions of noun phrase and verb phrase, it would be difficult to explain why "the wumpus is dead" is a sentence whereas "wumpus the dead is" is not.

NOUN PHRASE

VERB PHRASE

SENTENCE

NONTERMINAL SYMBOLS

REWRITE RULES

Category names such as NP, VP, and S are called **nonterminal symbols**. Grammars define nonterminals using **rewrite rules**. We will adopt the Backus–Naur form (BNF) notation for rewrite rules, which is described in Appendix B on page 984. In this notation, the meaning of a rule such as

$$S \rightarrow NP\ VP$$

is that an S may consist of any NP followed by any VP.

The component steps of communication

A typical communication episode, in which speaker S wants to inform hearer H about proposition P using words W, is composed of seven processes:

INTENTION

Intention. Somehow, speaker S decides that there is some proposition P that is worth saying to hearer H. For our example, the speaker has the intention of having the hearer know that the wumpus is no longer alive.

GENERATION

Generation. The speaker plans how to turn the proposition P into an utterance that makes it likely that the hearer, upon perceiving the utterance in the current situation, can infer

GENERATIVE CAPACITY

Grammatical formalisms can be classified by their **generative capacity**: the set of languages they can represent. Chomsky (1957) describes four classes of grammatical formalisms that differ only in the form of the rewrite rules. The classes can be arranged in a hierarchy, where each class can be used to describe all the languages that can be described by a less powerful class, as well as some additional languages. Here we list the hierarchy, most powerful class first:

Recursively enumerable grammars use unrestricted rules: both sides of the rewrite rules can have any number of terminal and nonterminal symbols, as in the rule $A\ B \rightarrow C$. These grammars are equivalent to Turing machines in their expressive power.

Context-sensitive grammars are restricted only in that the right-hand side must contain at least as many symbols as the left-hand side. The name "context-sensitive" comes from the fact that a rule such as $A\ S\ B \rightarrow A\ X\ B$ says that an S can be rewritten as an X in the context of a preceding A and a following B. Context-sensitive grammars can represent languages such as $a^n b^n c^n$ (a sequence of n copies of a followed by the same number of bs and then cs).

In **context-free grammars** (or **CFG**s), the left-hand side consists of a single nonterminal symbol. Thus, each rule licenses rewriting the nonterminal as the right-hand side in *any* context. CFGs are popular for natural language and programming language grammars, although it is now widely accepted that at least some natural languages have constructions that are not context-free (Pullum, 1991). Context-free grammars can represent $a^n b^n$, but not $a^n b^n c^n$.

Regular grammars are the most restricted class. Every rule has a single nonterminal on the left-hand side and a terminal symbol optionally followed by a nonterminal on the right-hand side. Regular grammars are equivalent in power to finite-state machines. They are poorly suited for programming languages, because they cannot represent constructs such as balanced opening and closing parentheses (a variation of the $a^n b^n$ language). The closest they can come is representing $a^* b^*$, a sequence of any number of as followed by any number of bs.

The grammars higher up in the hierarchy have more expressive power, but the algorithms for dealing with them are less efficient. Up to the mid 1980s, linguists focused on context-free and context-sensitive languages. Since then, there has been increased emphasis on regular grammars, brought about by the need to process megabytes and gigabytes of online text very quickly, even at the cost of a less complete analysis. As Fernando Pereira put it, "The older I get, the further down the Chomsky hierarchy I go." To see what he means, compare Pereira and Warren (1980) with Mohri, Pereira, and Riley (2002).

the meaning P (or something close to it). Assume that the speaker is able to come up with the words "The wumpus is dead," and call this W.

SYNTHESIS

Synthesis. The speaker produces the physical realization W' of the words W. This can be via ink on paper, vibrations in air, or some other medium. In Figure 22.1, we show the agent synthesizing a string of sounds W' written in the phonetic alphabet defined on page 569: "[thaxwahmpaxsihzdehd]." The words are run together; this is typical of quickly spoken speech.

PERCEPTION

Perception. H perceives the physical realization W' as W_2' and decodes it as the words W_2. When the medium is speech, the perception step is called **speech recognition**; when it is printing, it is called **optical character recognition**. Both moved from being esoteric to being commonplace in the 1990s, due largely to increased desktop computing power.

ANALYSIS

Analysis. H infers that W_2 has possible meanings $P_1, \ldots, P_n$. We divide analysis into three main parts: syntactic interpretation (or parsing), semantic interpretation, and pragmatic interpretation. **Parsing** is the process of building a **parse tree** for an input string, as shown in Figure 22.1. The interior nodes of the parse tree represent phrases and the leaf nodes represent words. **Semantic interpretation** is the process of extracting the meaning of an utterance as an expression in some representation language. Figure 22.1 shows two possible semantic interpretations: that the wumpus is not alive and that it is tired (a colloquial meaning of *dead*). Utterances with several possible interpretations are said to be **ambiguous**. **Pragmatic interpretation** takes into account the fact that the same words can have different meanings in

PARSING

PARSE TREE

SEMANTIC INTERPRETATION

PRAGMATIC INTERPRETATION

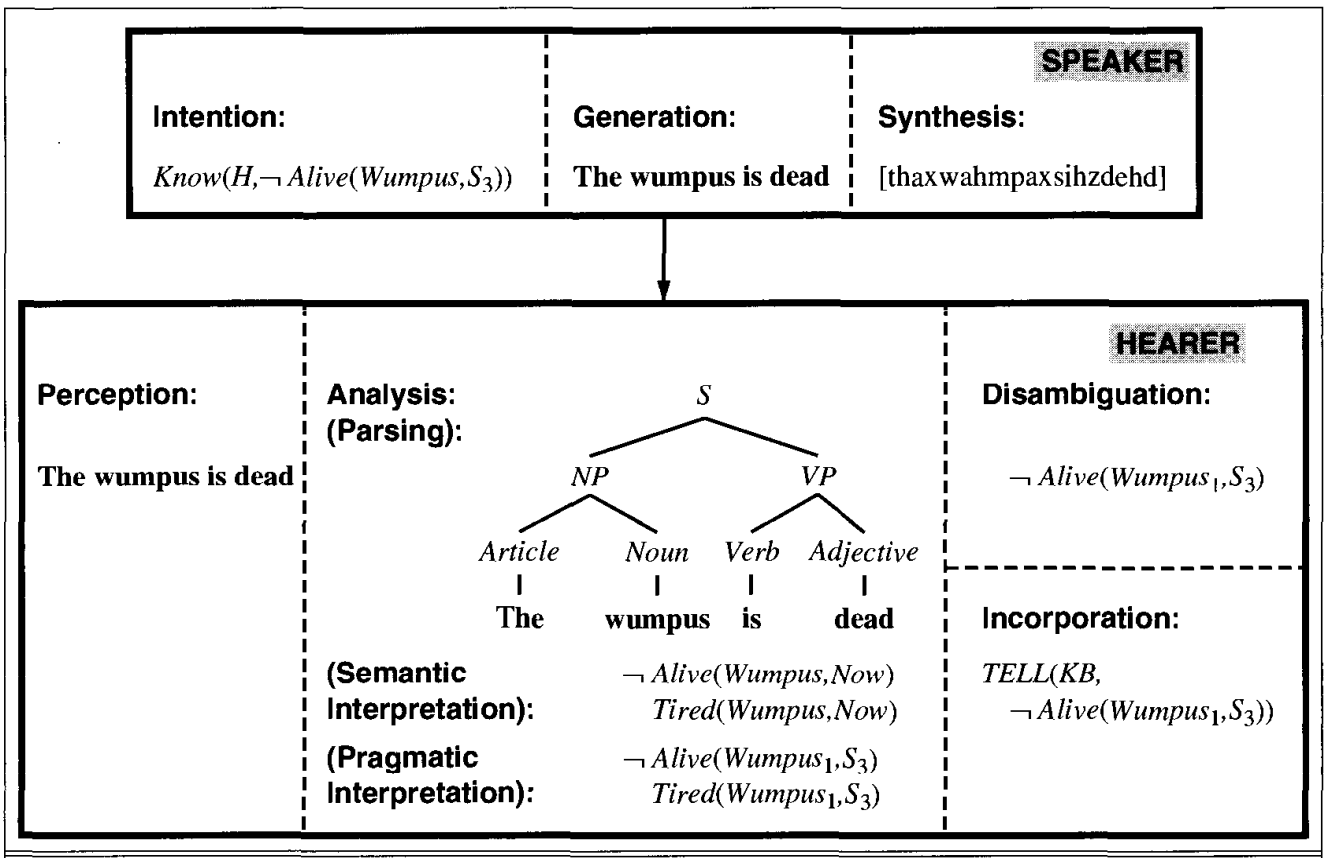


Figure 22.1 Seven processes involved in communication, using the example sentence "The wumpus is dead."

different situations. Whereas syntactic interpretation is a function of one argument, the string, pragmatic interpretation is a function of the utterance and the context or situation in which it is uttered. In the example, pragmatics does two things: replace the constant *Now* with the constant S_3, which stands for the current situation, and replace $Wumpus$ with $Wumpus_1$, which stands for the single Wumpus that is known to be in this cave. In general, pragmatics can contribute much more to the final interpretation of an utterance; consider "I'm looking at the diamond" when spoken by a jeweler or by a baseball player. In Section 22.7, we will see that pragmatics allows us to interpret "It is dead" as meaning that the wumpus is dead if we are in a situation where the wumpus is salient.

DISAMBIGUATION **Disambiguation**. H infers that S intended to convey P_i (where ideally $P_i = P$). Most speakers are not intentionally ambiguous, but most utterances have several feasible interpretations. Communication works because the hearer does the work of figuring out which interpretation is the one the speaker probably meant to convey. Notice that this is the first time we have used the word *probably*, and disambiguation is the first process that depends heavily on uncertain reasoning. Analysis generates possible interpretations; if more than one interpretation is found, then disambiguation chooses the one that is best.

INCORPORATION **Incorporation**. H decides to believe P_i (or not). A totally naive agent might believe everything it hears, but a sophisticated agent treats the speech act as evidence for P_i, not confirmation of it.

Putting it all together, we get the agent program shown in Figure 22.2. Here the agent acts as a robot slave that can be commanded by a master. On each turn, the slave will answer a question or obey a command if the master has made one, and it will believe any statements made by the master. It will also comment (once) on the current situation if it has nothing more pressing to do, and it will plan its own action if left alone. Here is a typical dialog:

ROBOT SLAVE	MASTER
I feel a breeze.	Go to 1 2.
Nothing is here.	Go north.
I feel a breeze and I smell a stench and I see a glitter.	Grab the gold.

22.2 A FORMAL GRAMMAR FOR A FRAGMENT OF ENGLISH

In this section, we define a formal grammar for a small fragment of English that is suitable for making statements about the wumpus world. We will call this language $\mathcal{E}_0$. Later sections will improve on $\mathcal{E}_0$ to make it somewhat closer to real English. We are unlikely ever to devise a complete grammar for English, if only because no two persons would agree entirely on what constitutes valid English.

The Lexicon of $\mathcal{E}_0$

LEXICON First we define the **lexicon**, or list of allowable words. The words are grouped into the categories or parts of speech familiar to dictionary users: nouns, pronouns, and names to denote

```
function NAIVE-COMMUNICATING-AGENT(percept) returns action
  static: KB, a knowledge base
          state, the current state of the environment
          action, the most recent action, initially none

  state ← UPDATE-STATE(state, action, percept)
  words ← SPEECH-PART(percept)
  semantics ← DISAMBIGUATE(PRAGMATICS(SEMANTICS(PARSE(words))))
  if words = None and action is not a SAY then /* Describe the state */
     return SAY(GENERATE-DESCRIPTION(state))
  else if TYPE[semantics] = Command then /* Obey the command */
     return CONTENTS[semantics]
  else if TYPE[semantics] = Question then /* Answer the question */
     answer ← ASK(KB, semantics)
     return SAY(GENERATE-DESCRIPTION(answer))
  else if TYPE[semantics] = Statement then /* Believe the statement */
     TELL(KB, CONTENTS[semantics])
  /* If we fall through to here, do a "regular" action */
  return FIRST(PLANNER(KB, state))
```

Figure 22.2 A communicating agent that accepts commands, questions, and statements. The agent can also describe the current state or perform a "regular" non-speech-act action when there is nothing to say.

things, verbs to denote events, adjectives to modify nouns, and adverbs to modify verbs. Categories that are perhaps less familiar to some readers are articles (such as *the*), prepositions (*in*), and conjunctions (*and*). Figure 22.3 shows a small lexicon.

Each of the categories ends in ... to indicate that there are other words in the category. However, it should be noted that there are two distinct reasons for the missing words. For nouns, verbs, adjectives, and adverbs, it is in principle infeasible to list them all. Not only are there tens of thousands of members in each class, but new ones–like *MP3* or *anime*—are being added constantly. These four categories are called **open classes**. The other categories (pronoun, article, preposition, and conjunction) are called **closed classes**. They have a small number of words (a few to a few dozen) that can in principle be enumerated in full. Closed classes change over the course of centuries, not months. For example, "thee" and "thou" were commonly used pronouns in the 17th century, were on the decline in the 19th, and are seen today only in poetry and some regional dialects.

OPEN CLASSES

CLOSED CLASSES

The Grammar of $\mathcal{E}_0$

The next step is to combine the words into phrases. We will use five nonterminal symbols to define the different kinds of phrases: sentence (S), noun phrase (NP), verb phrase (VP),

Noun	→	**stench** | **breeze** | **glitter** | **nothing** | **agent**
		| **wumpus** | **pit** | **pits**| **gold** | **east** | ...
Verb	→	**is** | **see** | **smell** | **shoot** | **feel** | **stinks**
		| **go** | **grab** | **carry** | **kill** | **turn** | ...
Adjective	→	**right** | **left** | **east** | **dead** | **back** | **smelly** | ...
Adverb	→	**here** | **there** | **nearby** | **ahead**
		| **right** | **left** | **east** | **south** | **back** | ...
Pronoun	→	**me** | **you** | **I** | **it** | ...
Name	→	**John** | **Mary** | **Boston** | **Aristotle** | ...
Article	→	**the** | **a** | **an** | ...
Preposition	→	**to** | **in** | **on** | **near** | ...
Conjunction	→	**and** | **or** | **but** | ...
Digit	→	**0**| **1**| **2**| **3**| **4**| **5**| **6**| **7**| **8**| **9**

Figure 22.3 The lexicon for $\mathcal{E}_0$.

S	→	*NP VP*	I + feel a breeze
	|	*S Conjunction S*	I feel a breeze + and + I smell a wumpus
NP	→	*Pronoun*	I
	|	*Name*	John
	|	*Noun*	pits
	|	*Article Noun*	the + wumpus
	|	*Digit Digit*	3 4
	|	*NP PP*	the wumpus + to the east
	|	*NP RelClause*	the wumpus + that is smelly
VP	→	*Verb*	stinks
	|	*VP NP*	feel + a breeze
	|	*VP Adjective*	is + smelly
	|	*VP PP*	turn + to the east
	|	*VP Adverb*	go + ahead
PP	→	*Preposition NP*	to + the east
RelClause	→	**that** *VP*	that + is smelly

Figure 22.4 The grammar for $\mathcal{E}_0$, with example phrases for each rule.

prepositional phrase (PP), and relative clause ($RelClause$).[1] Figure 22.4 shows a grammar for $\mathcal{E}_0$, with an example for each rewrite rule. $\mathcal{E}_0$ generates good English sentences such as the following:

John is in the pit
The wumpus that stinks is in 2 2
Mary is in Boston and John stinks

OVERGENERATION

UNDERGENERATION

Unfortunately, the grammar **overgenerates**: that is, it generates sentences that are not grammatical, such as "Me go Boston" and "I smell pit gold wumpus nothing east." It also **undergenerates**: there are many sentences of English that it rejects, such as "I think the wumpus is smelly." (Another shortcoming is that the grammar does not capitalize the first word of a sentence, nor add punctuation at the end. That is because it is designed primarily for speech, not writing.)

22.3 Syntactic Analysis (Parsing)

We have already defined **parsing** as the process of finding a parse tree for a given input string. That is, a call to the parsing function PARSE, such as

PARSE("the wumpus is dead", $\mathcal{E}_0$, S)

should return a parse tree with root S whose leaves are "the wumpus is dead" and whose internal nodes are nonterminal symbols from the grammar $\mathcal{E}_0$. You can see such a tree in Figure 22.1. In linear text, we write the tree as

[S: [NP: [$Article$: **the**][$Noun$: **wumpus**]]
[VP: [$Verb$: **is**][$Adjective$: **dead**]]] .

Parsing can be seen as a process of searching for a parse tree. There are two extreme ways of specifying the search space (and many variants in between). First, we can start with the S symbol and search for a tree that has the words as its leaves. This is called **top-down parsing** (because the S is drawn at the top of the tree). Second, we could start with the words and search for a tree with root S. This is called **bottom-up parsing**.[2] Top-down parsing can be precisely defined as a search problem as follows:

TOP-DOWN PARSING

BOTTOM-UP PARSING

- The **initial state** is a parse tree consisting of the root S and unknown children: [S: ?]. In general, each state in the search space is a parse tree.
- The **successor function** selects the leftmost node in the tree with unknown children. It then looks in the grammar for rules that have the root label of the node on the left-hand side. For each such rule, it creates a successor state where the ? is replaced by a list corresponding to the right-hand side of the rule. For example, in $\mathcal{E}_0$ there are two rules for S, so the tree [S: ?] would be replaced by the following two successors:

[1] A relative clause follows and modifies a noun phrase. It consists of a relative pronoun (such as "who" or "that") followed by a verb phrase. (Another kind of relative clause is discussed in exercise 22.12.) An example of a relative clause is *that stinks* in "The wumpus *that stinks* is in 2 2."

[2] The reader might notice that top-down and bottom-up parsing are analogous to backward and forward chaining, respectively, as described in Chapter 7. We will see shortly that the analogy is exact.

[S: [S: ?] [$Conjunction$: ?] [S: ?]]
[S: [NP: ?] [VP: ?]]

The second of these has seven successors, one for each rewrite rule of NP.

- The **goal test** checks that the leaves of the parse tree correspond exactly to the input string, with no unknowns and no uncovered inputs.

One big problem for top-down parsing is dealing with so-called **left-recursive rules**—that is, rules of the form $X \rightarrow X \ldots$. With a depth-first search, such a rule would lead us to keep replacing X with [X: X ...] in an infinite loop. With a breadth-first search we would successfully find parses for valid sentences, but when given an invalid sentence, we would get stuck in an infinite search space.

The formulation of bottom-up parsing as a search is as follows:

- The **initial state** is a list of the words in the input string, each viewed as a parse tree that is just a single leaf node—for example; [**the**, **wumpus**, **is**, **dead**]. In general, each state in the search space is a list of parse trees.
- The **successor function** looks at every position i in the list of trees and at every right-hand side of a rule in the grammar. If the subsequence of the list of trees starting at i matches the right-hand side, then the subsequence is replaced by a new tree whose category is the left-hand side of the rule and whose children are the subsequence. By "matches," we mean that the category of the node is the same as the element in the right-hand side. For example, the rule $Article \rightarrow$ **the** matches the subsequence consisting of the first node in the list [**the**, **wumpus**, **is**, **dead**], so a successor state would be [[$Article$: **the**], **wumpus**, **is**, **dead**].
- The **goal test** checks for a state consisting of a single tree with root S.

