

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.

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.

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).

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) gives the agent some indication that its behavior is undesirable. Reinforcement learning typically includes the subproblem of learning how the environment works.

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**.

SUPERVISED
LEARNING

UNSUPERVISED
LEARNING

REINFORCEMENT
LEARNING

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

EXAMPLE

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:

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

HYPOTHESIS

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.

GENERALIZATION

PROBLEM OF
INDUCTION

HYPOTHESIS SPACE

CONSISTENT

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

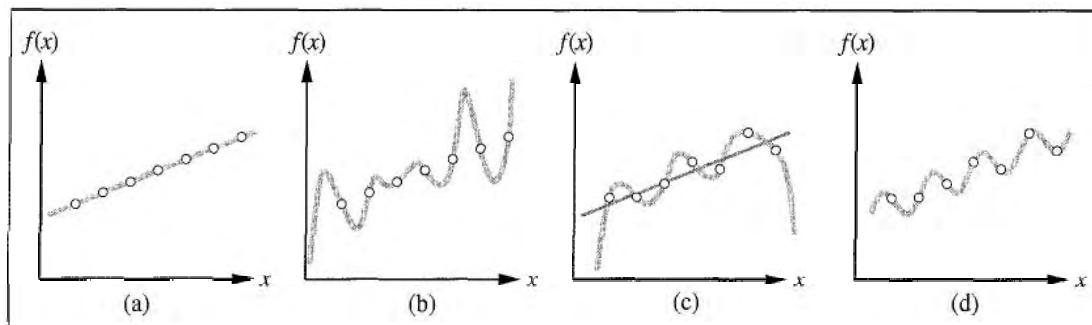


Figure 18.1 (a) Example $(x, f(x))$ pairs and a consistent, linear hypothesis. (b) A consistent, degree-7 polynomial hypothesis for the same data set. (c) A different data set that admits an exact degree-6 polynomial fit or an approximate linear fit. (d) A simple, exact sinusoidal fit to the same data 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's² 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.

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.

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."



OCKHAM'S RAZOR



REALIZABLE
UNREALIZABLE



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

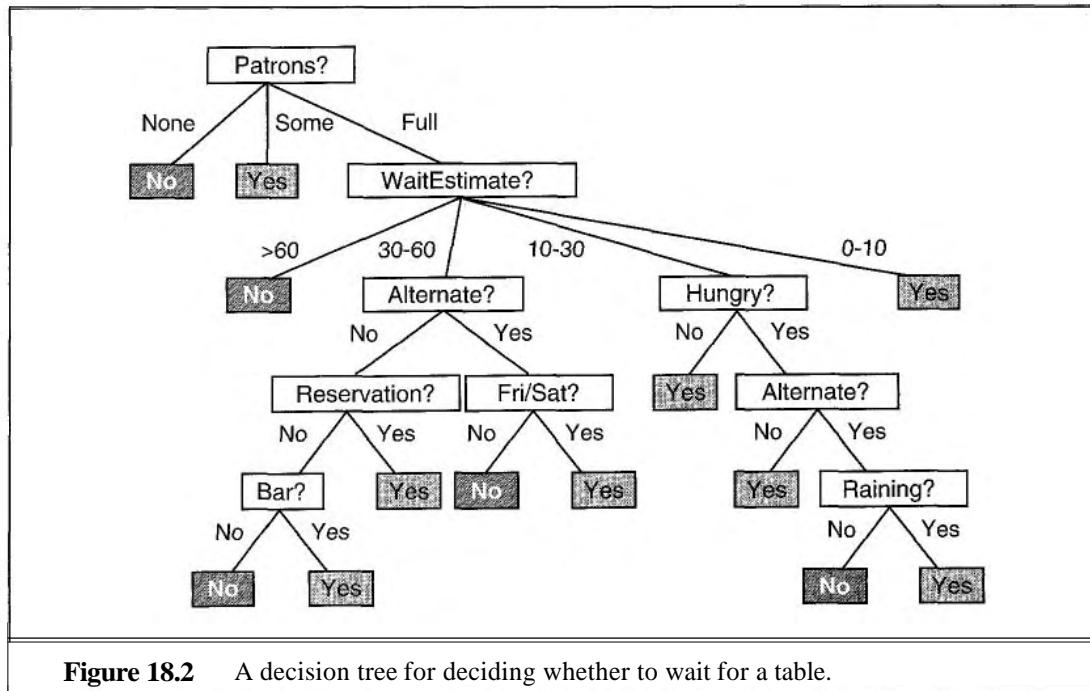
A decision tree takes as input an object or situation described by a set of attributes and returns a "decision"—the predicted output value for the input. The input attributes can be discrete or continuous. For now, we assume discrete inputs. The output value can also be discrete or continuous; learning 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.

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).