See Figure 22.5 for an example of bottom-up parsing.

step	*list of nodes*	*subsequence*	*rule*
INIT	**the wumpus is dead**	**the**	Article → **the**
2	*Article* **wumpus is dead**	**wumpus**	Noun → **wumpus**
3	*Article Noun* **is dead**	*Article Noun*	*NP* → *Article Noun*
4	*NP* **is dead**	**is**	*Verb* → **is**
5	*NP Verb* **dead**	**dead**	*Adjective* → **dead**
6	*NP Verb Adjective*	*Verb*	*VP* → *Verb*
7	*NP VP Adjective*	*VP Adjective*	*VP* → *VP Adjective*
8	*NP VP*	*NP VP*	*S* → *NP VP*
GOAL	*S*		

Figure 22.5 Trace of a bottom up parse on the string "The wumpus is dead." We start with a list of nodes consisting of words. Then we replace subsequences that match the right-hand side of a rule with a new node whose root is the left-hand side. For example, in the third line the *Article* and *Noun* nodes are replaced by an *NP* node that has those two nodes as children. The top-down parse would produce a similar trace, but in the opposite direction.

Both top-down and bottom-up parsing can be inefficient, because of the multiplicity of ways in which multiple parses for different phrases can be combined. Both can waste time searching irrelevant portions of the search space. Top-down parsing can generate intermediate nodes that could never be latched by the words, and bottom-up parsing can generate partial parses of the words that could not appear in an S.

Even if we had a perfect heuristic function that allowed us to search without any irrelevant digressions, these algorithms would still be inefficient, because some sentences have *exponentially many* parse trees. The next subsection shows what to do about that.

Efficient parsing

Consider the following two sentences:

Have the students in section 2 of Computer Science 101 take the exam.
Have the students in section 2 of Computer Science 101 taken the exam?

Even though they share the first 10 words, these sentences have very different parses, because the first is a command and the second is a question. A left-to-right parsing algorithm would have to guess whether the first word is part of a command or a question and will not be able to tell if the guess is correct until at least the eleventh word, *take* or *taken*. If the algorithm guesses wrong, it will have to backtrack all the way to the first word. This kind of backtracking is inevitable, but if our parsing algorithm is to be efficient, it must avoid reanalyzing "the students in section 2 of Computer Science 101" as an NP each time it backtracks.

In this section, we will develop a parsing algorithm that avoids this source of inefficiency. The basic idea is an example of **dynamic programming**: *every time we analyze a substring, store the results so we won't have to re-analyze it later.* For example, once we discover that "the students in section 2 of Computer Science 101" is an NP, we can record that result in a data structure known as a **chart**. Algorithms that do this are called **chart parsers**. Because we are dealing with context-free grammars, any phrase that was found in the context of one branch of the search space can work just as well in any other branch of the search space.

CHART

VERTICES

EDGES

The chart for an n-word sentence consists of $n+1$ **vertices** and a number of **edges** that connect vertices. Figure 22.6 shows a chart with six vertices (circles) and three edges (lines). For example, the edge labeled

$$[0, 5,\ S \rightarrow NP\ VP\ \bullet]$$

means that an NP followed by a VP combine to make an S that spans the string from 0 to 5. The symbol $\bullet$ in an edge separates what has been found so far from what remains to be found.[3] Edges with $\bullet$ at the end are called **complete edges**. The edge

$$[0, 2,\ S \rightarrow NP\ \bullet\ VP]$$

says that an NP spans the string from 0 to 2 (the first two words) and that if we could find a VP to follow it, then we would have an S. Edges like this with the dot before the end are called incomplete edges, and we say that the edge is looking for a VP.

[3] It is because of the $\bullet$ that edges are sometimes called **dotted rules**.

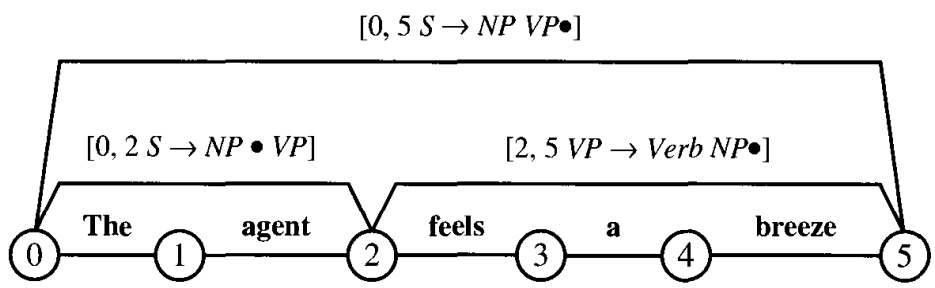


Figure 22.6 Part of the chart for the sentence "The agent feels a breeze." All six vertices are shown, but only three of the edges that would make up a complete parse.

```
function CHART-PARSE(words, grammar) returns chart

  chart ← array[0... LENGTH(words)] of empty lists
  ADD-EDGE([0, 0, S' → • S])
  for i ← from 0 to LENGTH(words) do
     SCANNER(i, words[i])
  return chart

procedure ADD-EDGE(edge)
   /* Add edge to chart, and see if it extends or predicts another edge. */
   if edge not in chart[END(edge)] then
      append edge to chart[END(edge)]
      if edge has nothing after the dot then EXTENDER(edge)
      else PREDICTOR(edge)

procedure SCANNER(j, word)
   /* For each edge expecting a word of this category here, extend the edge. */
   for each [i, j, A → α • B β] in chart[j] do
      if word is of category B then
         ADD-EDGE([i, j+1, A → α B • β])

procedure PREDICTOR([i, j, A → α • B β])
   /* Add to chart any rules for B that could help extend this edge */
   for each (B → γ) in REWRITES-FOR(B, grammar) do
      ADD-EDGE([j, j, B → • γ])

procedure EXTENDER([j, k, B → γ •])
   /* See what edges can be extended by this edge */
   e_B ← the edge that is the input to this procedure
   for each [i, j, A → α • B' β] in chart[j] do
      if B = B' then
         ADD-EDGE([i, k, A → α e_B • β])
```

Figure 22.7 The chart-parsing algorithm. S is the start symbol and S' is a new nonterminal symbol. $chart[j]$ is the list of edges that end at vertex j. The Greek letters match a string of zero or more symbols.

Figure 22.7 shows the chart-parsing algorithm. The main idea is to combine the best of top-down and bottom-up parsing. The procedure PREDICTOR is top-down: it makes entries into the chart that say what symbols are desired at what locations. SCANNER is the bottom-up procedure that starts from the words, but it will use a word only to extend an existing chart entry. Similarly, EXTENDER builds constituents bottom-up, but only to extend an existing chart entry.

We use a trick to start the whole algorithm: we add the edge $[0, 0, S' \rightarrow \bullet\ S]$ to the chart, where S is the grammar's start symbol, and S' is a new symbol that we just invented. The call to ADD-EDGE causes the PREDICTOR to add edges for the rules that can yield an S—that is, $[S \rightarrow NP\ VP]$. Then we look at the first constituent of that rule, NP, and add rules for every way to yield an NP. Eventually, the predictor adds, in a top-down fashion, all possible edges that could be used in the service of creating the final S.

When the predictor for S' is finished, we enter a loop that calls SCANNER for each word in the sentence. If the word at position j is a member of a category B that some edge is looking for at j, then we extend that edge, noting the word as an instance of B. Notice that each call to SCANNER can end up calling PREDICTOR and EXTENDER recursively, thereby interleaving the top-down and bottom-up processing.

The other bottom-up component, EXTENDER,[4] takes a complete edge with left hand side B and uses it to extend any incomplete rule in the chart that ends where the complete edge starts if the incomplete rule is looking for a B.

Figures 22.8 and 22.9 show a chart and trace of the algorithm parsing the sentence "I feel it" (which is an answer to the question "Do you feel a breeze?"). Thirteen edges (labeled a–m) are recorded in the chart, including five complete edges (shown above the vertices of the chart) and eight incomplete ones (below the vertices). Note the cycle of predictor, scanner, and extender actions. For example, the predictor uses the fact that edge (a) is looking for an S to license the prediction of an NP (edge b) and then a $Pronoun$ (edge c). Then the scanner recognizes that there is a $Pronoun$ in the right place (edge d), and the extender combines the incomplete edge b with the complete edge d to yield a new edge, e.

The chart-parsing algorithm avoids building a large class of edges that would have been examined by the simple bottom-up procedure. Consider the sentence "The ride the horse gave was wild." A bottom-up parse would label "ride the horse" as a VP and then discard the parse tree when it is found not to fit into a larger S. But $\mathcal{E}_0$ does not allow a VP to follow "the," so the chart-parsing algorithm will never predict a VP at that point and thus will avoid wasting time building the VP constituent there. Algorithms that work from left to right and avoid building these impossible constituents are called **left-corner** parsers, because they build up a parse tree that starts with the grammar's start symbol and extends down to the leftmost word in the sentence (the left corner). An edge is added to the chart only if it can serve to extend this parse tree. (See Figure 22.10 for an example.)

LEFT-CORNER

The chart parser uses only polynomial time and space. It requires $O(kn^2)$ space to store the edges, where n is the number of words in the sentence and k is a constant that depends

[4] Traditionally, our EXTENDER procedure has been called COMPLETER. This name is misleading, because the procedure does not complete edges: it takes a complete edge as input and extends incomplete edges.

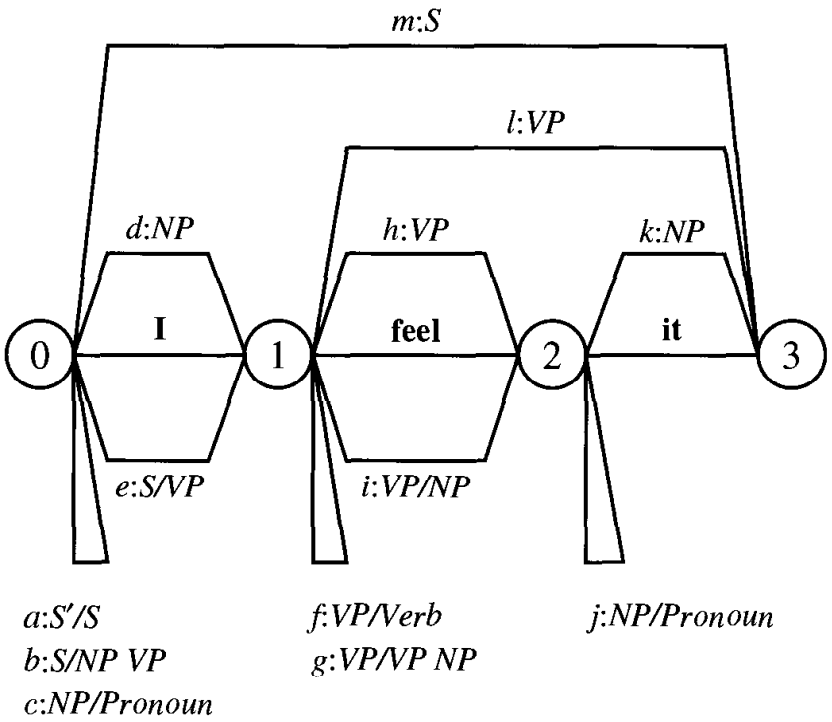


Figure 22.8 Chart for a parse of "$_0$ I $_1$ feel $_2$ it $_3$." The notation *m:S* means that edge m has an S on the left-hand side, while the notation *f:VP/Verb* means that edge f has a VP on the left-hand side, but is looking for a *Verb*. There are five complete edges above the vertices and eight incomplete edges below.

Edge	Procedure	Derivation
a	INITIALIZER	$[0,0,\ S' \rightarrow \bullet S]$
b	PREDICTOR(a)	$[0,0,\ S \rightarrow \bullet NP\ VP]$
c	PREDICTOR(b)	$[0,0,\ NP \rightarrow \bullet Pronoun]$
d	SCANNER(c)	$[0,1,\ NP \rightarrow Pronoun \bullet]$
e	EXTENDER(b,d)	$[0,1,\ S \rightarrow NP \bullet VP]$
f	PREDICTOR(e)	$[1,1,\ VP \rightarrow \bullet Verb]$
g	PREDICTOR(e)	$[1,1,\ VP \rightarrow \bullet VP\ NP]$
h	SCANNER(f)	$[1,2,\ VP \rightarrow Verb \bullet]$
i	EXTENDER(g,h)	$[1,2,\ VP \rightarrow VP \bullet NP]$
j	PREDICTOR(g)	$[2,2,\ NP \rightarrow \bullet Pronoun]$
k	SCANNER(j)	$[2,3,\ NP \rightarrow Pronoun \bullet]$
l	EXTENDER(i,k)	$[1,3,\ VP \rightarrow VP\ NP \bullet]$
m	EXTENDER(e,l)	$[0,3,\ S \rightarrow NP\ VP \bullet]$

Figure 22.9 Trace of a parse of "$_0$ I $_1$ feel $_2$ it $_3$." For each edge a-m, we show the procedure used to derive the edge from other edges already in the chart. Some edges were omitted for brevity.

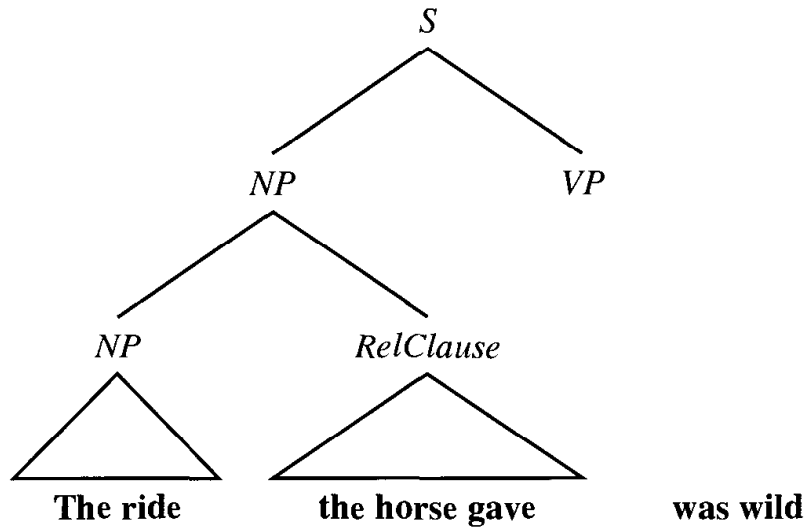


Figure 22.10 A left-corner parsing algorithm avoids predicting a *VP* starting with "ride," but does predict a *VP* starting with "was," because the grammar expects a *VP* following an *NP*. The triangle over "the horse gave" means that the words have a parse as a *RelClause*, but with additional intermediate constituents that are not shown.

on the grammar. When it can build no more edges it stops, so we know that the algorithm terminates (even when there are left-recursive rules). In fact, it takes time $O(n^3)$ in the worst case, which is the best that can be achieved for context-free grammars. The bottleneck for CHART-PARSE is EXTENDER, which must try to extend each of $O(n)$ incomplete edges ending at position j with each of $O(n)$ complete edges starting at j, for each of $n+1$ different values of j. Multiplying these together, we get $O(n^3)$. This gives us something of a paradox: how can an $O(n^3)$ algorithm return an answer that might contain an exponential number of parse trees? Consider an example: the sentence

"Fall leaves fall and spring leaves spring"

is ambiguous because each word (except "and") can be either a noun or a verb, and "fall" and "spring" can be adjectives as well. Altogether, the sentence has four parses:[5]

[*S*: [*S*: [*NP*: Fall leaves] fall] and [*S*: [*NP*: spring leaves] spring] ;
[*S*: [*S*: [*NP*: Fall leaves] fall] and [*S*: spring [*VP*: leaves spring]] ;
[*S*: [*S*: Fall [*VP*: leaves fall]] and [*S*: [*NP*: spring leaves] spring] ;
[*S*: [*S*: Fall [*VP*: leaves fall]] and [*S*: spring [*VP*: leaves spring]] .

If we had n ambiguous conjoined subsentences, we would have 2^n ways of choosing parses for the subsentences.[6] How does the chart parser avoid exponential processing time? There are actually two answers. First, the CHART-PARSE algorithm itself is actually a *recognizer*, not a parser. If there is a complete edge of the form $[0, n, \ S \ \rightarrow \ \alpha \ \bullet]$ in the chart, then we have recognized an S. Recovering the parse tree from this edge is not considered part of

[5] The parse [*S*: Fall [*VP*: leaves fall]] is equivalent to "Autumn abandons autumn."

[6] There also would be $O(n!)$ ambiguity in the way the components conjoin with each other—for example, (X and (Y and Z)) versus ((X and Y) and Z). But that is another story, one that is told quite well by Church and Patil (1982).

CHART-PARSE's job, but it can be done. Note, in the last line of EXTENDER, that we build up α as a list of edges, e_B, not just a list of category names. So to convert an edge into a parse tree, simply look recursively at the component edges, converting each $[i, j,\ X \rightarrow \alpha\ \bullet]$ into the tree $[X : \alpha]$. This is straightforward, but it gives us only *one* parse tree.