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

$$\forall s \text{ } \textit{WillWait}(s) \Leftrightarrow (P_1(s) \vee P_2(s) \vee \dots \vee P_n(s)) ,$$

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,

$$\exists r_2 \text{ } \textit{Nearby}(r_2, r) \wedge \textit{Price}(r, p) \wedge \textit{Price}(r_2, p_2) \wedge \textit{Cheaper}(p_2, p)$$

(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.

PARTY 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.

MAJORITY FUNCTION

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, \mathbf{X} , and a single Boolean output value y . A set of examples $(\mathbf{X}_1, y_1), \dots, (\mathbf{X}_{12}, y_{12})$ is shown in Figure 18.3.

Example	Attributes										Goal <i>WillWait</i>
	Alt	Bar	Fri	Hun	Pat	Price	Rain	Res	Type	Est	
X_1	Yes	No	No	Yes	Some	\$\$\$	No	Yes	French	0–10	Yes
X_2	Yes	No	No	Yes	Full	\$	No	No	Thai	3040	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.

TRAINING SET

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

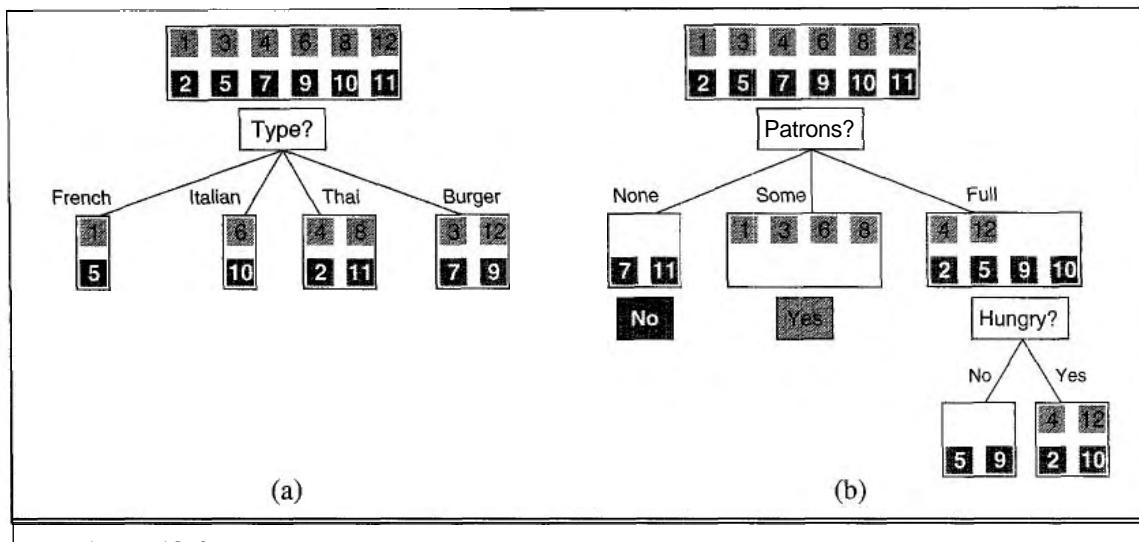


Figure 18.4 Splitting the examples by testing on attributes. (a) Splitting on *Type* brings us no nearer to distinguishing between positive and negative examples. (b) Splitting on *Patrons* does a good job of separating positive and negative examples. After splitting on *Patrons*, *Hungry* is a fairly 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, attrs, default) returns a decision tree
  inputs: examples, set of examples
           attrzbs, 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 attrzbs is empty then return MAJORITY-VALUE(examples)
  else
    best  $\leftarrow$  CHOOSE-ATTRIBUTE(attrs, examples)
    tree  $\leftarrow$  a new decision tree with root test best
    m  $\leftarrow$  MAJORITY-VALUE(examples)
    for each value vi of best do
      examplesi  $\leftarrow$  {elements of examples with best = vi}
      subtree  $\leftarrow$  DECISION-TREE-LEARNING(examplesi, attrs - best, m)
      add a branch to tree with label vi and subtree subtree
  return tree

```

Figure 18.5 The decision tree learning algorithm.

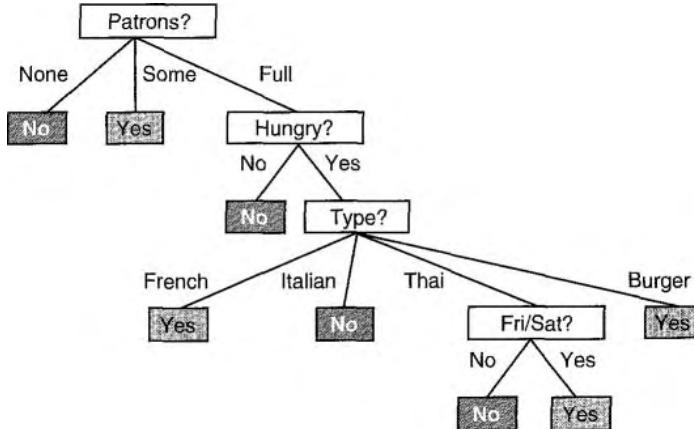


Figure 18.6 The decision tree induced from the 12-example training set.

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

$$I(P(v_1), \dots, P(v_n)) = \sum_{i=1} -P(v_i) \log_2 P(v_i)$$

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

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

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

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

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, \dots, 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

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

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

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:

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

TEST SET

LEARNING CURVE

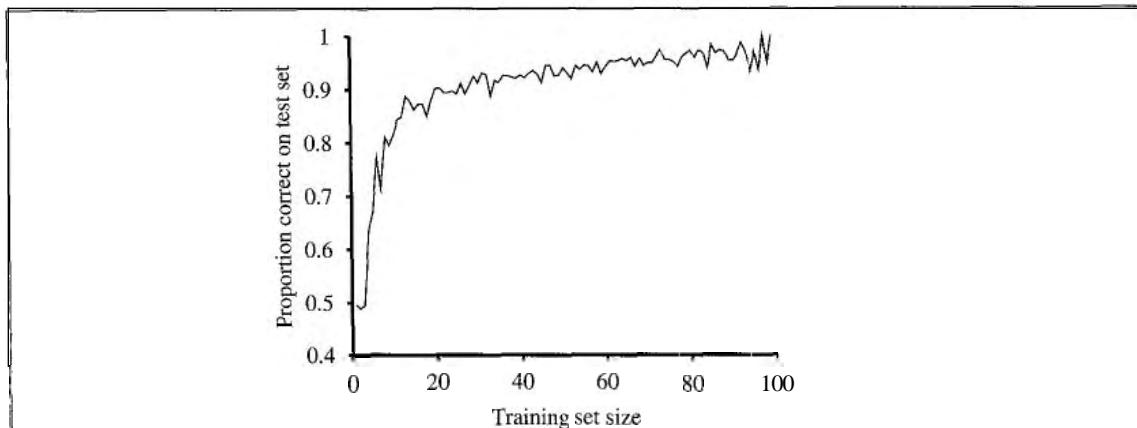


Figure 18.7 A learning curve for the decision tree algorithm on 100 randomly generated examples in the restaurant domain. The graph summarizes 20 trials.

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.

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?

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.

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:

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

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.

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).

◇ **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 $\text{Weight} > 160$ 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.

◇ **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).

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

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

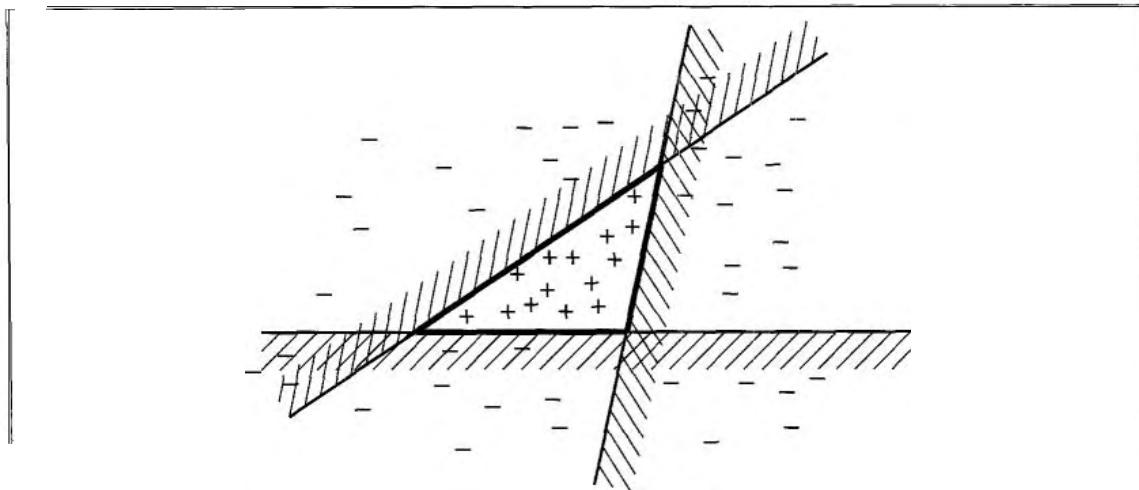


Figure 18.8 Illustration of the increased expressive power obtained by ensemble learning. We take three linear threshold hypotheses, each of which classifies positively on the non-shaded side, and classify as positive any example classified positively by all three. The resulting triangular region is 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 i th example appears w_j times, using randomization to handle fractional weights.

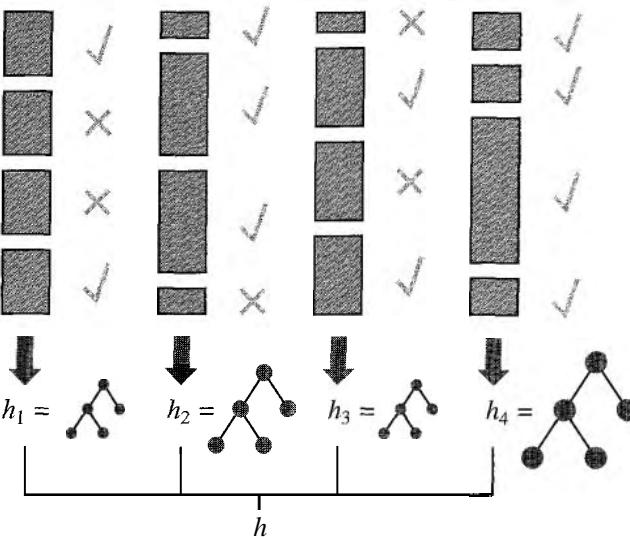


Figure 18.9 How the boosting algorithm works. Each shaded rectangle corresponds to an example; the height of the rectangle corresponds to the weight. The ticks and crosses indicate whether the example was classified correctly by the current hypothesis. The size of the decision tree indicates the weight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

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.

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

WEAK LEARNING

DECISION STUMP