The second answer is that if you want all possible parses, you'll have to dig deeper into the chart. While we're converting the edge $[i, j,\ X \rightarrow \alpha\ \bullet]$ into the tree $[X : \alpha]$, we'll also look to see whether there are any other edges of the form $[i, j,\ X \rightarrow \beta\ \bullet]$. If there are, these edges will generate additional parses. Now we have a choice of what to do with them. We could enumerate all the possibilities, and that means that the paradox would be resolved and we would require an exponential amount of time to list the parses. Or we could prolong the mystery a little longer and represent the parses with a structure called a **packed forest**, which looks like this:

PACKED FOREST

$$[S\text{: } [S\text{: } \left\{ \begin{array}{l} [NP\text{: Fall leaves}]\ [VP\text{: fall}] \\ [NP\text{: Fall}]\ [VP\text{: leaves fall}] \end{array} \right\}] \text{ and } [S\text{: } \left\{ \begin{array}{l} [NP\text{: spring leaves}]\ [VP\text{: spring}] \\ [NP\text{: spring}]\ [VP\text{: leaves spring}] \end{array} \right\}]]$$

The idea is that each node can be either a regular parse tree node or a set of tree nodes. This enables us to return a representation of an exponential number of parses in a polynomial amount of space and time. Of course, when $n = 2$, there is not much difference between 2^n and $2n$, but for large n, such a representation offers considerable saving. Unfortunately, this simple packed forest approach won't handle all the $O(n!)$ ambiguity in how the conjunctions associate. Maxwell and Kaplan (1995) show how a more complex representation based on the principles of truth maintenance systems can pack the trees even tighter.

$$\begin{array}{rcl}
S & \rightarrow & NP_S\ VP \mid \ldots \\
NP_S & \rightarrow & Pronoun_S \mid Name \mid Noun \mid \ldots \\
NP_O & \rightarrow & Pronoun_O \mid Name \mid Noun \mid \ldots \\
VP & \rightarrow & VP\ NP_O \mid \ldots \\
PP & \rightarrow & Preposition\ NP_O \\
Pronoun_S & \rightarrow & \textbf{I} \mid \textbf{you} \mid \textbf{he} \mid \textbf{she} \mid \textbf{it} \mid \ldots \\
Pronoun_O & \rightarrow & \textbf{me} \mid \textbf{you} \mid \textbf{him} \mid \textbf{her} \mid \textbf{it} \mid \ldots
\end{array}$$

$$\begin{array}{rcl}
S & \rightarrow & NP(Subjective)\ VP \mid \ldots \\
NP(case) & \rightarrow & Pronoun(case) \mid Name \mid Noun \mid \ldots \\
VP & \rightarrow & VP\ NP(Objective) \mid \ldots \\
PP & \rightarrow & Preposition\ NP(Objective) \\
Pronoun(Subjective) & \rightarrow & \textbf{I} \mid \textbf{you} \mid \textbf{he} \mid \textbf{she} \mid \textbf{it} \mid \ldots \\
Pronoun(Objective) & \rightarrow & \textbf{me} \mid \textbf{you} \mid \textbf{him} \mid \textbf{her} \mid \textbf{it} \mid \ldots
\end{array}$$

Figure 22.11 Top: A BNF grammar for the language $\mathcal{E}_1$, which handles subjective and objective cases in noun phrases and thus does not over-generate quite so badly. The portions that are identical to $\mathcal{E}_0$ have been omitted. Bottom: A definite clause grammar (DCG) of $\mathcal{E}_1$.

22.4 AUGMENTED GRAMMARS

We saw in Section 22.2 that the simple grammar for $\mathcal{E}_0$ generates "I smell a stench" and many other sentences of English. Unfortunately, it also generates many non-sentences such as "Me smell a stench." To avoid this problem, our grammar would have to know that "me" is not a valid NP when it is the subject of a sentence. Linguists say that the pronoun "I" is in the subjective case, and "me" is in the objective case.[7] When we take case into account, we realize that the $\mathcal{E}_0$ grammar is not context-free: it is not true that any NP is equal to any other regardless of context. We can fix the problem by introducing new categories such as NP_S and NP_O, to stand for noun phrases in the subjective and objective case, respectively. We would also need to split the category $Pronoun$ into the two categories $Pronoun_S$ (which includes "I") and $Pronoun_O$ (which includes "me"). The top part of Figure 22.11 shows the complete BNF grammar for case agreement; we call the resulting language $\mathcal{E}_1$. Notice that all the NP rules must be duplicated, once for NP_S and once for NP_O.

AGREEMENT

Unfortunately, $\mathcal{E}_1$ still overgenerates. English and many other languages require **agreement** between the subject and main verb of a sentence. For example, if "I" is the subject, then "I smell" is grammatical, but "I smells" is not. If "it" is the subject, we get the reverse. In English, the agreement distinctions are minimal: most verbs have one form for third-person singular subjects (he, she, or it), and a second form for all other combinations of person and number. There is one exception: "I am / you are / he is" has three forms. If we multiply these three distinctions by the two distinctions of NP_S and NP_O, we end up with six forms of NP. As we discover more distinctions, we end up with an exponential number.

AUGMENT

The alternative is to **augment** the existing rules of the grammar instead of introducing new rules. We will first give an example of what we would like an augmented rule to look like (see the bottom half of Figure 22.11) and then formally define how to interpret the rules. Augmented rules allow for *parameters* on nonterminal categories. Figure 22.11 shows how to describe $\mathcal{E}_1$ using augmented rules. The categories NP and $Pronoun$ have a parameter indicating their case. (Nouns do not have case in English, although they do in many other languages.) In the rule for S, the NP must be in the subjective case, whereas in the rules for VP and PP, the NP must be in the objective case. The rule for NP takes a variable, *case*, as its argument. The intent is that the NP can have any case, but if the NP is rewritten as a $Pronoun$, then it must have the same case. This use of a variable—avoiding a decision where the distinction is not important—is what keeps the size of the rule set from growing exponentially with the number of features.

DEFINITE CLAUSE GRAMMAR

This formalism for augmentations is called **definite clause grammar** or DCG, because each grammar rule can be interpreted as a definite clause in Horn logic.[8] First we will show how a normal, unaugmented rule can be interpreted as a definite clause. We consider each

[7] The subjective case is also sometimes called the nominative case and the objective case is sometimes called the accusative case. Many languages also have a dative case for words in the indirect object position.

[8] Recall that a definite clause, when written as an implication, has exactly one atom in its consequent, and a conjunction of zero or more atoms in its antecedent. Two examples are $A \wedge B \Rightarrow C$ and just C.

category symbol to be a predicate on strings, so that $NP(s)$ is true if the string s forms an NP. The CFG rule

$$S \rightarrow NP\ VP$$

is shorthand for the definite clause

$$NP(s_1) \wedge VP(s_2) \Rightarrow S(s_1 + s_2)\ .$$

Here $s_1 + s_2$ denotes the concatenation of two strings, so this rule says that if the string s_1 is an NP and the string s_2 is a VP, then their concatenation is an S, which is exactly how we were already interpreting the CFG rule. It is important to note that *DCGs allow us to talk about parsing as logical inference.* This makes it possible to reason about languages and strings in many different ways. For example, it means we can do bottom-up parsing using forward chaining or top-down parsing using backward chaining. We will see that it also means that we can use the same grammar for both parsing and generation.

The real benefit of the DCG approach is that we can *augment* the category symbols with additional arguments other than the string argument. For example, the rule

$$NP(case) \rightarrow Pronoun(case)$$

is shorthand for the definite clause

$$Pronoun(case, s_1) \Rightarrow NP(case, s_1)\ .$$

This says that if the string s_1 is a $Pronoun$ with case specified by the variable *case*, then s_1 is also an NP with the same case. In general, we can augment a category symbol with any number of arguments, and the arguments are parameters that are subject to unification as in regular Horn clause inference.

There is a price to pay for this convenience: we are providing the grammar writer with the full power of a theorem-prover, so we give up the guarantees of $O(n^3)$ syntactic parsing; parsing with augmentations can be NP-complete or even undecidable, depending on the augmentations.

A few more tricks are necessary to make DCG work; for example, we need a way to specify terminal symbols, and it is convenient to have a way *not* to add the automatic string argument. Putting everything together, we define definite clause grammar as follows:

- The notation $X \rightarrow Y\ Z\ \ldots$ translates as $Y(s_1) \wedge Z(s_2) \wedge \ldots \Rightarrow X(s_1 + s_2 + \ldots)$.
- The notation $X \rightarrow Y \mid Z \mid \ldots$ translates as $Y(s) \vee Z(s) \vee \ldots \Rightarrow X(s)$.
- In either of the preceding rules, any nonterminal symbol Y can be augmented with one or more arguments. Each argument can be a variable, a constant, or a function of arguments. In the translation, these arguments precede the string argument (e.g., $NP(case)$ translates as $NP(case, s_1)$).
- The notation $\{P(\ldots)\}$ can appear on the right-hand side of a rule and translates verbatim into $P(\ldots)$. This allows the grammar writer to insert a test for $P(\ldots)$ without having the automatic string argument added.
- The notation $X \rightarrow$ **word** translates as $X([word])$.

The problem of subject–verb agreement could also be handled with augmentations, but we defer that to Exercise 22.2. Instead, we address a harder problem: verb subcategorization.

Verb	Subcats	Example Verb Phrase
give	$[NP, PP]$ $[NP, NP]$	give the gold in 3 3 to me give me the gold
smell	$[NP]$ $[Adjective]$ $[PP]$	smell a wumpus smell awful smell like a wumpus
is	$[Adjective]$ $[PP]$ $[NP]$	is smelly is in 2 2 is a pit
died	$[]$	died
believe	$[S]$	believe the wumpus is dead

Figure 22.12 Examples of verbs with their subcategorization lists.

Verb subcategorization

$\mathcal{E}_1$ is an improvement over $\mathcal{E}_0$, but the $\mathcal{E}_1$ grammar still overgenerates. One problem is in the way verb phrases are put together. We want to accept verb phrases like "give me the gold" and "go to 1 2." All these are in $\mathcal{E}_1$, but unfortunately so are "go me the gold" and "give to 1 2." The language $\mathcal{E}_2$ eliminates these VPs by stating explicitly which phrases can follow which verbs. We call this list the **subcategorization** list for the verb. The idea is that the category $Verb$ is broken into subcategories—one for verbs that have no object, one for verbs that take a single object, and so on.

SUBCATEGORIZATION

SUBCATEGORIZATION LIST

COMPLEMENTS

To implement this idea, we give each verb a **subcategorization list** that lists the verb's **complements**. A complement is an obligatory phrase that follows the verb within the verb phrase. So in "Give the gold to me," the NP "the gold" and the PP "to me" are complements of "give."[9] We would write this as

$$Verb([NP, PP]) \rightarrow \textbf{give} \mid \textbf{hand} \mid \ldots$$

It is possible for a verb to have several different subcategorizations, just as it is possible for a word to belong to several different categories. In fact, "give" also has the subcategorization list $[NP, NP]$, as in "Give me the gold." We can treat this like any other kind of ambiguity. Figure 22.12 gives some examples of verbs and their subcategorization lists (or **subcats** for short).

To integrate verb subcategorization into the grammar, we take three steps. The first step is to augment the category VP to take a subcategorization argument, $VP(subcat)$, that indicates the list of complements that are needed to form a complete VP. For example, "give" can be made into a complete VP by adding $[NP, PP]$, "give the gold" can be made complete by adding $[PP]$, and "give the gold to me" is already a complete VP; therefore its

[9] This is one definition of *complement*, but other authors have different terminology. Some say that the subject of the verb is also a complement. Others say that only the prepositional phrase is a complement and that the noun phrase should be called an **argument**.

subcategorization list is the empty list, [ ]. That gives us these rules for VP:

$$\begin{array}{lcl} VP(subcat) & \rightarrow & Verb(subcat) \\ & | & VP(subcat + [NP])\ NP(Objective) \\ & | & VP(subcat + [Adjective])\ Adjective \\ & | & VP(subcat + [PP])\ PP\ . \end{array}$$

The last line can be read as "A VP with a given subcat list, *subcat*, can be formed by an embedded VP followed by a PP, as long as the embedded VP has a subcat list that starts with the elements of the list *subcat* and ends with the symbol PP." For example, a $VP([\,])$ is formed by a $VP([PP])$ followed by a PP. The first line says that a VP with subcategorization list *subcat* can be formed by a *Verb* with the same subcategorization list. For example, a $VP([NP])$ can be formed by a $Verb([NP])$. One example of such a verb is "grab," so "grab the gold" is a $VP([\,])$.

The second step is to change the rule for S to say that it requires a verb phrase that has all its complements and thus has the subcat list [ ]. This means that "I grab the gold" is a legal sentence, but "You give" is not. The new rule,

$$S \rightarrow NP(Subjective)\ VP([\,])\ ,$$

can be read as "A sentence can be composed of a NP in the subjective case, followed by a VP that has a null subcat list." Figure 22.13 shows a parse tree using this grammar.

ADJUNCTS

The third step is to remember that, in addition to complements, verb phrases (and other phrases) can also take **adjuncts**, which are phrases that are not licensed by the individual verb but rather may appear in any verb phrase. Phrases representing time and place are adjuncts, because almost any action or event can have a time or place. For example, the adverb "now" in "I smell a wumpus now" and the PP "on Tuesday" in "give me the gold on Tuesday" are adjuncts. Here are two rules that allow propositional and adverbial adjuncts on any VP:

$$\begin{array}{lcl} VP(subcat) & \rightarrow & VP(subcat)\ PP \\ & | & VP(subcat)\ Adverb\ . \end{array}$$

Generative capacity of augmented grammars

RULE SCHEMA

Each augmented rule is a **rule schema**, that stands for a set of rules, one for each possible combination of values for the augmented constituents. The generative capacity of augmented grammars depends on the number of combinations. If there is a finite number, then the augmented grammar is equivalent to a context-free grammar: the rule schema could be replaced with individual context-free rules. But if there are an infinite number of values, then augmented grammars can represent non-context-free languages. For example, the context-sensitive language $a^n b^n c^n$ can be represented as:

$$S(n) \rightarrow A(n)\ B(n)\ C(n)$$

$$\begin{array}{lcl@{\qquad\qquad}lcl} A(1) & \rightarrow & \mathbf{a} & A(n+1) & \rightarrow & \mathbf{a}\ A(n) \\ B(1) & \rightarrow & \mathbf{b} & B(n+1) & \rightarrow & \mathbf{b}\ B(n) \\ C(1) & \rightarrow & \mathbf{c} & C(n+1) & \rightarrow & \mathbf{c}\ C(n) \end{array}$$

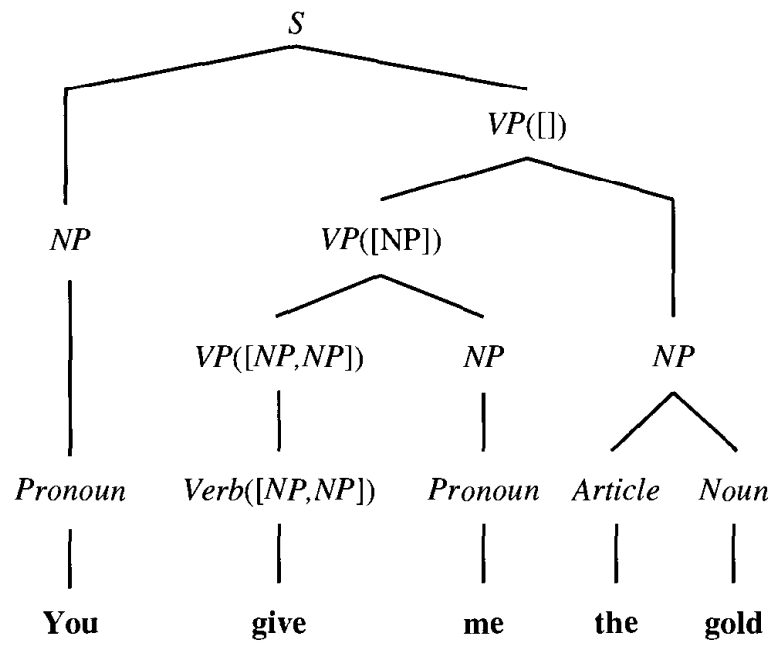


Figure 22.13 Parse tree for "You give me the gold" showing subcategorization of the verb and verb phrase.

22.5 SEMANTIC INTERPRETATION

So far, we have only looked at the syntactic analysis of language. In this section, we turn to the **semantics**—the extraction of the *meaning* of utterances. For this chapter we are using first-order logic as our representation language, so semantic interpretation is the process of associating an FOL expression with a phrase. Intuitively, the meaning of the phrase "the wumpus" is the big, hairy beast that we represent in logic as the logical term $Wumpus_1$, and the meaning of "the wumpus is dead" is the logical sentence $Dead(Wumpus_1)$. This section will make that intuition more precise. We'll start with a simple example: a rule for describing grid locations:

$$NP \rightarrow Digit\ Digit\ .$$

We will augment the rule by adding to each constituent an argument representing the semantics of the constituent. We get

$$NP([x, y]) \rightarrow Digit(x)\ Digit(y)\ .$$

This says that a string consisting of a digit with semantics x followed by another digit with semantics y forms an NP with semantics $[x, y]$, which is our notation for a square in the grid.

COMPOSITIONAL SEMANTICS

Notice that the semantics of the whole NP is composed largely of the semantics of the constituent parts. We have seen this idea of **compositional semantics** before: in logic, the meaning of $P \wedge Q$ is determined by the meaning of P, Q, and $\wedge$; in arithmetic, the meaning of $x+y$ is determined by the meaning of x, y, and $+$. Figure 22.14 shows how DCG notation can be used to augment a grammar for arithmetic expressions with semantics and Figure 22.15

$$
\begin{aligned}
Exp(x) &\rightarrow Exp(x_1)\ Operator(op)\ Exp(x_2)\ \{x = Apply(op, x_1, x_2)\} \\
Exp(x) &\rightarrow (\ Exp(x)\ ) \\
Exp(x) &\rightarrow Number(x) \\
Number(x) &\rightarrow Digit(x) \\
Number(x) &\rightarrow Number(x_1)\ Digit(x_2)\ \{x = 10 \times x_1 + x_2\} \\
Digit(x) &\rightarrow x\ \{0 \leq x \leq 9\} \\
Operator(x) &\rightarrow x\ \{x \in \{+, -, \div, \times\}\}
\end{aligned}
$$

Figure 22.14 A grammar for arithmetic expressions, augmented with semantics. Each variable x_i represents the semantics of a constituent. Note the use of the $\{test\}$ notation to define logical predicates that must be satisfied, but that are not constituents.

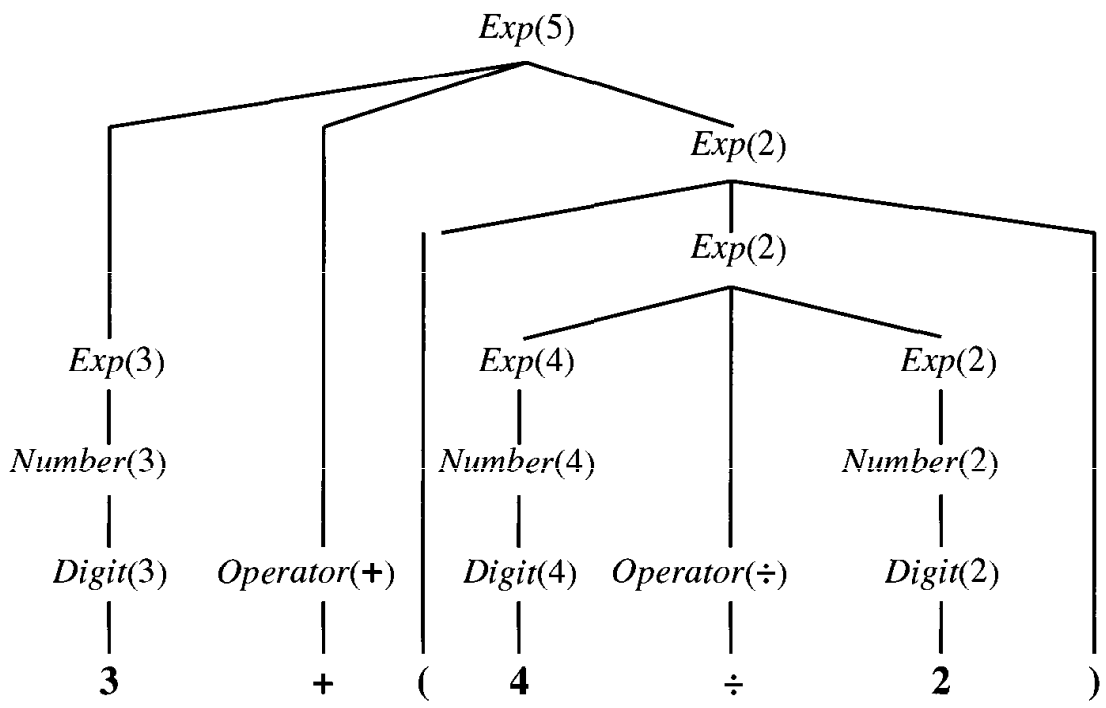


Figure 22.15 Parse tree with semantic interpretations for the string "$3 + (4 \div 2)$".

shows the parse tree for $3 + (4 \div 2)$ according to this grammar. The root of the parse tree is $Exp(5)$, an expression whose semantic interpretation is 5.

The semantics of an English fragment

We are now ready to write the semantic augmentations for a fragment of English. We start by determining what semantic representations we want to associate with what phrases. We will use the simple example sentence "John loves Mary." The *NP* "John" should have as its semantic interpretation the logical term $John$, and the sentence as a whole should have as its interpretation the logical sentence $Loves(John, Mary)$. That much seems clear. The complicated part is the *VP* "loves Mary." The semantic interpretation of this phrase is neither a logical term nor a complete logical sentence. Intuitively, "loves Mary" is a description that

might or might not apply to a particular person. (In this case, it applies to John.) This means that "loves Mary" is a **predicate** that, when combined with a term that represents a person (the person doing the loving), yields a complete logical sentence. Using the λ-notation (see page 248), we can represent "loves Mary" as the predicate

$$\lambda x\ Loves(x, Mary)\ .$$

Now we need a rule that says "an NP with semantics obj followed by a VP with semantics rel yields a sentence whose semantics is the result of applying rel to obj:"

$$S(rel(obj)) \rightarrow NP(obj)\ VP(rel)\ .$$

The rule tells us that the semantic interpretation of "John loves Mary" is

$$(\lambda x\ Loves(x, Mary))(John)\ ,$$

which is equivalent to $Loves(John, Mary)$.

The rest of the semantics follows in a straightforward way from the choices we have made so far. Because VPs are represented as predicates, it is a good idea to be consistent and represent verbs as predicates as well. The verb "loves" is represented as $\lambda y\ \lambda x\ Loves(x, y)$, the predicate that, when given the argument $Mary$, returns the predicate $\lambda x\ Loves(x, Mary)$.

The $VP \rightarrow Verb\ NP$ rule applies the predicate that is the semantic interpretation of the verb to the object that is the semantic interpretation of the NP to get the semantic interpretation of the whole VP. We end up with the grammar shown in Figure 22.16 and the parse tree shown in Figure 22.17.

$$S(rel(obj)) \rightarrow NP(obj)\ VP(rel)$$
$$VP(rel(obj)) \rightarrow Verb(rel)\ NP(obj)$$
$$NP(obj) \rightarrow Name(obj)$$

$$Name(John) \rightarrow \textbf{John}$$
$$Name(Mary) \rightarrow \textbf{Mary}$$
$$Verb(\lambda x\ \lambda y\ Loves(x, y)) \rightarrow \textbf{loves}$$

Figure 22.16 A grammar that can derive a parse tree and semantic interpretation for "John loves Mary" (and three other sentences). Each category is augmented with a single argument representing the semantics.

Time and tense

Now suppose we want to represent the difference between "John loves Mary" and "John loved Mary." English uses verb tenses (past, present, and future) to indicate the relative time of an event. One good choice to represent the time of events is the event calculus notation of Section 10.3. In event calculus, our two sentences have the following interpretations:

$$e \in Loves(John, Mary) \wedge During(Now, e)\ ;$$
$$e \in Loves(John, Mary) \wedge After(Now, e)\ .$$

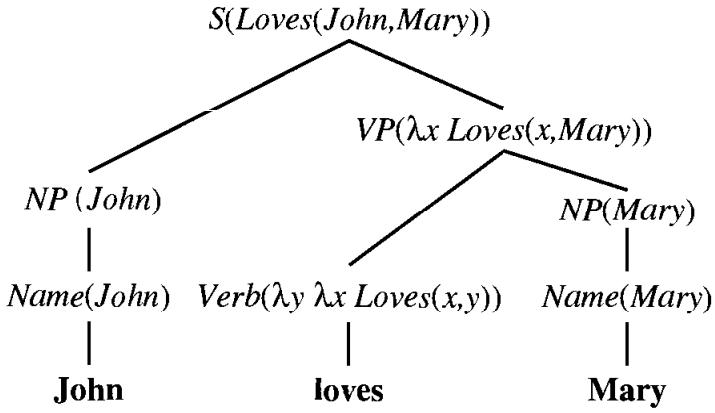


Figure 22.17 A parse tree with semantic interpretations for the string "John loves Mary".

This suggests that our two lexical rules for the words "loves" and "loved" should be these:

$$Verb(\lambda x\ \lambda y\ e \in Loves(John, Mary) \wedge During(Now, e)) \rightarrow \textbf{loves}\ ;$$
$$Verb(\lambda x\ \lambda y\ e \in Loves(x, y) \wedge After(Now, e)) \rightarrow \textbf{loved}\ .$$

Other than this change, everything else about the grammar remains the same, which is encouraging news; it suggests we are on the right track if we can so easily add a complication like the tense of verbs (although we have just scratched the surface of a complete grammar for time and tense). With this success as a warm-up, we are now ready to tackle a much harder representation problem.

Quantification

Consider the sentence "Every agent smells a wumpus." The sentence is actually ambiguous; the preferred meaning is that the agents might be smelling different wumpuses, but an alternative meaning is that there is a single wumpus that everyone smells.[10] The two interpretations can be represented as follows:

$$\begin{aligned}
&\forall a\ \ a \in Agents\ \Rightarrow \\
&\qquad \exists w\ \ w \in Wumpuses \wedge \exists e\ \ e \in Smell(a, w) \wedge During(Now, e)\ ; \\
&\exists w\ \ w \in Wumpuses\ \forall a\ \ a \in Agents\ \Rightarrow \\
&\qquad \exists e\ \ e \in Smell(a, w) \wedge During(Now, e)\ .
\end{aligned}$$

We will defer the problem of ambiguity and for now look only at the first interpretation. We'll try to analyze it compositionally, breaking it into the NP and VP components:

$$\begin{array}{ll}
\text{Every agent} & NP(\forall a\ \ a \in Agents\ \Rightarrow\ P) \\
\text{smells a wumpus} & VP(\exists w\ \ w \in Wumpuses\ \wedge \\
& \qquad \exists e\ \ (e \in Smell(a, w) \wedge During(Now, e))\ .
\end{array}$$

Right away, there are two difficulties. First, the semantics of the entire sentence appears to be the semantics of the NP, with the semantics of the VP filling in the P part. That means that we cannot form the semantics of the sentence with $rel(obj)$. We could do it with $obj(rel)$, which seems a little odd (at least at first glance). The second problem is that we need to get

[10] If this interpretation seems unlikely, consider "Every Protestant believes in a just God."

the variable a as an argument of the relation *Smell*. In other words, the semantics of the sentence is formed by plugging the semantics of the *VP* into the correct argument slot of the *NP*, while also plugging the variable a from the *NP* into the correct argument slot of the semantics of the *VP*. It looks as if we need two functional compositions and promises to be rather confusing. The complexity stems from the fact that the semantic structure is very different from the syntactic structure.

INTERMEDIATE FORM

To avoid this confusion, many modern grammars take a different tack. They define an **intermediate form** to mediate between syntax and semantics. The intermediate form has two key properties. First, it is structurally similar to the syntax of the sentence and thus can be easily constructed through compositional means. Second, it contains enough information that it can be translated into a regular first-order logical sentence. Because it sits between the syntactic and logical forms, it is called a **quasi-logical form**.[11] In this section, we will use a quasi-logical form that includes all of first-order logic and is augmented by lambda expressions and one new construction, which we will call a **quantified term**. The quantified term that is the semantic interpretation of "every agent" is written

QUASI-LOGICAL FORM

QUANTIFIED TERM

$$[\forall a\ a \in Agents]\ .$$

This looks like a logical sentence, but it is used in the same way that a logical term is used. The interpretation of "Every agent smells a wumpus" in quasi-logical form is

$$\exists e\ \ (e \in Smell([\forall a\ a \in Agents], [\exists w\ w \in Wumpuses]) \wedge During(Now, e))\ .$$

To generate quasi-logical form, many of our rules remain unchanged. The rule for S still creates the semantics of the S with $rel(obj)$. Some rules do change; the lexical rule for "a" is

$$Article(\exists)\ \rightarrow\ \mathbf{a}$$

and the rule for combining an article with a noun is

$$NP([q\,x\ sem(x)])\ \rightarrow\ Article(q)\ Noun(sem)\ .$$

This says that the semantics of the *NP* is a quantified term, with a quantifier specified by the article, with a new variable x, and with a proposition formed by applying the semantics of the noun to the variable x. The other rules for *NP* are similar. Figure 22.18 shows the semantic types and example forms for each syntactic category under the quasi-logical form approach. Figure 22.19 shows the parse of "every agent smells a wumpus" using this approach, and Figure 22.20 shows the complete grammar.

Now we need to convert the quasi-logical form into real first-order logic by turning quantified terms into real terms. This is done by a simple rule: For each quantified term $[q\,x\ P(x)]$ within a quasi-logical form *QLF*, replace the quantified term with x, and replace *QLF* with $q\,x\ P(x)\ op\ QLF$, where op is $\Rightarrow$ when q is $\forall$ and is $\wedge$ when q is $\exists$ or $\exists!$. For example, the sentence "Every dog has a day" has the quasi-logical form:

$$\exists e\ \ (e \in Has([\forall d\ d \in Dogs], [\exists a\ a \in Days], Now))\ .$$

[11] Some quasi-logical forms have the third property that they can succinctly represent ambiguities that could be represented in logical form only by a long disjunction.

Category	Semantic Type	Example	Quasi-Logical Form
S	*sentence*	I sleep.	$\exists e \;\; e \in Sleep(Speaker) \wedge During(Now, e)$
NP	*object*	a dog	$[\exists d \; Dog(d)]$
PP	$object^2 \rightarrow sentence$	in [2,2]	$\lambda x \; In(x, [2,2])$
$RelClause$	$object \rightarrow sentence$	that sees me	$\lambda x \; \exists e \;\; e \in Sees(x, Speaker) \wedge During(Now, e)$
VP	$object^n \rightarrow sentence$	sees me	$\lambda x \; \exists e \;\; e \in Sees(x, Speaker) \wedge During(Now, e)$
$Adjective$	$object \rightarrow sentence$	smelly	$\lambda x \; Smelly(x)$
$Adverb$	$event \rightarrow sentence$	today	$\lambda e \; During(e, Today)$
$Article$	*quantifier*	the	$\exists!$
$Conjunction$	$sentence^2 \rightarrow sentence$	and	$\lambda p, q \; (p \wedge q)$
$Digit$	*object*	7	7
$Noun$	$object \rightarrow sentence$	wumpus	$\lambda x \; x \in Wumpuses$
$Preposition$	$object^2 \rightarrow sentence$	in	$\lambda x \; \lambda y \; In(x, y)$
$Pronoun$	*object*	I	$Speaker$
$Verb$	$object^n \rightarrow sentence$	eats	$\lambda y \; \lambda x \; \exists e \;\; e \in Eats(x, y) \wedge During(Now, e)$

Figure 22.18 Table showing the type of quasi-logical form expression for each syntactic category. The notation $t \rightarrow r$ denotes a function that takes an argument of type t and returns a result of type r. For example, the semantic type for *Preposition* is $object^2 \rightarrow sentence$, which means that the semantics of a preposition is a function that, when applied to two logical objects, will yield a logical sentence.

We did not specify which of the two quantified terms gets pulled out first, so there are actually two possible logical interpretations:

$$\forall d \;\; d \in Dogs \;\Rightarrow\; \exists a \;\; a \in Days \wedge \exists e \;\; e \in Has(d, a, Now) \;;$$
$$\exists a \;\; a \in Days \wedge \forall d \;\; d \in Dogs \;\Rightarrow\; \exists e \;\; e \in Has(d, a, Now) \;.$$

The first one says that each dog has his own day, while the second says that there is a special day that all dogs share. Choosing between them is a job for disambiguation. Often, the left-to-right order of the quantified terms matches the left-to-right order of the quantifiers, but other factors come into play. The advantage of quasi-logical form is that it succinctly represents all the possibilities. The disadvantage is that it doesn't help you choose between them; for that we need the full power of disambiguation using all sources of evidence.

Pragmatic Interpretation

We have shown how an agent can perceive a string of words and use a grammar to derive a set of possible semantic interpretations. Now we address the problem of completing the interpretation by adding context-dependent information about the current situation to each candidate interpretation.

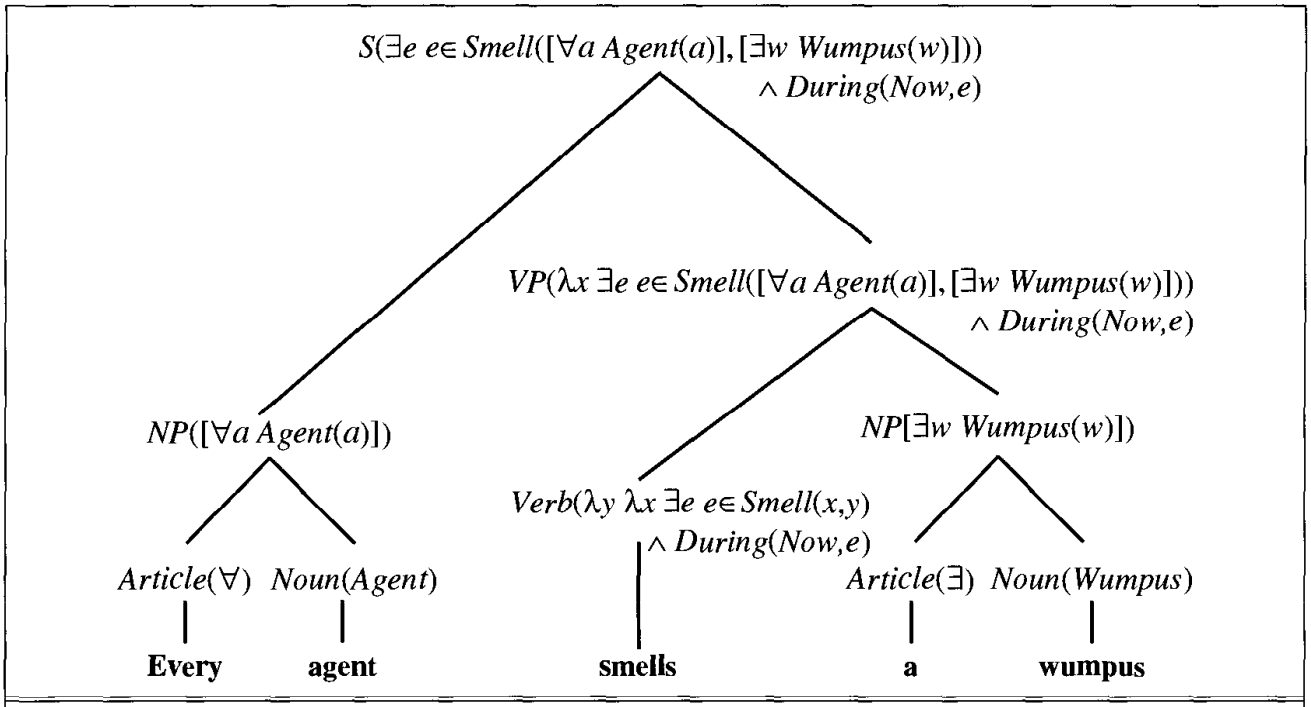


Figure 22.19 Parse tree for the sentence "Every agent smells a wumpus," showing both syntactic structure and semantic interpretations.

$S(rel(obj)) \rightarrow NP(obj)\ VP(rel)$
$S(conj(sem_1, sem_2)) \rightarrow S(sem_1)\ Conjunction(conj)\ S(sem_2)$

$NP(sem) \rightarrow Pronoun(sem)$
$NP(sem) \rightarrow Name(sem)$
$NP([q\,x\ sem(x)]) \rightarrow Article(q)\ Noun(sem)$
$NP([q\,x\ obj \wedge rel(x)]) \rightarrow NP([q\,x\ obj])\ PP(rel)$
$NP([q\,x\ obj \wedge rel(x)]) \rightarrow NP([q\,x\ obj])\ RelClause(rel)$
$NP([sem_1, sem_2]) \rightarrow Digit(sem_1)\ Digit(sem_2)$

$VP(sem) \rightarrow Verb(sem)$
$VP(rel(obj)) \rightarrow VP(rel)\ NP(obj)$
$VP(sem_1(sem_2) \rightarrow VP(sem_1)\ Adjective(sem_2)$
$VP(sem_1(sem_2)) \rightarrow VP(sem_1)\ PP(sem_2)$

$RelClause(sem) \rightarrow$ **that** $VP(sem)$

$PP(\lambda x\ rel(x, obj)) \rightarrow Preposition(rel)\ NP(obj)$

Figure 22.20 A grammar with semantics in quasi-logical form.

INDEXICALS

The most obvious need for pragmatic information is in resolving the meaning of **indexicals**, which are phrases that refer directly to the current situation. For example, in the sentence "I am in Boston today," the interpretations of the indexicals "I" and "today" depend on who uttered the sentence when. We represent indexicals by "constants" (such as *Speaker*) which actually are **fluents**—that is, they depend on the situation. The hearer who perceives a speech act should also perceive who the speaker is and use this information to resolve the indexical. For example, the hearer might know $T((Speaker = Agent_B), Now)$.

A command such as "go to 2 2" implicitly refers to the hearer. So far, our grammar for S covers only declarative sentences. We can easily extend it to cover commands.[12]

A command can be formed from a VP, where the subject is implicitly the hearer. We need to distinguish commands from statements, so we alter the rules for S to include the type of speech act as part of the quasi-logical form:

$$S(Statement(Speaker, rel(obj))) \rightarrow NP(obj)\ VP(rel)$$
$$S(Command(Speaker, rel(Hearer))) \rightarrow VP(rel)\ .$$

So the quasi-logical form for "Go to 2 2" is[13]

$$Command(\exists e\ \ e \in Go(Hearer, [2, 2]))\ .$$

Language generation with DCGs

So far, we have concentrated on *parsing* language, not on generating it. Generation is a topic of similar richness. Choosing the right utterance to express a proposition involves many of the same choices that parsing the utterance does.

Remember that a DCG is a logical programming system that specifies constraints between a string and the parse of a string. We know that a logic programming definition of the $Append$ predicate can be used both to tell us that in $Append([1, 2], [3], x)$ we have $x = [1, 2, 3]$ and to enumerate the values of x and y that make $Append(x, y, [1, 2, 3])$ true. In the same way, we can write a definition of S that can be used in two ways: to parse, we ask $S(sem, [John, Loves, Mary])$ and get back $sem = Loves(John, Mary)$; to generate, we ask $S(Loves(John, Mary), words)$ and get back $words = [John, Loves, Mary]$. We can also test a grammar by asking $S(sem, words)$ and getting back as an answer a stream of $[sem, words]$ pairs that are generated by the grammar.