```

function ADABOOST(examples, L, M) returns a weighted-majority hypothesis
  inputs: examples, set of N labelled examples  $(x_1, y_1), \dots, (x_N, y_N)$ 
          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]  $\leftarrow$  L(examples, w)
    error  $\leftarrow$  0
    for j = 1 to N do
      if h[m]( $x_j$ )  $\neq y_j$  then error  $\leftarrow$  error + w[j]
    for j = 1 to N do
      if h[m]( $x_j$ ) =  $y_j$  then w[j]  $\leftarrow$  w[j] . error/(1 - error)
    w  $\leftarrow$  NORMALIZE(w)
    z[m]  $\leftarrow$  log(1 - error)/error
  return WEIGHTED-MAJORITY(~z)

```

Figure 18.10 The ADABOOST variant of the boosting method for ensemble learning. The algorithm generates hypotheses by successively reweighting the training 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.

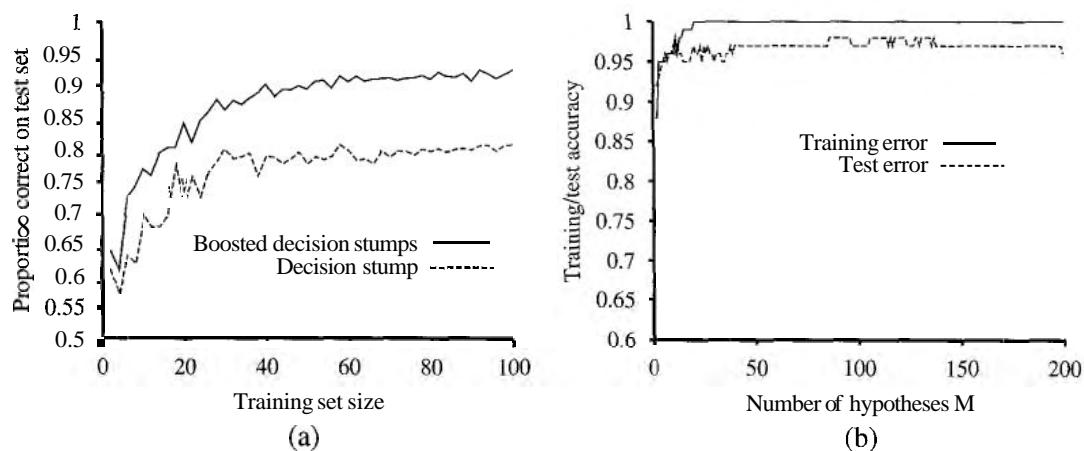


Figure 18.11 (a) Graph showing the performance of boosted decision stumps with $M = 5$ versus decision stumps on the restaurant data. (b) The proportion correct on the training set and the test set as a function of M, the number of hypotheses in the ensemble. Notice that the test set accuracy improves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

COMPUTATIONAL
LEARNING THEORY



PROMPTLY
APPROXIMATELY
CORRECT

PAC-LEARNING

STATIONARITY

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.

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.

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.

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.

ERROR

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:

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

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

 ϵ -BALL

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 H_{bad} .

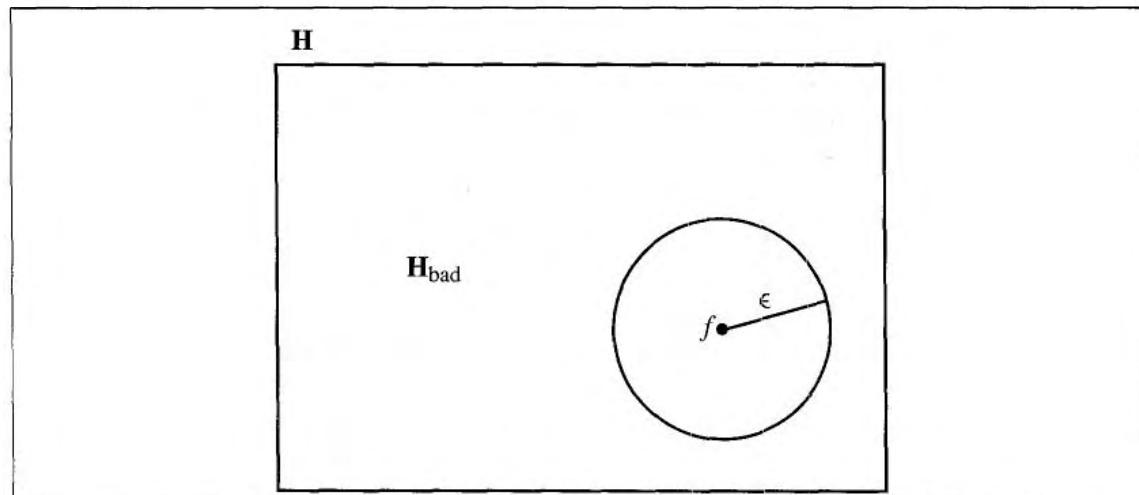


Figure 18.12 Schematic diagram of hypothesis space, showing the " ϵ -ball" around the true function f .

We can calculate the probability that a "seriously wrong" hypothesis $h_b \in H_{\text{bad}}$ 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 H_{bad} contains at least one consistent hypothesis is bounded by the sum of the individual probabilities:

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

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 6:

$$|\mathbf{H}|(1 - \epsilon)^N \leq 6.$$

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

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

examples. Thus, if a learning algorithm returns a hypothesis that is consistent with this many examples, then with probability at least $1 - 6$, 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 6, is called the **sample complexity** of the hypothesis space.

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.

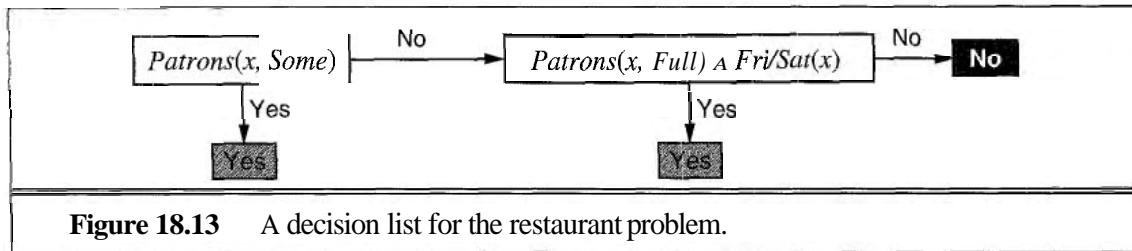
Learning decision lists

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.



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

The first task is to show that $k\text{-DL}$ is learnable—that is, that any function in $k\text{-DL}$ 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 $\text{Conj}(n, k)$. 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 $3^{|\text{Conj}(n, k)|}$ distinct sets of component tests. Each of these sets of tests can be in any order, so

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

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

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

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 $k\text{-DL}$ 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 $k\text{-DL}$ 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
   $t \leftarrow$  a test that matches a nonempty subset examples, of examples
    such that the members of examples, are all positive or all negative
  if there is no such t then return failure
  if the examples in examples, are positive then  $o \leftarrow Yes$  else  $o \leftarrow No$ 
  return a decision list with initial test t and outcome o and remaining tests given by
    DECISION-LIST-LEARNING(examples – examples,)

```

Figure 18.14 An algorithm for learning decision lists.

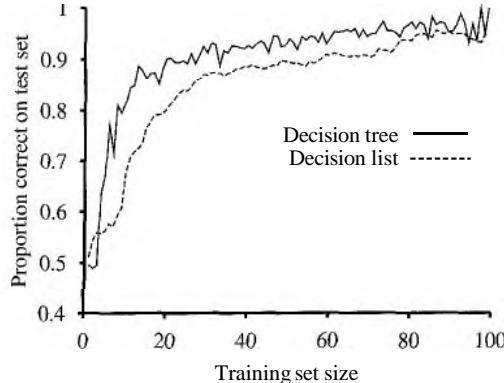


Figure 18.15 Learning curve for DECISION-LIST-LEARNING algorithm on the restaurant data. The curve for DECISION-TREE-LEARNING is shown for comparison.

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

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 contributor 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).

BAGGING

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.

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).

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.

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).

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 $\ln |\mathcal{H}|$ 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).