This approach works for the simple grammars in this chapter, but there can be difficulties in scaling up to larger grammars. The search strategy used by the logical inference engine is important; depth-first strategies can lead to infinite loops. Some care must be taken in the

[12] To implement a complete communicating agent we would also need a grammar of questions. Questions are beyond the scope of this book because they impose **long-distance dependencies** between constituents. For example, in "Whom did the agent tell you to give the gold to?" the final word "to" should be parsed as a PP with a missing NP; the missing NP is licensed by the first word of the sentence, "who." A complex system of augmentations is used to make sure that the missing NPs match up with the licensing words.

[13] Note that the quasi-logical form for a command does not include the time of the event (e.g., $During(Now, e)$). That is because the "go" is actually the untensed version of the word, not the present tense version. You can't tell the difference with "go," but observe that the correct form of a command is "Be good!" (using the untensed form "be"), not "Are good!" To ensure the correct tense is used, we could augment VPs with a tense argument and write $VP(rel, untensed)$ on the right-hand side of the command rule.

exact details of the semantic form. It could be that a given grammar has no way to express the logical form $X \wedge Y$ for some values of X and Y, but can express $Y \wedge X$; this suggests that we need a way to canonicalize semantic forms, or we need to extend the unification routine so that $X \wedge Y$ can unify with $Y \wedge X$.

Serious work in generation tends to use more complex generation models that are distinct from the parsing grammar and offer more control over exactly how components of the semantics are expressed. Systemic grammar is one approach that makes it easy to put emphasis on the most important parts of the semantic form.

22.6 Ambiguity and Disambiguation

In some cases, hearers are consciously aware of ambiguity in an utterance. Here are some examples taken from newspaper headlines:

Squad helps dog bite victim.
Helicopter powered by human flies.
Once-sagging cloth diaper industry saved by full dumps.
Portable toilet bombed; police have nothing to go on.
British left waffles on Falkland Islands.
Teacher strikes idle kids.
Milk drinkers are turning to powder.
Drunk gets nine months in violin case.

But most of the time the language we hear seems unambiguous. Thus, when researchers first began to use computers to analyze language in the 1960s they were quite surprised to learn that *almost every utterance is highly ambiguous, even though the alternative interpretations might not be apparent to a native speaker.* A system with a large grammar and lexicon might find thousands of interpretations for a perfectly ordinary sentence. Consider "The batter hit the ball," which seems to have an unambiguous interpretation in which a baseball player strikes a baseball. But we get a different interpretation if the previous sentence is "The mad scientist unleashed a tidal wave of cake mix towards the ballroom." This example relies on **lexical ambiguity**, in which a word has more than one meaning. Lexical ambiguity is quite common; "back" can be an adverb (go back), an adjective (back door), a noun (the back of the room) or a verb (back up your files). "Jack" can be a name, a noun (a playing card, a six-pointed metal game piece, a nautical flag, a fish, a male donkey, a socket, or a device for raising heavy objects), or a verb (to jack up a car, to hunt with a light, or to hit a baseball hard).

LEXICAL AMBIGUITY

SYNTACTIC AMBIGUITY

Syntactic ambiguity (also known as **structural ambiguity**) can occur with or without lexical ambiguity. For example, the string "I smelled a wumpus in 2,2" has two parses: one where the prepositional phrase "in 2,2" modifies the noun and one where it modifies the verb. The syntactic ambiguity leads to a **semantic ambiguity**, because one parse means that the wumpus is in 2,2 and the other means that a stench is in 2,2. In this case, getting the wrong interpretation could be a deadly mistake.

SEMANTIC AMBIGUITY

Semantic ambiguity can occur even in phrases with no lexical or syntactic ambiguity. For example, the noun phrase "cat person" can be someone who likes felines or the lead of the movie *Attack of the Cat People.* A "coast road" can be a road that follows the coast or one that leads to it.

Finally, there can be ambiguity between literal and figurative meanings. Figures of speech are important in poetry, but are surprisingly common in everyday speech as well. A

METONYMY

metonymy is a figure of speech in which one object is used to stand for another. When we hear "Chrysler announced a new model," we do not interpret it as saying that companies can talk; rather we understand that a spokesperson representing the company made the announcement. Metonymy is common and is often interpreted unconsciously by human hearers. Unfortunately, our grammar as it is written is not so facile. To handle the semantics of metonymy properly, we need to introduce a whole new level of ambiguity. We do this by providing *two* objects for the semantic interpretation of every phrase in the sentence: one for the object that the phrase literally refers to (Chrysler) and one for the metonymic reference (the spokesperson). We then have to say that there is a relation between the two. In our current grammar, "Chrysler announced" gets interpreted as

$$\exists\, x, e \;\; x = Chrysler \wedge e \in Announce(x) \wedge After(Now, e)\ .$$

We need to change that to

$$\exists\, m, x, e \;\; x = Chrysler \wedge e \in Announce(m) \wedge After(Now, e) \\ \wedge\, Metonymy(m, x)\ .$$

This says that there is one entity x that is equal to Chrysler, and another entity m that did the announcing, and that the two are in a metonymy relation. The next step is to define what kinds of metonymy relations can occur. The simplest case is when there is no metonymy at all—the literal object x and the metonymic object m are identical:

$$\forall\, m, x \;\; (m = x) \;\Rightarrow\; Metonymy(m, x)\ .$$

For the Chrysler example, a reasonable generalization is that an organization can be used to stand for a spokesperson of that organization:

$$\forall\, m, x \;\; x \in Organizations \wedge Spokesperson(m, x) \;\Rightarrow\; Metonymy(m, x)\ .$$

Other metonymies include the author for the works (I read *Shakespeare*) or more generally the producer for the product (I drive a *Honda*) and the part for the whole (The Red Sox need a strong *arm*). Some examples of metonymy, such as "The *ham sandwich* on Table 4 wants another beer," are more novel and are interpreted with respect to a situation.

The rules we have outlined here allow us to construct an explanation for "Chrysler announced a new model," but the explanation doesn't follow by logical deduction. We need to use probabilistic or nonmonotonic reasoning to come up with candidate explanations.

METAPHOR

A **metaphor** is a figure of speech in which a phrase with one literal meaning is used to suggest a different meaning by way of an analogy. Most people think of metaphor as a tool used by poets that does not play a large role in everyday text. However, a number of basic metaphors are so common that we do not even recognize them as such. One such metaphor is the idea that *more is up*. This metaphor allows us to say that prices have risen, climbed, or

skyrocketed, that the temperature has dipped or fallen, that one's confidence has plummeted, or that a celebrity's popularity has jumped or soared.

There are two ways to approach metaphors like this. One is to compile all knowledge of the metaphor into the lexicon—to add new senses of the words "rise," "fall," "climb," and so on, that describe them as dealing with quantities on any scale rather than just altitude. This approach suffices for many applications, but it does not capture the generative character of the metaphor that allows humans to use new instances such as "nosedive" or "blasting through the roof" without fear of misunderstanding. The second approach is to include explicit knowledge of common metaphors and use them to interpret new uses as they are read. For example, suppose the system knows the "more is up" metaphor. That is, it knows that logical expressions that refer to a point on a vertical scale can be interpreted as being about corresponding points on a quantity scale. Then the expression "sales are high" would get a literal interpretation along the lines of $Altitude(Sales, High)$, which could be interpreted metaphorically as $Quantity(Sales, Much)$.

Disambiguation

As we said before, *disambiguation is a question of diagnosis.* The speaker's intent to communicate is an unobserved cause of the words in the utterance, and the hearer's job is to work backwards from the words and from knowledge of the situation to recover the most likely intent of the speaker. In other words, the hearer is solving for

$$\underset{intent}{\operatorname{argmax}}\, Likelihood(intent|words, situation)\ ,$$

where *Likelihood* can either be probability or any numeric measure of preference. Some sort of preference is needed because syntactic and semantic interpretation rules alone cannot identify a unique correct interpretation of a phrase or sentence. So we divide the work: syntactic and semantic interpretation is responsible for enumerating a set of candidate interpretations, and the disambiguation process chooses the best one.

Note that we talk about the intent of the speech act, not just the actual proposition that the speaker is proclaiming. For example, after hearing a politician say, "I am not a crook," we might assign a probability of only 50% to the proposition that the politician is not a criminal, and 99.999% to the proposition that the speaker is not a hooked shepherd's staff. Still, we assign a higher probability to the interpretation

$$Assert(Speaker, \neg(Speaker \in Criminals))$$

because this is a more likely thing to say.

RIGHT ASSOCIATION

Consider again the ambiguous example "I smelled a wumpus in 2,2." One preference heuristic is the rule of **right association**, which says that when it is time to decide where in the parse tree to place the *PP* "in 2,2," we should prefer to attach it to the rightmost existing constituent, which in this case is the *NP* "a wumpus." Of course, this is only a heuristic; for the sentence "I smelled a wumpus with my nose," the heuristic would be outweighed by the fact that the *NP* "a wumpus with my nose" is unlikely.

Disambiguation is made possible by combining evidence, using all the techniques for knowledge representation and uncertain reasoning that we have seen throughout this book.

We can break the knowledge down into four models:

1. The **world model**: the likelihood that a proposition occurs in the world.
2. The **mental model**: the likelihood that the speaker forms the intention of communicating a certain fact to the hearer, given that it occurs. This approach combines models of what the speaker believes, what the speaker believes the hearer believes, and so on.
3. The **language model**: the likelihood that a certain string of words will be chosen, given that the speaker has the intention of communicating a certain fact. The CFG and DCG models presented in this chapter have a Boolean model of likelihood: either a string can have a certain interpretation or it cannot. In the next chapter, we will see a probabilistic version of CFG that makes for a more informed language model for disambiguation.
4. The **acoustic model**: the likelihood that a particular sequence of sounds will be generated, given that the speaker has chosen a given string of words. Section 15.6 covered speech recognition.

22.7 Discourse Understanding

DISCOURSE

A **discourse** is any string of language—usually one that is more than one sentence long. Textbooks, novels, weather reports and conversations are all discourses. So far we have largely ignored the problems of discourse, preferring to dissect language into individual sentences that can be studied *in vitro*. This section studies sentences in their native habitat. We will look at two particular subproblems: reference resolution and coherence.

Reference resolution

REFERENCE RESOLUTION

Reference resolution is the interpretation of a pronoun or a definite noun phrase that refers to an object in the world.[14] The resolution is based on knowledge of the world and of the previous parts of the discourse. Consider the passage

> "John flagged down the waiter. He ordered a ham sandwich."

To understand that "he" in the second sentence refers to John, we need to have understood that the first sentence mentions two people and that John is playing the role of a customer and hence is likely to order, whereas the waiter is not. Usually, reference resolution is a matter of selecting a referent from a list of candidates, but sometimes it involves the creation of new candidates. Consider the following sentence:

> "After John proposed to Marsha, they found a preacher and got married. For the honeymoon, they went to Hawaii."

Here, the definite noun phrase "the honeymoon" refers to something that was only implicitly alluded to by the verb "married." The pronoun "they" refers to a group that was not explicitly mentioned before: John and Marsha (but *not* the preacher).

[14] In linguistics, reference to something that has already been introduced is called **anaphoric** reference. Reference to something yet to be introduced is called **cataphoric** reference, as with the pronoun "he" in "When he won his first tournament, Tiger was 20."

Choosing the best referent is a process of disambiguation that relies on combining a variety of syntactic, semantic, and pragmatic information. Some clues are in the form of constraints. For example, pronouns must agree in gender and number with their antecedents: "he" can refer to John, but not Marsha; "they" can refer to a group, but not a single person. Pronouns must also obey syntactic constraints for reflexivity. For example, in "He saw him in the mirror" the two pronouns must refer to different people, whereas in "He saw himself," they must refer to the same person. There are also constraints for semantic consistency. In "He ate it," the pronoun "he" must refer to something that eats and "it" to something that can be eaten.

Some clues are preferences that do not always hold. For example, when adjacent sentences have a parallel structure, it is preferable for pronominal reference to follow that structure. So in

Marsha flew to San Francisco from New York. John flew there from Boston.

we prefer for "there" to refer to San Francisco because it plays the same syntactic role. Absent a parallel structure, there is a preference for subjects over objects as antecedents. Thus, in

Marsha gave Sally the homework assignment. Then she left.

"Marsha," the subject of the first sentence, is the preferred antecedent for "she." Another preference is for the entity that has been discussed most prominently. Considered in isolation, the pair of sentences

Dana dropped the cup on the plate. It broke.

poses a problem: it is not clear whether the cup or the plate is the referent of "it." But in a larger context the ambiguity is resolved:

> Dana was quite fond of the blue cup. The cup had been a present from a close friend. Unfortunately, one day while setting the table. Dana dropped the cup on the plate. It broke.

Here, the cup is the focus of attention and hence is the preferred referent.

A variety of reference resolution algorithms have been devised. One of the first (Hobbs, 1978) is remarkable because it underwent a degree of statistical verification that was unusual for the time. Using three different genres of text, Hobbs reports an accuracy of 92%. This assumed that a correct parse was generated by a parser; not having one available, Hobbs constructed the parses by hand. The Hobbs algorithm works as a search: it searches sentences starting from the current sentence and going backwards. This technique ensures that more recent candidates will be considered first. Within a sentence it searches breadth first, from left to right. This ensures that subjects will be considered before objects. The algorithm chooses the first candidate that satisfies the constraints just outlined.

The structure of coherent discourse

Open up this book to 10 random pages, and copy down the first sentence from each page. The result is bound to be incoherent. Similarly, if you take a coherent 10-sentence passage and permute the sentences, the result is incoherent. This demonstrates that sentences in natural

- **Enable** or **cause:** S_1 brings about a change of state (which may be implicit) that causes or enables S_2. Example: "I went outside. I drove to school." (Going outside enables the implicit getting into a car.)
- **Explanation:** The reverse of enablement: S_2 causes or enables S_1 and thus is an explanation for it. Example: "I was late for school. I overslept."
- **Ground-Figure:** S_1 describes a setting or background for S_2. Example: "It was a dark and stormy night. *Rest of story.*"
- **Evaluation:** From S_2 infer that S_1 is part of the speaker's plan for executing the segment as a speech act. Example: "A funny thing happened. *Rest of story.*"
- **Exemplification:** S_2 is an example of the general principle in S_1. Example: "This algorithm reverses a list. The input $[A, B, C]$ is mapped to $[C, B, A]$."
- **Generalization:** S_1 is an example of the general principle in S_2. Example: "$[A, B, C]$ is mapped to $[C, B, A]$. In general, the algorithm reverses a list."
- **Violated Expectation:** Infer $\neg P$ from S_2, negating the normal inference of P from S_1. Example: "This paper is weak. On the other hand, it is interesting."

Figure 22.21 A list of coherence relations, taken from Hobbs (1990). Each relation holds between two adjacent text segments, S_1 and S_2.

language discourse are quite different from sentences in logic. In logic, if we TELL sentences A, B and C to a knowledge base, in any order, we end up with the conjunction $A \wedge B \wedge C$. In natural language, sentence order matters; consider the difference between "Go two blocks. Turn right." and "Turn right. Go two blocks."

A discourse has structure above the level of a sentence. We can examine this structure with the help of a grammar of discourse:

$$Segment(x) \rightarrow S(x)$$
$$Segment(CoherenceRelation(x, y)) \rightarrow Segment(x)\ Segment(y)\ .$$

COHERENCE RELATIONS

This grammar says that a discourse is composed of segments, where each segment is either a sentence or a group of sentences and where segments are joined by **coherence relations**. In the text "Go two blocks. Turn right," the coherence relation is that the first sentence *enables* the second: the listener should turn right only after traveling two blocks. Different researchers have proposed different inventories of coherence relations; Figure 22.21 lists a representative set. Now consider the following story:

(1) A funny thing happened yesterday.
(2) John went to a fancy restaurant.
(3) He ordered the duck.
(4) The bill came to $50.
(5) John got a shock when he realized he had no money.
(6) He had left his wallet at home.
(7) The waiter said it was all right to pay later.
(8) He was very embarrassed by his forgetfulness.

Here, sentence (1) stands in the *Evaluation* relation to the rest of the discourse; (1) is the speaker's metacomment on the discourse. Sentence (2) enables (3), and together the (2–3) pair cause (4), with the implicit intermediate state that John ate the duck. Now (2–4) serve as the ground for the rest of the discourse. Sentence (6) is an explanation of (5), and (5–6) enable (7). Note that this is an *Enable* and not a *Cause*, because the waiter might have had a different reaction. Together, (5–7) cause (8). Exercise 22.13 asks you to draw the parse tree for this discourse.

Coherence relations serve to bind a discourse together. They guide the speaker in deciding what to say and what to leave implicit, and they guide the hearer in recovering the speaker's intent. Coherence relations can serve as a filter on the ambiguity of sentences: individually, the sentences might be ambiguous, but most of these ambiguous interpretations do not fit together into a coherent discourse.

So far we have looked at reference resolution and discourse structure separately. But the two are actually intertwined. The theory of Grosz and Sidner (1986), for example, accounts for where the speaker's and hearer's attention is focused during the discourse. Their theory includes a pushdown stack of **focus spaces**. Certain utterances cause the focus to shift by pushing or popping elements off the stack. For example, in the restaurant story, the sentence "John went to a fancy restaurant" pushes a new focus onto the stack. Within that focus, the speaker can use a definite *NP* to refer to "the waiter" (rather than "a waiter"). If the story continued with "John went home," then the focus space would be popped from the stack and the discourse could no longer refer to the waiter with "the waiter" or "he."

FOCUS SPACES

22.8 Grammar Induction

GRAMMAR INDUCTION

Grammar induction is the task of learning a grammar from data. It is an obvious task to attempt, given that it has proven to be so difficult to construct a grammar by hand and that billions of example utterances are available for free on the Internet. It is a difficult task because the space of possible grammars is infinite and because verifying that a given grammar generates a set of sentences is computationally expensive.

One interesting model is the SEQUITUR system (Nevill-Manning and Witten, 1997). It requires no input except a single text (which does not need to be predivided into sentences). It produces a grammar of a very specialized form: a grammar that generates only a single string, namely, the original text. Another way to look at this is that SEQUITUR learns just enough grammar to parse the text. Here is the bracketing it discovers for one sentence within a larger text of news stories:

[Most Labour] [sentiment [[would still] [favor the] abolition]] [[of [the House]] [of Lords]]

It has correctly picked out constituents such as the *PP* "of the House of Lords," although it also goes against traditional analysis in, for example, grouping "the" with the preceding verb rather than the following noun.