KOLMOGOROV COMPLEXITY

MINIMUM DESCRIPTION LENGTH

UNIFORM CONVERGENCE THEORY
VC DIMENSION

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

CLASS PROBABILITY

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

PRIOR KNOWLEDGE

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.

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 i th example X_i . For instance, the first example from Figure 18.3 is described by the sentences

$$\text{Alternate}(X_1) \wedge \neg\text{Bar}(X_1) \wedge \neg\text{Fri/Sat}(X_1) \wedge \text{Hungry}(X_1) \wedge \dots$$

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

$$\text{WillWait}(X_1).$$

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 $\forall x \ Q(x) \Leftrightarrow C_i(x)$. 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):

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

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.

The hypothesis space H is the set of all hypotheses $\{H_1, \dots, H_n\}$ 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

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

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:

- 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 $\text{Patrons}(X_{13}, \text{Full}) \wedge \text{Wait}(X_{13}, 0\text{-}10) \wedge \neg\text{Hungry}(X_{13}) \wedge \dots \wedge \text{WillWait}(X_{13})$

CANDIDATE
DEFINITION

EXTENSION

FALSE NEGATIVE

would be a false negative for the hypothesis H_r given earlier. From H_r and the example description, we can deduce both $\text{WillWait}(X_{13})$, which is what the example says, and $\neg \text{WillWait}(X_{13})$, 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-hypothesissearch

CURRENT-BEST-HYPOTHESIS

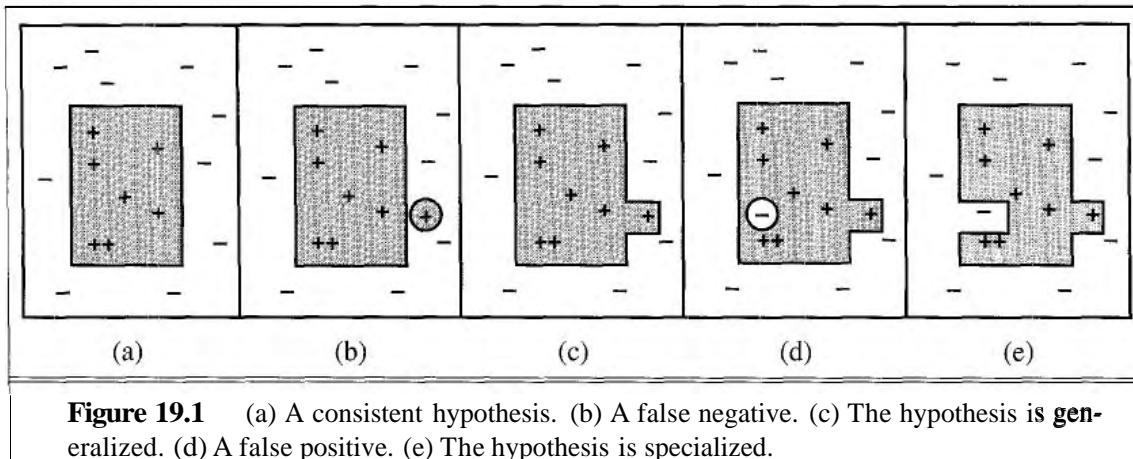
The idea behind **current-best-hypothesis** search is to maintain a single hypothesis, and to adjust it as new examples arrive in order to maintain consistency. The basic algorithm was described by John Stuart Mill (1843), and may well have appeared even earlier.

GENERALIZATION

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.

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.



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  $\leftarrow$  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  $\leftarrow$  choose a specialization of H consistent with examples
        else if e is false negative for H then
            H  $\leftarrow$  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

Alternate(x) *A Patrons*(x , *Some*), then one possible generalization is given by $C_1(x) \equiv \text{Patrons}(x, \text{Some})$. 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

$$H_1 : \forall x \text{ WillWait}(x) \Leftrightarrow \text{Alternate}(x).$$

- 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

$$H_2 : \forall x \text{ WillWait}(x) \Leftrightarrow \text{Alternate}(x) \wedge \text{Patrons}(x, \text{Some})$$

- 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

$$H_3 : \forall x \text{ WillWait}(x) \Leftrightarrow \text{Patrons}(x, \text{Some})$$

- 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:

$$H_4 : \forall x \text{ WillWait}(x) \Leftrightarrow \text{Patrons}(x, \text{Some}) \\ \vee (\text{Patrons}(x, \text{Full}) \wedge \text{Fri/Sat}(x))$$

Already, the hypothesis is starting to look reasonable. Obviously, there are other possibilities consistent with the first four examples; here are two of them:

$$H'_4 : \forall x \text{ WillWait}(x) \Leftrightarrow \neg \text{WaitEstimate}(x, 30-60).$$

$$H''_4 : \forall x \text{ WillWait}(x) \Leftrightarrow \text{Patrons}(x, \text{Some}) \\ \vee (\text{Patrons}(x, \text{Full}) \wedge \text{WaitEstimate}(x, 10-30)).$$

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-commitmentsearch

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

$$H_1 \vee H_2 \vee H_3 \dots \vee H_n .$$

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 \leftarrow the set of all hypotheses

for each example *e* in *examples* **do**

if *V* is not empty **then** *V* \leftarrow VERSION-SPACE-UPDATE(*V, e*)

return *V*

function VERSION-SPACE-UPDATE(*V, e*) **returns** an updated version space

V \leftarrow {*h* \in *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

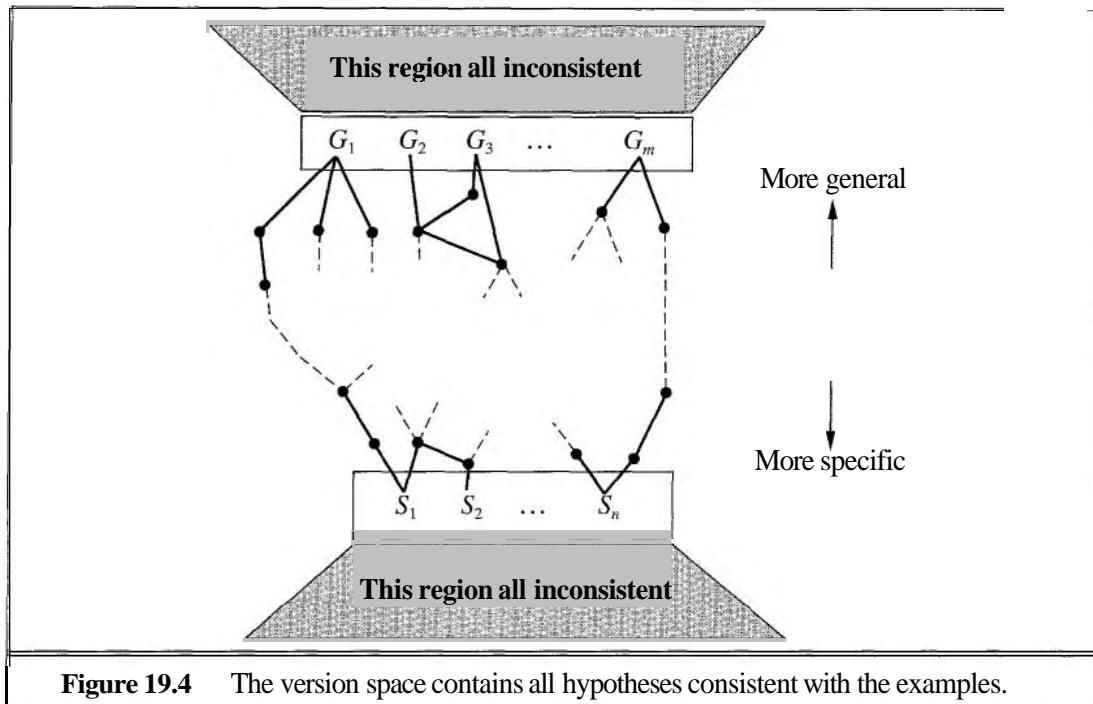


Figure 19.4 The version space contains all hypotheses consistent with the examples.

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.

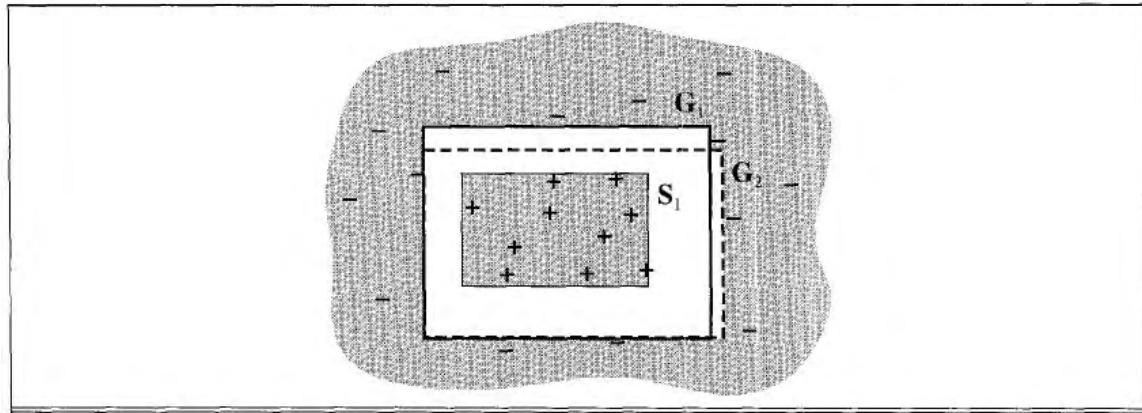


Figure 19.5 The extensions of the members of G and S . No known examples lie in between the two sets of boundaries.