SEQUITUR is based on the idea that a good grammar is a compact grammar. In particular, it enforces the following two constraints: (1) No pair of adjacent symbols should appear

more than once in the grammar. If the symbol pair $A\ B$ appears on the right-hand side of several rules, then we should replace the pair with a new nonterminal that we will call C and add the rule $C \rightarrow A\ B$. (2) Every rule should be used at least twice. If a nonterminal C appears only once in the grammar, then we should eliminate the rule for C and replace its single use with the rule's right-hand side. These two constraints are applied in a greedy search that scans the input text from left to right, incrementally building a grammar as it goes, and imposing the constraints as soon as possible. Figure 22.22 shows the algorithm in operation on the input text "*abcdbcabcd*." The algorithm recovers an optimally compact grammar for the text.

	Input	Grammar	Comments
1	a	$S \rightarrow a$	
2	ab	$S \rightarrow ab$	
3	abc	$S \rightarrow abc$	
4	$abcd$	$S \rightarrow abcd$	
5	$abcdb$	$S \rightarrow abcdb$	
6	$abcdbc$	$S \rightarrow abcdbc$	bc twice
		$S \rightarrow aAdA;\ A \rightarrow bc$	
7	$abcdbca$	$S \rightarrow aAdAa;\ A \rightarrow bc$	
8	$abcdbcab$	$S \rightarrow aAdAab;\ A \rightarrow bc$	
9	$abcdbcabc$	$S \rightarrow aAdAabc;\ A \rightarrow bc$	bc twice
		$S \rightarrow aAdAaA;\ A \rightarrow bc$	aA twice
		$S \rightarrow BdAB;\ A \rightarrow bc;\ B \rightarrow aA$	
10	$abcdbcabcd$	$S \rightarrow BdABd;\ A \rightarrow bc;\ B \rightarrow aA$	Bd twice
		$S \rightarrow CAC;\ A \rightarrow bc;\ B \rightarrow aA;\ C \rightarrow Bd$	B only once
		$S \rightarrow CAC;\ A \rightarrow bc;\ C \rightarrow aAd$	

Figure 22.22 A trace of SEQUITUR inducing a grammar for the input text "*abcdbcabcd*." We start with a rule for S and add each symbol to the end of this rule in turn. After adding the sixth symbol, we have the first occurrence of a repeated pair: bc. So we replace both occurrences of bc with the new nonterminal A and add the rule $A \rightarrow bc$. After three more symbols are added, the ninth causes another repetition of bc, so again we replace it with A. This leads to two occurrences of aA, so we replace them with a new nonterminal, B. After adding the tenth and last terminal symbol we get two occurrences of Bd, so we replace them with the new nonterminal C. But now B appears only once, in the right-hand side of the C rule, so we replace B by its expansion, aA.

In the next chapter, we will see other grammar induction algorithms that work with probabilistic context-free grammars. But now we turn to the problem of learning a grammar that is augmented with semantics. Since an augmented grammar is a Horn clause logic program, the techniques of inductive logic programming are appropriate. CHILL (Zelle and Mooney, 1996) is an inductive logic programming (ILP) program that learns a grammar and a specialized parser for that grammar from examples. The target domain is natural language

database queries. The training examples consist of pairs of word strings and corresponding queries—for example;

> What is the capital of the state with the largest population?
> $Answer(c, Capital(s, c) \wedge Largest(p, State(s) \wedge Population(s, p)))$

CHILL's task is to learn a predicate $Parse(words, query)$ that is consistent with the examples and, hopefully, generalizes well to other examples. Applying ILP directly to learn this predicate results in poor performance: the induced parser has only about 20% accuracy. Fortunately, ILP learners can improve by adding knowledge. In this case, most of the *Parse* predicate was defined as a logic program, and CHILL's task was reduced to inducing the control rules that guide the parser to select one parse over another. With this additional background knowledge, CHILL achieves 70% to 85% accuracy on various database query tasks.

22.9 SUMMARY

Natural language understanding is one of the most important subfields of AI. It draws on ideas from philosophy and linguistics, as well as on techniques of logical and probabilistic knowledge representation and reasoning. Unlike other areas of AI, natural language understanding requires an empirical investigation of actual human behavior—which turns out to be complex and interesting.

- Agents send signals to each other to achieve certain purposes: to inform, to warn, to elicit help, to share knowledge, or to promise something. Sending a signal in this way is called a **speech act**. Ultimately, all speech acts are an attempt to get another agent to believe something or do something.
- Language consists of conventional **signs** that convey meaning. Many animals use signs in this sense. Humans appear to be the only animals that use **grammar** to produce an unbounded variety of structured messages.
- Communication involves three steps by the speaker: the intention to convey an idea, the mental generation of words, and their physical synthesis. The hearer then has four steps: perception, analysis, disambiguation, and incorporation of the meaning. All language use is **situated**, in the sense that the meaning of an utterance can depend on the situation in which it is produced.
- Formal language theory and **phrase structure** grammars (and in particular, **context-free** grammar) are useful tools for dealing with some aspects of natural language.
- Sentences in a context-free language can be parsed in $O(n^3)$ time by a **chart parser**.
- It is convenient to **augment** a grammar to handle such problems as subject–verb agreement and pronoun case. **Definite clause grammar** (DCG) is a formalism that allows for augmentations. With DCG, parsing and semantic interpretation (and even generation) can be done using logical inference.
- **Semantic interpretation** can also be handled by an augmented grammar. A quasi-logical form can be a useful intermediate between syntactic trees and semantics.

- **Ambiguity** is a very important problem in natural language understanding; most sentences have many possible interpretations, but usually only one is appropriate. Disambiguation relies on knowledge about the world, about the current situation, and about language use.
- Most language exists in the context of multiple sentences, not just a single one. **Discourse** is the study of connected texts. We saw how to resolve pronominal references across sentences and how sentences are joined into coherent segments.
- **Grammar induction** can learn a grammar from examples, although there are limitations on how well the grammar will generalize.

BIBLIOGRAPHICAL AND HISTORICAL NOTES

SEMIOTICS

The study of signs and symbols as elements of language was named **semiotics** by John Locke (1690), although it was not developed until the 20th century (Peirce, 1902; de Saussure, 1993). Recent overview texts include Eco's (1979) and Cobley's (1997).

The idea of language as action stems from 20th-century linguistically oriented philosophy (Wittgenstein, 1953; Grice, 1957; Austin, 1962) and particularly from the book *Speech Acts* (Searle, 1969). A precursor to the idea of speech acts was Protagoras's (c. 430 B.C.) identification of four types of sentence: prayer, question, answer, and injunction. A plan-based model of speech acts was suggested first by Cohen and Perrault (1979). Connecting language to action by using plan recognition to understand stories was studied by Wilensky (1983). Cohen, Morgan, and Pollack (1990) collect more recent work in this area.

Like semantic networks, context-free grammars (also known as phrase structure grammars) are a reinvention of a technique first used by ancient Indian grammarians (especially Panini, c. 350 B.C.) studying Shastric Sanskrit (Ingerman, 1967). They were reinvented by Noam Chomsky (1956) for the analysis of English syntax and independently by John Backus for the analysis of Algol-58 syntax. Naur (1963) extended Backus's notation and is now credited (Backus, 1996) with the "N" in BNF, which originally stood for "Backus Normal Form." Knuth (1968) defined a kind of augmented grammar called **attribute grammar** that is useful for programming languages. Definite clause grammars were introduced by Colmerauer (1975) and developed and popularized by Pereira and Warren (1980). The Prolog programming language was invented by Alain Colmerauer specifically for the problem of parsing the French language. Colmerauer actually introduced a formalism called metamorphosis grammar that went beyond definite clauses, but DCG followed soon after.

ATTRIBUTE GRAMMAR

There have been many attempts to write formal grammars of natural languages, both in "pure" linguistics and in computational linguistics. Machine-oriented grammars include those developed in the Linguistic String Project at New York University (Sager, 1981) and the XTAG project at the University of Pennsylvania (Doran *et al.*, 1994). A good example of a modern DCG system is the Core Language Engine (Alshawi, 1992). There are several comprehensive but informal grammars of English (Jespersen, 1965; Quirk *et al.*, 1985; McCawley, 1988; Huddleston and Pullum, 2002). Good textbooks on linguistics include Sag and

Wasow's (1999) introduction to syntax and the semantics texts by Chierchia and McConnell-Ginet (1990) and by Heim and Kratzer (1998). McCawley's (1993) text concentrates on logic for linguists.

Since the mid-1980s, there has been a trend toward putting more information in the lexicon and less in the grammar. Lexical-functional grammar, or LFG, (Bresnan, 1982) was the first major grammar formalism to be highly lexicalized. If we carry lexicalization to an extreme, we end up with **categorial grammar**, in which there can be as few as two grammar rules, or **dependency grammar** (Melćuk and Polguere, 1988), in which there are no phrases, only words. Sleator and Temperley (1993) describe a popular parser that uses dependency grammar. Tree-Adjoining Grammar, or TAG, (Joshi, 1985) is not strictly lexical, but it is gaining popularity in its lexicalized form (Schabes *et al.*, 1988). Wordnet (Fellbaum, 2001) is a publicly-available dictionary of about 100,000 words and phrases, categorized into parts of speech and linked by semantic relations such as synonym, antonym, and part-of.

The first computerized parsing algorithms were demonstrated by Yngve (1955). Efficient algorithms were developed in the late 1960s, with a few twists since then (Kasami, 1965; Younger, 1967; Graham *et al.*, 1980). Our chart parser is closest to Earley's (1970). A good summary appears in the text on parsing and compiling by Aho and Ullman (1972). Maxwell and Kaplan (1993) show how chart parsing with augmentations can be made efficient in the average case. Church and Patil (1982) address the resolution of syntactic ambiguity.

Formal semantic interpretation of natural languages originates within philosophy and formal logic and is especially closely related to Alfred Tarski's (1935) work on the semantics of formal languages. Bar-Hillel was the first to consider the problems of pragmatics and propose that they could be handled by formal logic. For example, he introduced C. S. Peirce's (1902) term *indexical* into linguistics (Bar-Hillel, 1954). Richard Montague's essay "English as a formal language" (1970) is a kind of manifesto for the logical analysis of language, but the book by Dowty *et al.* (1991) and the article by Lewis (1972) are more readable. A complete collection of Montague's contributions has been edited by Thomason (1974). In artificial intelligence, the work of McAllester and Givan (1992) continues the Montagovian tradition, adding many new technical insights.

The idea of an intermediate or quasi-logical form to handle problems such as quantifier scoping goes back to Woods (1978) and is present in many recent systems (Alshawi, 1992; Hwang and Schubert, 1993).

The first NLP system to solve an actual task was probably the BASEBALL question answering system (Green *et al.*, 1961), which handled questions about a database of baseball statistics. Close after that was Woods's (1973) LUNAR, which answered questions about the rocks brought back by the Apollo program. Roger Schank and his students built a series of programs (Schank and Abelson, 1977; Wilensky, 1978; Schank and Riesbeck, 1981; Dyer, 1983) that all had the task of understanding language. The emphasis, however, was less on language *per se* and more on representation and reasoning. The problems included representing stereotypical situations (Cullingford, 1981), describing human memory organization (Rieger, 1976; Kolodner, 1983), and understanding plans and goals (Wilensky, 1983).

Natural language generation was considered from the earliest days of machine translation in the 1950s, but it didn't appear as a monolingual concern until the 1970s. The work

by Simmons and Slocum (1972) and Goldman (1975) are representative. PENMAN (Bateman *et al.*, 1989) was one of the first full-scale generation systems, based on Systemic Grammar (Kasper, 1988). In the 1990s, two important public-domain generation systems, KPML (Bateman, 1997) and FUF (Elhadad, 1993), became available. Important books on generation include McKeown (1985), Hovy (1988), Patten (1988) and Reiter and Dale (2000).

Some of the earliest work on disambiguation was Wilks's (1975) theory of **preference semantics**, which tried to find interpretations that minimize the number of semantic anomalies. Hirst (1987) describes a system with similar aims that is closer to the compositional semantics described in this chapter. Hobbs *et al.* (1993) describes a quantitative framework for measuring the quality of a syntactic and semantic interpretation. Since then, it has become more common to use an explicitly Bayesian framework (Charniak and Goldman, 1992; Wu, 1993). In linguistics, optimality theory (Linguistics) (Kager, 1999) is based on the idea of building soft constraints into the grammar, giving a natural ranking to interpretations, rather than having the grammar generate all possibilities with equal rank. Norvig (1988) discusses the problems of considering multiple simultaneous interpretations, rather than settling for a single maximum likelihood interpretation. Literary critics (Empson, 1953; Hobbs, 1990) have been ambiguous about whether ambiguity is something to be resolved or cherished.

Nunberg (1979) outlines a formal model of metonymy. Lakoff and Johnson (1980) give an engaging analysis and catalog of common metaphors in English. Ortony (1979) presents a collection of articles on metaphor; Martin (1990) offers a computational approach to metaphor interpretation.

Our treatment of reference resolution follows Hobbs (1978). A more complex model by Lappin and Leass (1994) is based on a quantitative scoring mechanism. More recent work (Kehler, 1997; Ge *et al.*, 1998) has used machine learning to tune the quantitative parameters. Two excellent surveys of reference resolution are the books by Hirst (1981) and Mitkov (2002).

In 1758, David Hume's *Enquiry Concerning the Human Understanding* argued that discourse is connected by "three principles of connexion among ideas, namely *Resemblance*, *Contiguity* in time or place, and *Cause* or *Effect*." So began a long history of trying to define coherence relations. Hobbs (1990) gives us the set used in this chapter; Mann and Thompson (1983) provide a more elaborate set that includes solutionhood, evidence, justification, motivation, reason, sequence, enablement, elaboration, restatement, condition, circumstance, cause, concession, background, and thesis–antithesis. That model evolved into rhetorical structure theory (RST), which is perhaps the most prominent theory today (Mann and Thompson, 1988). This chapter borrows some of the examples from the chapter in Jurafsky and Martin (2000) written by Andrew Kehler.

Grosz and Sidner (1986) present a theory of discourse coherence based on shifting one's focus of attention, and Grosz *et al.* (1995) offer a related theory based on the notion of centering. Joshi, Webber, and Sag (1981) collect important early work on discourse. Webber presents a model of the interacting constraints of syntax and discourse on what can be said at any point in the discourse (1983) and of the way verb tense interacts with discourse (1988).

The first important result on **grammar induction** was a negative one: Gold (1967) showed that it is not possible to reliably learn a correct context-free grammar, given a set of

strings from that grammar. Essentially, the idea is that, given a set of strings $s_1, s_2 \ldots s_n$, the correct grammar could be all-inclusive ($S \rightarrow word^*$), or it could be a copy of the input ($S \rightarrow s_1 \mid s_2 \mid \ldots \mid s_n$), or anywhere in between. Prominent linguists, such as Chomsky (1957, 1980) and Pinker (1989, 2000), have used Gold's result to argue that there must be an innate **universal grammar** that all children have from birth. The so-called *Poverty of the Stimulus* argument is that children have no language inputs other than positive examples: their parents and peers produce mostly accurate examples of their language, and very rarely correct mistakes. Therefore, because Gold proved that learning a CFG from positive examples is impossible, the children must already "know" the grammar and be merely tuning some of its parameters of this innate grammar and learning vocabulary. While this argument continues to hold sway throughout much of Chomskian linguistics, it has been dismissed by some other linguists (Pullum, 1996; Elman *et al.*, 1997) and most computer scientists. As early as 1969, Horning showed that it *is* possible to learn, in the sense of PAC learning, a *probabilistic* context-free grammar. Since then there have been many convincing empirical demonstrations of learning from positive examples alone, such as the ILP work of Mooney (1999) and Muggleton and De Raedt (1994) and the remarkable Ph.D. theses of Schütze (1995) and de Marcken (1996). It is possible to learn other grammar formalisms, such as regular languages (Oncina and Garcia, 1992; Denis, 2001), and regular tree languages (Carrasco *et al.*, 1998), and finite state automata (Parekh and Honavar, 2001).

UNIVERSAL GRAMMAR

The SEQUITUR system is due to Nevill-Manning and Witten (1997). Interestingly, they, as well as de Marcken, remark that their grammar induction schemes are also good compression schemes. This is in accordance with the principle of minimal description length encoding: a good grammar is a grammar that minimizes the sum of two lengths: the length of the grammar and the length of the parse tree of the text.

Inductive Logic Programming work for language learning includes the CHILL system by Zelle and Mooney (1996) and a program by Mooney and Califf (1995) that learned rules for the past tense of verbs better than past neural net or decision tree systems. Cussens and Dzeroski (2000) edited a collection of papers on learning language in logic.

The Association for Computational Linguistics (ACL) holds regular conferences and publishes the journal *Computational Linguistics*. There is also an International Conference on Computational Linguistics (COLING). *Readings in Natural Language Processing* (Grosz *et al.*, 1986) is an anthology containing many important early papers. Dale *et al.* (2000) emphasize practical tools for building NLP systems. The textbook by Jurafsky and Martin (2000) gives a comprehensive introduction to the field. Allen (1995) is a slightly older treatment. Pereira and Sheiber (1987) and Covington (1994) offer concise overviews of syntactic processing based on implementations in Prolog. The *Encyclopedia of AI* has many useful articles on the field; see especially the entries "Computational Linguistics" and "Natural Language Understanding."

EXERCISES

22.1 Read the following text once for understanding, and remember as much of it as you can. There will be a test later.

> The procedure is actually quite simple. First you arrange things into different groups. Of course, one pile may be sufficient depending on how much there is to do. If you have to go somewhere else due to lack of facilities that is the next step, otherwise you are pretty well set. It is important not to overdo things. That is, it is better to do too few things at once than too many. In the short run this may not seem important but complications can easily arise. A mistake is expensive as well. At first the whole procedure will seem complicated. Soon, however, it will become just another facet of life. It is difficult to foresee any end to the necessity for this task in the immediate future, but then one can never tell. After the procedure is completed one arranges the material into different groups again. Then they can be put into their appropriate places. Eventually they will be used once more and the whole cycle will have to be repeated. However, this is part of life.

22.2 Using DCG notation, write a grammar for a language that is just like $\mathcal{E}_1$, except that it enforces agreement between the subject and verb of a sentence and thus does not generate "I smells the wumpus."

22.3 Augment the $\mathcal{E}_1$ grammar so that it handles article–noun agreement. That is, make sure that "agents" is an NP, but "agent" and "an agents" are not.

22.4 Outline the major differences between Java (or any other computer language with which you are familiar) and English, commenting on the "understanding" problem in each case. Think about such things as grammar, syntax, semantics, pragmatics, compositionality, context-dependence, lexical ambiguity, syntactic ambiguity, reference finding (including pronouns), background knowledge, and what it means to "understand" in the first place.

22.5 Which of the following are reasons for introducing a quasi-logical form?

a. To make it easier to write simple compositional grammar rules.
b. To extend the expressiveness of the semantic representation language.
c. To be able to represent quantifier scoping ambiguities (among others) in a succinct form.
d. To make it easier to do semantic disambiguation.

22.6 Determine what semantic interpretation would be given to the following sentences by the grammar in this chapter:

a. It is a wumpus.
b. The wumpus is dead.
c. The wumpus is in 2,2.

Would it be a good idea to have the semantic interpretation for "It is a wumpus" be simply $\exists x \;\; x \in Wumpuses$? Consider alternative sentences such as "It was a wumpus."

22.7 Without looking back at Exercise 22.1, answer the following questions:

a. What are the four steps that are mentioned?

b. What step is left out?

c. What is "the material" that is mentioned in the text?

d. What kind of mistake would be expensive?

e. Is it better to do too few or too many? Why?

22.8 This exercise concerns grammars for very simple languages.

a. Write a context-free grammar for the language $a^n b^n$.

b. Write a context-free grammar for the palindrome language: the set of all strings whose second half is the reverse of the first half.

c. Write a context-sensitive grammar for the duplicate language: the set of all strings whose second half is the same as the first half.

22.9 Consider the sentence "Someone walked slowly to the supermarket" and the following lexicon:

Pronoun $\rightarrow$ **someone**	*V* $\rightarrow$ **walked**
Adv $\rightarrow$ **slowly**	*Prep* $\rightarrow$ **to**
Det $\rightarrow$ **the**	*Noun* $\rightarrow$ **supermarket**

Which of the following three grammars, combined with the lexicon, generates the given sentence? Show the corresponding parse tree(s).

(A):	(B):	(C):
$S \rightarrow NP\ VP$	$S \rightarrow NP\ VP$	$S \rightarrow NP\ VP$
$NP \rightarrow Pronoun$	$NP \rightarrow Pronoun$	$NP \rightarrow Pronoun$
$NP \rightarrow Article\ Noun$	$NP \rightarrow Noun$	$NP \rightarrow Article\ NP$
$VP \rightarrow VP\ PP$	$NP \rightarrow Article\ NP$	$VP \rightarrow Verb\ Adv$
$VP \rightarrow VP\ Adv\ Adv$	$VP \rightarrow Verb\ Vmod$	$Adv \rightarrow Adv\ Adv$
$VP \rightarrow Verb$	$Vmod \rightarrow Adv\ Vmod$	$Adv \rightarrow PP$
$PP \rightarrow Prep\ NP$	$Vmod \rightarrow Adv$	$PP \rightarrow Prep\ NP$
$NP \rightarrow Noun$	$Adv \rightarrow PP$	$NP \rightarrow Noun$
	$PP \rightarrow Prep\ NP$	

For each of the preceding three grammars, write down three sentences of English and three sentences of non-English generated by the grammar. Each sentence should be significantly different, should be at least six words long, and should be based on an entirely new set of lexical entries (which you should define). Suggest ways to improve each grammar to avoid generating the non-English sentences.

22.10 Implement a version of the chart-parsing algorithm that returns a packed tree of all edges that span the entire input.

22.11 Implement a version of the chart-parsing algorithm that returns a packed tree for the longest leftmost edge, and then if that edge does not span the whole input, continues the parse

from the end of that edge. Show why you will need to call PREDICT before continuing. The final result is a list of packed trees such that the list as a whole spans the input.

22.12 (Derived from Barton *et al.* (1987).) This exercise concerns a language we call *Buffalo*^{n}, which is very much like English (or at least $\mathcal{E}_0$), except that the only word in its lexicon is *buffalo*. Here are two sentences from the language:

Buffalo buffalo buffalo Buffalo buffalo.

Buffalo Buffalo buffalo buffalo buffalo Buffalo buffalo.

In case you don't believe these are sentences, here are two English sentences with corresponding syntactic structure:

Dallas cattle bewilder Denver cattle.

Chefs London critics admire cook French food.

Write a grammar for *Buffalo*^{n}. The lexical categories are city, plural noun, and (transitive) verb, and there should be one grammar rule for sentence, one for verb phrase, and three for noun phrase: plural noun, noun phrase preceded by a city as a modifier, and noun phrase followed by a reduced relative clause. A reduced relative clause is a clause that is missing the relative pronoun. In addition, the clause consists of a subject noun phrase followed by a verb without an object. An example reduced relative clause is "London critics admire" in the example above. Tabulate the number of possible parses for *Buffalo*^{n} for n up to 10. Extra credit: Carl de Marcken calculated that there are 121,030,872,213,055,159,681,184,485 *Buffalo*^{n} sentences of length 200 (for the grammar he used). How did he do that?

22.13 Draw a discourse parse tree for the story on page 823 about John going to a fancy restaurant. Use to the two grammar rules for *Segment*, giving the proper *CoherenceRelation* for each node. (You needn't show the parse for individual sentences.) Now do the same for a 5 to 10–sentence discourse of your choosing.

22.14 We forgot to mention that the text in Exercise 22.1 is entitled "Washing Clothes." Reread the text and answer the questions in Exercise 22.7. Did you do better this time? Bransford and Johnson (1973) used this text in a better controlled experiment and found that the title helped significantly. What does this tell you about discourse comprehension?

23 PROBABILISTIC LANGUAGE PROCESSING

In which we see how simple, statistically trained language models can be used to process collections of millions of words, rather than just single sentences.

In Chapter 22, we saw how an agent could communicate with another agent (human or software), using utterances in a common language. Complete syntactic and semantic analysis of the utterances is *necessary* to extract the full meaning of the utterances, and is *possible* because the utterances are short and restricted to a limited domain.

CORPUS-BASED

In this chapter, we consider the **corpus-based** approach to language understanding. A corpus (plural *corpora*) is a large collection of text, such as the billions of pages that make up the World Wide Web. The text is written by and for humans, and the task of the software is to make it easier for the human to find the right information. This approach implies the use of statistics and learning to take advantage of the corpus, and it usually entails probabilistic language models that can be learned from data and that are simpler than the augmented DCGs of Chapter 22. For most tasks, the volume of data more than makes up for the simpler language model. We will look at three specific tasks: information retrieval (Section 23.2), information extraction (Section 23.3), and machine translation (Section 23.4). But first we present an overview of probabilistic language models.

23.1 PROBABILISTIC LANGUAGE MODELS

Chapter 22 gave us a *logical* model of language: we used CFGs and DCGs to characterize a string as either a member or a nonmember of a language. In this section, we will introduce several *probabilistic* models. Probabilistic models have several advantages. They can conveniently be trained from data: learning is just a matter of counting occurrences (with some allowances for the errors of relying on a small sample size). Also, they are more robust (because they can accept *any* string, albeit with a low probability), they reflect the fact that not 100% of speakers agree on which sentences are actually part of a language, and they can be used for disambiguation: probability can be used to choose the most likely interpretation.

PROBABILISTIC LANGUAGE MODEL

A **probabilistic language model** defines a probability distribution over a (possibly infinite) set of strings. Example models that we have already seen are the bigram and trigram

language models used in speech recognition (Section 15.6). A unigram model assigns a probability $P(w)$ to each word in the lexicon. The model assumes that words are chosen independently, so the probability of a string is just the product of the probability of its words, given by $\prod_i P(w_i)$. The following 20-word sequence was generated at random from a unigram model of the words in this book:

> logical are as are confusion a may right tries agent goal the was diesel more object then information-gathering search is

A bigram model assigns a probability $P(w_i|w_{i-1})$ to each word, given the previous word. Figure 15.21 listed some of these bigram probabilities. A bigram model of this book generates the following random sequence:

> planning purely diagnostic expert systems are very similar computational approach would be represented compactly using tic tac toe a predicate

In general, an n-gram model conditions on the previous $n-1$ words, assigning a probability for $P(w_i|w_{i-(n-1)} \ldots w_{i-1})$. A trigram model of this book generates this random sequence:

> planning and scheduling are integrated the success of naive bayes model is just a possible prior source by that time

Even with this small sample, it should be clear that the trigram model is better than the bigram model (which is better than the unigram model), both for approximating the English language and for approximating the subject matter of an AI textbook. The models themselves agree: the trigram model assigns its random string a probability of 10^{-10}, the bigram 10^{-29}, and the unigram 10^{-59}.

At half a million words, this book does not contain enough data to produce a good bigram model, let alone a trigram model. There are about 15,000 different words in the lexicon of this book, so the bigram model includes $15,000^2 = 225$ million word pairs. Clearly, at least 99.8% of these pairs will have a count of zero, but we don't want our model to say that all these pairs are impossible. We need some way of **smoothing** over the zero counts. The simplest way to do this is called **add-one smoothing**: we add one to the count of every possible bigram. So if there are N words in the corpus and B possible bigrams, then each bigram with an actual count of c is assigned a probability estimate of $(c+1)/(N+B)$. This method eliminates the problem of zero-probability n-grams, but the assumption that every count should be incremented by exactly one is dubious and can lead to poor estimates.

SMOOTHING

ADD-ONE SMOOTHING

LINEAR INTERPOLATION SMOOTHING

Another approach is **linear interpolation smoothing**, which combines trigram, bigram, and unigram models by linear interpolation. We define our probability estimate as

$$\hat{P}(w_i|w_{i-2}w_{i-1}) = c_3 P(w_i|w_{i-2}w_{i-1}) + c_2 P(w_i|w_{i-1}) + c_1 P(w_i) \ ,$$

where $c_3 + c_2 + c_1 = 1$. The parameters c_i can be fixed, or they can be trained with an EM algorithm. It is possible to have values of c_i that are dependent on the n-gram counts, so that we place a higher weight on the probability estimates that are derived from higher counts.

One method for *evaluating* a language model is as follows: First, split your corpus into a training corpus and a test corpus. Determine the parameters of the model from the training data. Then calculate the probability assigned to the test corpus by the model; the higher the

probability the better. One problem with this approach is that $P(words)$ is quite small for long strings; the numbers could cause floating point underflow, or could just be hard to read. So instead of probability we can compute the **perplexity** of a model on a test string of words:

PERPLEXITY

$$Perplexity(words) = 2^{-\log_2(P(words))/N} ,$$

where N is the number of $words$. The lower the perplexity, the better the model. An n-gram model that assigns every word a probability of $1/k$ will have perplexity k; you can think of perplexity as the average branching factor.

SEGMENTATION

As an example of what n-gram models can do, consider the task of **segmentation**: finding the words boundaries in a text with no spaces. This task is necessary in Japanese and Chinese, languages that are written with no spaces between words, but we assume that most readers will be more comfortable with English. The sentence

Itiseasytoreadwordswithoutspaces

is in fact easy for us to read. You might think that is because we have our full knowledge of English syntax, semantics, and pragmatics. We will show that the sentence can be decoded easily by a simple unigram word model.

Earlier we saw how the Viterbi equation (15.9), can be used to solve the problem of finding the most probable sequence through a lattice of word possibilities. Figure 23.1 shows a version of the Viterbi algorithm specifically designed for the segmentation problem. It takes as input a unigram word probability distribution, $P(word)$, and a string. Then, for each position i in the string, it stores in $best[i]$ the probability of the most probable string spanning from the start up to i. It also stores in $words[i]$ the word ending at position i that yielded the best probability. Once it has built up the $best$ and $words$ arrays in a dynamic programming fashion, it then works backwards through $words$ to find the best path. In this case, with the unigram model from the book, the best sequence of words is indeed "It is easy to read words without spaces," with probability 10^{-25}. Comparing subparts of the sequence, we see for example that "easy" has unigram probability 2.6×10^{-4}, whereas the alternative "e as y" has a much lower probability, 9.8×10^{-12}, despite the fact that the words e and y are fairly common in equations in the book. Similarly, we have

$$P(\text{“without”}) = 0.0004 ;$$
$$P(\text{“with”}) = 0.005; \; P(\text{“out”}) = 0.0008 ;$$
$$P(\text{“with out”}) = 0.005 \times 0.0008 = 0.000004 .$$

Hence, "without" is 100 times more likely than "with out," according to the unigram model.

In this section we have discussed n-gram models over words, but there are also many uses of n-gram models over other units, such as characters or parts of speech.

Probabilistic context-free grammars

PROBABILISTIC CONTEXT-FREE GRAMMAR

n-gram models take advantage of co-occurrence statistics in the corpora, but they have no notion of grammar at distances greater than n. An alternative language model is the **probabilistic context-free grammar**, or PCFG,[1] which consists of a CFG wherein each rewrite

[1] PCFGs are also known as stochastic context-free grammars or SCFGs.

```
function VITERBI-SEGMENTATION(text, P) returns best words and their probabilities
  inputs: text, a string of characters with spaces removed
          P, a unigram probability distribution over words

  n ← LENGTH(text)
  words ← empty vector of length n + 1
  best ← vector of length n + 1, initially all 0.0
  best[0] ← 1.0
  /* Fill in the vectors best, words via dynamic programming */
  for i = 0 to n do
    for j = 0 to i − 1 do
      word ← text[j:i]
      w ← LENGTH(word)
      if P[word] × best[i - w] ≥ best[i] then
        best[i] ← P[word] × best[i − w]
        words[i] ← word
  /* Now recover the sequence of best words */
  sequence ← the empty list
  i ← n
  while i > 0 do
    push words[i] onto front of sequence
    i ← i − LENGTH(words[i])
  /* Return sequence of best words and overall probability of sequence */
  return sequence, best[i]
```

Figure 23.1 A Viterbi-based word segmentation algorithm. Given a string of words with spaces removed, it recovers the most probable segmentation into words.

rule has an associated probability. The sum of the probabilities across all rules with the same left-hand side is 1. Figure 23.2 shows a PCFG for a portion of the $\mathcal{E}_0$ grammar.

In the PCFG model, the probability of a string, $P(words)$, is just the sum of the probabilities of its parse trees. The probability of a given tree is the product of the probabilities of all the rules that make up the nodes of the tree. Figure 23.3 shows how to compute the probability of a sentence. It is possible to compute this probability by using a CFG chart parser to enumerate the possible parses and then simply adding up the probabilities. However, if we are interested only in the most probable parse then enumerating the unlikely ones is wasteful. We can use a variation of the Viterbi algorithm to find the most probable parse efficiently, or we can use a best-first search technique (such as A*).

The problem with PCFGs is that they are context-free. That means that the difference between P("eat a banana") and P("eat a bandanna") depends only on P("banana") versus P("bandanna") and not on the relation between "eat" and the respective objects. To get at that kind of relationship, we will need some kind of context-sensitive model, such as a **lexicalized PCFG**, in which the head of a phrase[2] can play a role in the probability of a

[2] The *head* of a phrase is the most important word, e.g., the noun of a noun phrase.

$$
\begin{array}{rcl}
S & \rightarrow & NP\ VP\ [1.00] \\
 & & \\
NP & \rightarrow & Pronoun\ [0.10] \\
 & | & Name\ [0.10] \\
 & | & Noun\ [0.20] \\
 & | & Article\ Noun\ [0.50] \\
 & | & NP\ PP\ [0.10] \\
 & & \\
VP & \rightarrow & Verb\ [0.60] \\
 & | & VP\ NP\ [0.20] \\
 & | & VP\ PP\ [0.20] \\
 & & \\
PP & \rightarrow & Preposition\ NP\ [1.00] \\
 & & \\
Noun & \rightarrow & \textbf{breeze}\ [0.10]\ |\ \textbf{wumpus}\ [0.15]\ |\ \textbf{agent}\ [0.05]\ |\ \ldots \\
Verb & \rightarrow & \textbf{sees}\ [0.15]\ |\ \textbf{smells}\ [0.10]\ |\ \textbf{goes}\ [0.25]\ |\ \ldots \\
Pronoun & \rightarrow & \textbf{me}\ [0.05]\ |\ \textbf{you}\ [0.10]\ |\ \textbf{I}\ [0.25]\ |\ \textbf{it}\ [0.20]\ |\ \ldots \\
Article & \rightarrow & \textbf{the}\ [0.30]\ |\ \textbf{a}\ [0.35]\ |\ \textbf{every}\ [0.05]\ |\ \ldots \\
Preposition & \rightarrow & \textbf{to}\ [0.30]\ |\ \textbf{in}\ [0.25]\ |\ \textbf{on}\ [0.05]\ |\ \ldots
\end{array}
$$

Figure 23.2 A probabilistic context-free grammar (PCFG) and lexicon for a portion of the $\mathcal{E}_0$ grammar. The numbers in square brackets indicate the probability that a left-hand-side symbol will be rewritten with the corresponding rule.

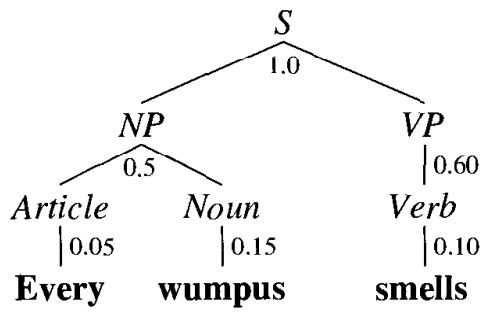


Figure 23.3 Parse tree for the sentence "Every wumpus smells," showing the probabilities of each subtree. The probability of the tree as a whole is $1.0 \times 0.5 \times 0.05 \times 0.15 \times 0.60 \times 0.10 = 0.000225$. Since this tree is the only parse of the sentence, that number is also the probability of the sentence.

containing phrase. With enough training data, we could have the rule for $VP \rightarrow VP\ NP$ be conditioned on the head of the embedded VP (eat) and on the head of the NP (banana). Lexicalized PCFGs thus capture some of the co-occurrence restrictions of n-gram models, along with the grammatical restrictions of CFG models.

One more problem is that PCFGs tend to have too strong a preference for shorter sentences. In a corpus such as the *Wall Street Journal*, the average length of a sentence is about

25 words. But a PCFG will usually end up assigning a fairly high probability to rules such as $S \rightarrow NP\ VP$ and $NP \rightarrow Pronoun$ and $VP \rightarrow Verb$. This means that the PCFG will assign fairly high probability to many short sentences, such as "He slept," whereas in the *Journal* we're more likely to see something like "It has been reported by a reliable government source that the allegation that he slept is credible." It seems that the phrases in the *Journal* really are not context-free; instead the writers have an idea of the expected sentence length and use that length as a soft global constraint on their sentences. This is hard to reflect in a PCFG.

Learning probabilities for PCFGs

To create a PCFG, we have all the difficulty of constructing a CFG, combined with the problem of setting the probabilities for each rule. This suggests that **learning** the grammar from data might be better than a knowledge engineering approach. Just as with speech recognition, there are two types of data we might be given: parsed and unparsed. The task is considerably easier if we have data that have been parsed into trees by linguists (or at least by trained native speakers). Creating such a corpus is a big investment, and the largest ones contain "only" about a million words. Given a corpus of trees, we can create a PCFG just by counting (and smoothing): For each nonterminal symbol, just look at all the nodes with that symbol as root, and create rules for each different combination of children in those nodes. For example, if the symbol NP appears 100,000 times, and there are 20,000 instances of NP with the list of children $[NP, PP]$, then create the rule

$$NP \rightarrow NP\ PP\ [0.20]\ .$$

The task is much harder if all we have is unparsed text. First of all, we actually have two problems: learning the structure of the grammar rules and learning the probabilities associated with each rule. (We have the same distinction in learning neural nets or Bayes nets.)

For the moment we will assume that the structure of the rules is given and that we are just trying to learn the probabilities. We can use an expectation–maximization (EM) approach, just as we did in learning HMMs. The parameters we are trying to learn are the rule probabilities. The hidden variables are the parse trees: we don't know whether a string of words $w_i \ldots w_j$ is or is not generated by a rule $X \rightarrow \alpha$. The E step estimates the probability that each subsequence is generated by each rule. The M step then estimates the probability of each rule. The whole computation can be done in a dynamic programming fashion with an algorithm called the **inside–outside algorithm** in analogy to the forward–backward algorithm for HMMs.

INSIDE–OUTSIDE ALGORITHM

The inside–outside algorithm seems magical in that it induces a grammar from unparsed text. But it has several drawbacks. First, it is slow: $O(n^3t^3)$, where n is the number of words in a sentence and t is the number of nonterminal symbols. Second, the space of probability assignments is very large, and empirically it seems that getting stuck in local maxima is a severe problem. Alternatives such as simulated annealing can be tried, at a cost of even more computation. Third, the parses that are assigned by the resulting grammars are often difficult to understand and unsatisfying to linguists. This makes it hard to combine handcrafted knowledge with automated induction.

Learning rule structure for PCFGs

Now suppose the structure of the grammar rules is not known. Our first problem is that the space of possible rule sets is infinite, so we don't know how many rules to consider nor how long each rule can be. One way to sidestep this problem is to learn a grammar in **Chomsky normal form**, which means that every rule is in one of the two forms

CHOMSKY NORMAL FORM

$$X \rightarrow Y\ Z$$
$$X \rightarrow \mathbf{t}\,,$$

where X, Y and Z are nonterminals and $\mathbf{t}$ is a terminal. Any context-free grammar can be rewritten as a Chomsky normal form grammar that recognizes the exact same language. We can then arbitrarily restrict ourself to n non-terminal symbols, thus yielding $n^3 + nv$ rules, where v is the number of terminal symbols. In practice, this approach has proven effective only for small grammars. An alternative approach called **Bayesian model merging** is similar to the SEQUITUR model (Section 22.8). The approach starts by building local models (grammars) of each sentence and then uses minimum description length to merge models.

BAYESIAN MODEL MERGING

23.2 INFORMATION RETRIEVAL

INFORMATION RETRIEVAL

Information retrieval is the task of finding documents that are relevant to a user's need for information. The best-known examples of information retrieval systems are search engines on the World Wide Web. A Web user can type a query such as [AI book] into a search engine and see a list of relevant pages. In this section, we will see how such systems are built. An information retrieval (henceforth IR) system can be characterized by:

1. **A document collection.** Each system must decide what it wants to treat as a document: a paragraph, a page, or a multi-page text.
2. **A query posed in a query language**. The query specifies what the user wants to know. The query language can be just a list of words, such as [AI book]; or it can specify a phrase of words that must be adjacent, as in ["AI book"]; it can contain Boolean operators as in [AI *AND* book]; it can include non-Boolean operators such as [AI *NEAR* book] or [AI book *SITE:*www.aaai.org].

   QUERY
   QUERY LANGUAGE
3. **A result set.** This is the subset of documents that the IR system judges to be **relevant** to the query. By *relevant*, we mean likely to be of use to the person who asked the query, for the particular information need expressed in the query.

   RESULT SET
   RELEVANT
4. **A presentation of the result set.** This can be as simple as a ranked list of document titles or as complex as a rotating color map of the result set projected onto a three-dimensional space.

   PRESENTATION

After reading the previous chapter, one might suppose that an information retrieval system could be built by parsing the document collection into a knowledge base of logical sentences and then parsing each query and ASKing the knowledge base for answers. Unfortunately, no one has ever succeeded in building a large-scale IR system that way. It is just too difficult

to build a lexicon and grammar that cover a large document collection, so all IR systems use simpler language models.

BOOLEAN KEYWORD MODEL

The earliest IR systems worked on a **Boolean keyword model**. Each word in the document collection is treated as a Boolean feature that is true of a document if the word occurs in the document and false if it does not. So the feature "retrieval" is true for the current chapter but false for Chapter 15. The query language is the language of Boolean expressions over features. A document is relevant only if the expression evaluates to true. For example, the query [information *AND* retrieval] is true for the current chapter and false for Chapter 15.

This model has the advantage of being simple to explain and implement. However, it has some disadvantages. First, the degree of relevance of a document is a single bit, so there is no guidance as to how to order the relevant documents for presentation. Second, Boolean expressions might be unfamiliar to users who are not programmers or logicians. Third, it can be hard to formulate an appropriate query, even for a skilled user. Suppose we try [information *AND* retrieval *AND* models *AND* optimization] and get an empty result set. We could try [information *OR* retrieval *OR* models *OR* optimization], but if that returns too many results, it is difficult to know what to try next.

Most IR systems use models based on the statistics of word counts (and sometimes other low-level features). We will explain a probabilistic framework that fits in well with the language models we have covered. The key idea is that, given a query, we want to find the documents that are most likely to be relevant. In other words, we want to compute

$$P(R = true|D, Q)$$

where D is a document, Q is a query, and R is a Boolean random variable indicating relevance. Once we have this, we can apply the probability ranking principle, which says that if we have to present the result set as an ordered list, we should do it in decreasing probability of relevance.

LANGUAGE MODELING

There are several ways to decompose the joint distribution $P(R = true|D, Q)$. We will show the one known as the **language modeling** approach, which estimates a language model for each document and then, for each query, computes the probability of the query, given the document's language model. Using r to denote the value $R = true$, we can rewrite the probability as follows:

$$\begin{aligned} P(r|D,Q) &= P(D,Q|r)P(r)/P(D,Q) && \text{(by Bayes' rule)} \\ &= P(Q|D,r)P(D|r)P(r)/P(D,Q) && \text{(by chain rule)} \\ &= \alpha\, P(Q|D,r)P(r|D)/P(D,Q) && \text{(by Bayes' rule, for fixed} D\text{)} . \end{aligned}$$

We said we were trying to maximize $P(r|D, Q)$, but, equivalently, we can maximize the odds ratio $P(r|D, Q)/P(\neg r|D, Q)$. That is, we can rank documents based on the score:

$$\frac{P(r|D,Q)}{P(\neg r|D,Q)} = \frac{P(Q|D,r)P(r|D)}{P(Q|D,\neg r)P(\neg r|D)} .$$

This has the advantage of canceling out the $P(D, Q)$ term. Now we will make the assumption that for irrelevant documents, the document is independent of the query. In other words, if a document is irrelevant to a query, then knowing the document won't help you figure out what the query is. This assumption can be expressed by

$$P(D, Q|\neg r) = P(D|\neg r)P(Q|\neg r) .$$

With the assumption, we get

$$\frac{P(r|D,Q)}{P(\neg r|D,Q)} = P(Q|D,r) \times \frac{P(r|D)}{P(\neg r|D)} .$$

The factor $P(r|D)/P(\neg r|D)$ is the query-independent odds that the document is relevant. This is a measure of document quality; some documents are more likely to be relevant to *any* query, because the document is just inherently of high quality. For academic journal articles we can estimate the quality by the number of citations, and for web pages we can use the number of hyperlinks to the page. In either case, we might give more weight to high-quality referrers. The age of a document might also be a factor in estimating its query-independent relevance.

The first factor, $P(Q|D,r)$, is the probability of a query given a relevant document. To estimate this probability we must choose a language model of how queries are related to relevant documents. One popular choice is to represent documents with a unigram word model. This is also known as the **bag of words** model in IR, because what matters is the frequency of each word in the document, not their order. In this model the (very short) documents "man bites dog" and "dog bites man" will behave identically. Clearly, they *mean* different things, but it is true that they are both relevant to queries about dogs and biting. Now to calculate the probability of a query given a relevant document, we just multiply the probabilities of the words in the query, according to the document unigram model. This is a **naive Bayes** model of the query. Using Q_j to indicate the jth word in the query, we have

BAG OF WORDS

$$P(Q|D,r) = \prod_j P(Q_j|D,r) .$$

This allows us to make the simplification

$$\frac{P(r|D,Q)}{P(\neg r|D,Q)} = \prod_j P(Q_j|D,r) \frac{P(r|D)}{P(\neg r|D)} .$$

At last, we are ready to apply these mathematical models to an example. Figure 23.4 shows unigram statistics for the words in the query [Bayes information retrieval model] over a document collection consisting of five selected chapters from this book. We assume that the chapters are of uniform quality, so we are interested only in computing the probability of the query given the document, for each document. We do this two times, once using an unsmoothed maximum likelihood estimator D_i and once using a model D'_i with add-one-smoothing. One would expect that the current chapter should be ranked highest for this query, and in fact in either model it is.

The smoothed model has the advantage that it is less susceptible to noise and that it can assign a nonzero probability of relevance to a document that doesn't contain all the words. The unsmoothed model has the advantage that it is easier to compute for collections with many documents: if we create an index that lists which documents mention each word, then we can quickly generate a result set by intersecting these lists, and we need to compute $P(Q|D_i)$ only for those documents in the intersection, rather than for every document.

Evaluating IR systems

How do we know whether an IR system is performing well? We undertake an experiment in which the system is given a set of queries and the result sets are scored with respect to human

Words	Query	Chapter 1 Intro	Chapter 13 Uncertainty	Chapter 15 Time	Chapter 22 NLP	Chapter 23 Current
Bayes	1	5	32	38	0	7
information	1	15	18	8	12	39
retrieval	1	1	1	0	0	17
model	1	9	7	160	9	63
N	4	14680	10941	18186	16397	12574
$P(Q\|D_i, r)$		1.5×10^{-14}	2.8×10^{-13}	0	0	1.2×10^{-11}
$P(Q\|D'_i, r)$		4.1×10^{-14}	7.0×10^{-13}	5.2×10^{-13}	1.7×10^{-15}	1.5×10^{-11}

Figure 23.4 A probabilistic IR model for the query [Bayes information retrieval model] over a document collection consisting of five chapters from this book. We give the word counts for each document–word pair and the total word count N for each document. We use two document models—D_i is an unsmoothed unigram word model of the ith document, and D'_i is the same model with add-one smoothing—and compute the probability of the query given each document for both models. The current chapter (23) is the clear winner, over 200 times more likely than any other chapter under either model.

relevance judgments. Traditionally there have been two measures used in the scoring: recall and precision. We will explain them with the help of an example. Imagine that an IR system has returned a result set for a single query, for which we know which documents are and are not relevant, out of a corpus of 100 documents. The document counts in each category are given in the following table:

	In result set	Not in result set
Relevant	30	20
Not relevant	10	40

PRECISION

Precision measures the proportion of documents in the result set that are actually relevant. In our example, the precision is $30/(30 + 10) = .75$. The false positive rate is $1 - .75 = .25$.

RECALL

Recall measures the proportion of all the relevant documents in the collection that are in the result set. In our example, recall is $30/(30 + 20) = .60$. The false negative rate is $1 - .60 = .40$. In a very large document collection, such as the World Wide Web, recall is difficult to compute, because there is no easy way to examine every page on the web for relevance. The best we can do is either to estimate recall by sampling or to ignore recall completely and just judge precision.

A system can trade off precision against recall. In the extreme, a system that returns every document in the document collection as its result set is guaranteed a recall of 100%, but will have low precision. Alternately, a system could return a single document and have low recall, but a decent chance at 100% precision. One way to summarize this tradeoff is

ROC CURVE

with an **ROC curve**. "ROC" stands for "receiver operating characteristic" (which is not very enlightening). It is a graph measuring the false negative rate on the y axis and false positive rate on the x axis, for various tradeoff points. The area under the curve is a summary of the effectiveness of an IR system.

Recall and precision were defined when IR searches were done primarily by librarians who were interested in thorough, scholarly results. Today, most queries (hundreds of millions per day) are done by Internet users who are less interested in thoroughness and more interested in finding an immediate answer. For them, one good measure is the average **reciprocal rank** of the first relevant result. That is, if a system's first result is relevant, it gets a score of 1 on the query, and if the first two are not relevant, but the third is, it gets a score of $1/3$. An alternative measure is **time to answer**, which measures how long it takes a user to find the desired answer to a problem. This gets closest to what we really want to measure, but it has the disadvantage that each experiment requires a fresh batch of human test subjects.

RECIPROCAL RANK

TIME TO ANSWER

IR refinements

The unigram word model treats all words as completely independent, but we know that some words are correlated: "couch" is closely related to both "couches" and "sofa." Many IR systems attempt to account for these correlations.

For example, if the query is [couch], it would be a shame to exclude from the result set those documents that mention "COUCH" or "couches" but not "couch." Most IR systems do **case folding** of "COUCH" to "couch," and many use a **stemming** algorithm to reduce "couches" to the stem form "couch." This typically yields a small increase in recall (on the order of 2% for English). However, it can harm precision. For example, stemming "stocking" to "stock" will tend to decrease precision for queries about either foot coverings or financial instruments, although it could improve recall for queries about warehousing. Stemming algorithms based on rules (e.g. remove "-ing") cannot avoid this problem, but newer algorithms based on dictionaries (don't remove "-ing" if the word is already listed in the dictionary) can. While stemming has a small effect in English, it is more important in other languages. In German, for example, it is not uncommon to see words like "Lebensversicherungsgesellschaftsangestellter" (life insurance company employee). Languages such as Finnish, Turkish, Inuit, and Yupik have recursive morphological rules that in principle generate words of unbounded length.

CASE FOLDING

STEMMING

The next step is to recognize **synonyms**, such as "sofa" for "couch." As with stemming, this has the potential for small gains in recall, but with a danger of decreasing precision if applied too aggressively. Those interested in football player Tim Couch would not want to wade through results about sofas. The problem is that "languages abhor absolute synonymy just as nature abhors a vacuum" (Cruse, 1986). That is, anytime there are two words that mean the same thing, speakers of the language conspire to modify the meanings to remove the confusion.

SYNONYMS

Many IR systems use word **bigrams** to some extent, although few implement a complete probabilistic bigram model. **Spelling correction** routines can be used to correct for errors in both documents and queries.

SPELLING CORRECTION

As a final refinement, IR can be improved by considering **metadata**—data outside of the text of the document. Examples include human-supplied keywords and hypertext links between documents.

METADATA

Presentation of result sets

The probability ranking principle says to take a result set and present it to the user as a list ordered by probability of relevance. This makes sense if a user is interested in finding all the relevant documents as quickly as possible. But it runs into trouble because it doesn't consider *utility*. For example, if there are two copies of the most relevant document in the collection, then once you have seen the first, the second has equal relevance, but zero utility. Many IR systems have mechanisms for eliminating results that are too similar to previous results.

RELEVANCE FEEDBACK

One of the most powerful ways to improve the performance of an IR system is to allow for **relevance feedback**—feedback from the user saying which documents from an initial result set are relevant. The system can then present a second result set of documents that are similar to those.

DOCUMENT CLASSIFICATION

DOCUMENT CLUSTERING

An alternative approach is to present the result set as a *labeled tree* rather than an ordered list. With **document classification**, the results are classified into a preexisting taxonomy of topics. For example, a collection of news stories might be classified into World News, Local news, Business, Entertainment, and Sports. With **document clustering**, the tree of categories is created from scratch for each result set. Classification is appropriate when there are a small number of topics in a collection, and clustering is appropriate for broader collections such as the World Wide Web. In either case, when the user issues a query, the result set is shown organized into folders based on the categories.

Classification is a supervised learning problem, and as such, it can be attacked with any of the methods from Chapter 18. One popular approach is decision trees. Given a training set of documents labeled with the correct categories, we could build a single decision tree whose leaves assign the document to the proper category. This works well when there are only a few categories, but for larger category sets we will build one decision tree for each category, with the leaves labeling the document as either a member or a nonmember of the category. Usually, the features tested at each node are individual words. For example, a node in the decision tree for the "Sports" category might test for the presence of the word "basketball." Boosted decision trees, naive Bayes models, and support vector machines have all been used to classify text; in many cases accuracy is in the 90–98% range for Boolean classification.

Clustering is an unsupervised learning problem. In Section 20.3 we saw how the EM algorithm can be used to improve an initial estimate of a clustering, based on a mixture of Gaussians model. The task of clustering documents is harder because we don't know that the data were generated by a nice Gaussian model and because we are dealing with a much higher dimensional space. A number of approaches have been developed.

AGGLOMERATIVE CLUSTERING

Agglomerative clustering creates a tree of clusters going all the way down to individual documents. The tree can be pruned at any level to yield a smaller number of categories, but that is considered outside the algorithm. We begin by considering each document as a separate cluster. Then we find the two clusters that are closest to each other according to some distance measure and merge these two clusters into one. We repeat the process until one cluster remains. The distance measure between two documents is some measure of the overlap between the words in the documents. For example, we could represent a document by a vector of word counts, and define the distance as the Euclidean distance between two

vectors. The distance measure between two clusters can be the distance to the median of the cluster, or it can take into account the average distance between members of the clusters. Agglomerative clustering takes time $O(n^2)$, where n is the number of documents.

K-MEANS CLUSTERING

K-means clustering creates a flat set of exactly k categories. It works as follows:

1. Pick k documents at random to represent the k categories.
2. Assign every document to the closest category.
3. Compute the mean of each cluster and use the k means to represent the new values of the k categories.
4. Repeat steps (2) and (3) until convergence.

K-means takes time $O(n)$, giving it one advantage over agglomerative clustering. It is often reported to be less accurate than agglomerative clustering, although some have reported that it can do almost as well (Steinbach *et al.*, 2000).

Regardless of the clustering algorithm used, there is one more task before a clustering can be used to present a result set: finding a good way of describing the cluster. In classification, the category names are already defined (e.g. "Earnings"), but in clustering we need to invent the category names. One way to do that is to choose a list of words that are representative of the cluster. Another option is to choose the title of one or more documents near the center of the cluster.

Implementing IR systems

So far, we have defined how IR systems work in the abstract, but we haven't explained how to make them efficient so that a Web search engine can return the top results from a multi-billion-page collection in a tenth of a second. The two key data structures for any IR system are the lexicon, which lists all the words in the document collection, and the inverted index, which lists all the places where each word appears in the document collection.

The **lexicon** is a data structure that supports one operation: given a word, it returns the location in the inverted index that stores the occurrences of the word. In some implementations it also returns the total number of documents that contain the word. The lexicon should be implemented with a hash table or similar data structure that allows this lookup to be fast. Sometimes a set of common words with little information content will be omitted from the lexicon. These **stop words** ("the," "of," "to," "be," "a," etc.) take up space in the index and don't improve the scoring of results. The only good reason for keeping them in the lexicon is for systems that support phrase queries: an index of stop words is necessary to efficiently retrieve hits for queries such as "to be or not to be."

STOP WORDS

INVERTED INDEX

HIT LISTS

The **inverted index**,[3] like the index in the back of this book, consists of a set of **hit lists**: places where each word occurs. For the Boolean keyword model, a hit list is just a list of documents. For the unigram model, it is a list of (document, count) pairs. To support phrase search, the hit list must also include the positions within each document where the word occurs.

[3] The term "inverted index" is redundant; a better term would be just "index." It is inverted in the sense that it is in a different order than the words in the text, but that is what all indices are like. But "inverted index" is the traditional term in IR.