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.

GENERALIZATION HIERARCHY

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 $\text{WaitEstimate}(x, 30\text{-}60) \vee \text{WaitEstimate}(x, >60)$, we might use the single literal $\text{LongWait}(x)$. The set of generalization and specialization operations can be easily extended to handle this.

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

ENTAILMENT
CONSTRAINT

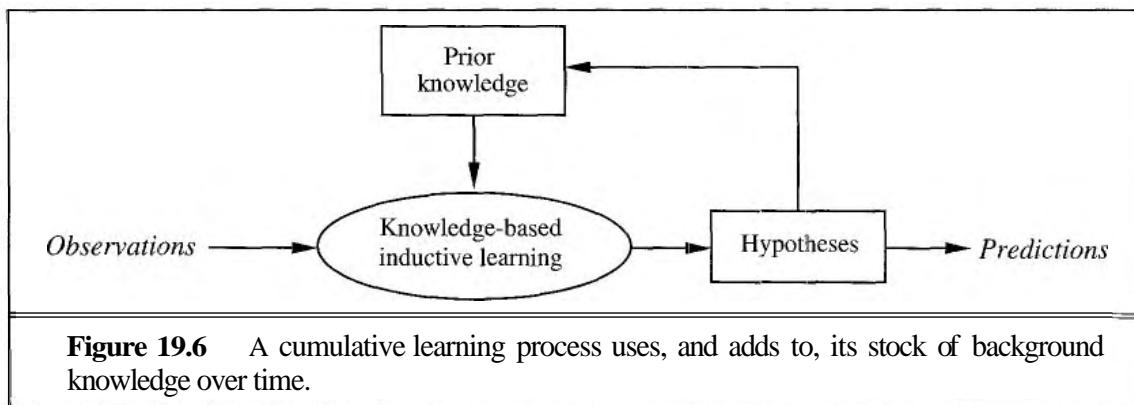
the following property (recall that \models means "logically entails"):

$$\text{Hypothesis A Descriptions} \models \text{Classifications}. \quad (19.3)$$

We call this kind of relationship an **entailment constraint**, in which *Hypothesis* is the “unknown.” Pure inductive learning means solving this constraint, where *Hypotheses* 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 *Hypotheses* = *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.



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:

$$\begin{aligned} \textit{Hypothesis} &\wedge \textit{Descriptions} \models \textit{Classifications} \\ \textit{Background} &\models \textit{Hypothesis} . \end{aligned}$$

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. If she does know, however, that the material of which an object is composed and its temperature together determine its conductance. 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:

$$\begin{aligned} \text{Hypothesis A Descriptions} &\models \text{Classifications}, \\ \text{Background A Descriptions} &\models \text{Hypothesis}. \end{aligned} \tag{19.4}$$

RELEVANCE

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:

$$\text{Background A Hypothesis} \wedge \text{Descriptions} \models \text{Classifications}. \tag{19.5}$$

KNOWLEDGE-BASED
INDUCTIVE
LEARNINGINDUCTIVE LOGIC
PROGRAMMING

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.

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:

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 $\text{ASK}(\text{Derivative}(X^2, X) = d, \text{KB})$, 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 rule² 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

$$\text{Arithmetic Unknown}(u) \Rightarrow \text{Derivative}(u^2, u) = 2u .$$

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 Mathematica*

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 $\text{Derivative}(X^2, X) = 2X$ 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:

$\text{Rewrite}(u, v) \wedge \text{Simplify}(v, w) \Rightarrow \text{Simplify}(u, w)$.
 $\text{Primitive}(u) \Rightarrow \text{Simplify}(u, u)$.
 $\text{ArithmeticUnknown}(u) \Rightarrow \text{Primitive}(u)$.
 $\text{Number}(u) \Rightarrow \text{Primitive}(u)$.
 $\text{Rewrite}(1 \times u, u)$.
 $\text{Rewrite}(0 + u, u)$.

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 $\text{Rewrite}(1 \times u, u)$, the variable x in the subgoal $\text{Rewrite}(x \times (y + z), v)$ must be bound to 1. Similarly, y must be bound to 0 in the subgoal $\text{Rewrite}(y + z, v')$ in order to use the rule $\text{Rewrite}(0 + u, u)$. Once we have the generalized proof tree, we take the leaves (with the necessary bindings) and form a general rule for the goal predicate:

$\text{Rewrite}(1 \times (0 + z), 0 + z) \wedge \text{Rewrite}(0 + z, z) \wedge \text{ArithmeticUnknown}(z)$
 $\Rightarrow \text{Simplify}(1 \times (0 + z), z)$.

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

$\text{ArithmeticUnknown}(z) \Rightarrow \text{Simplify}(1 \times (0 + z), z)$

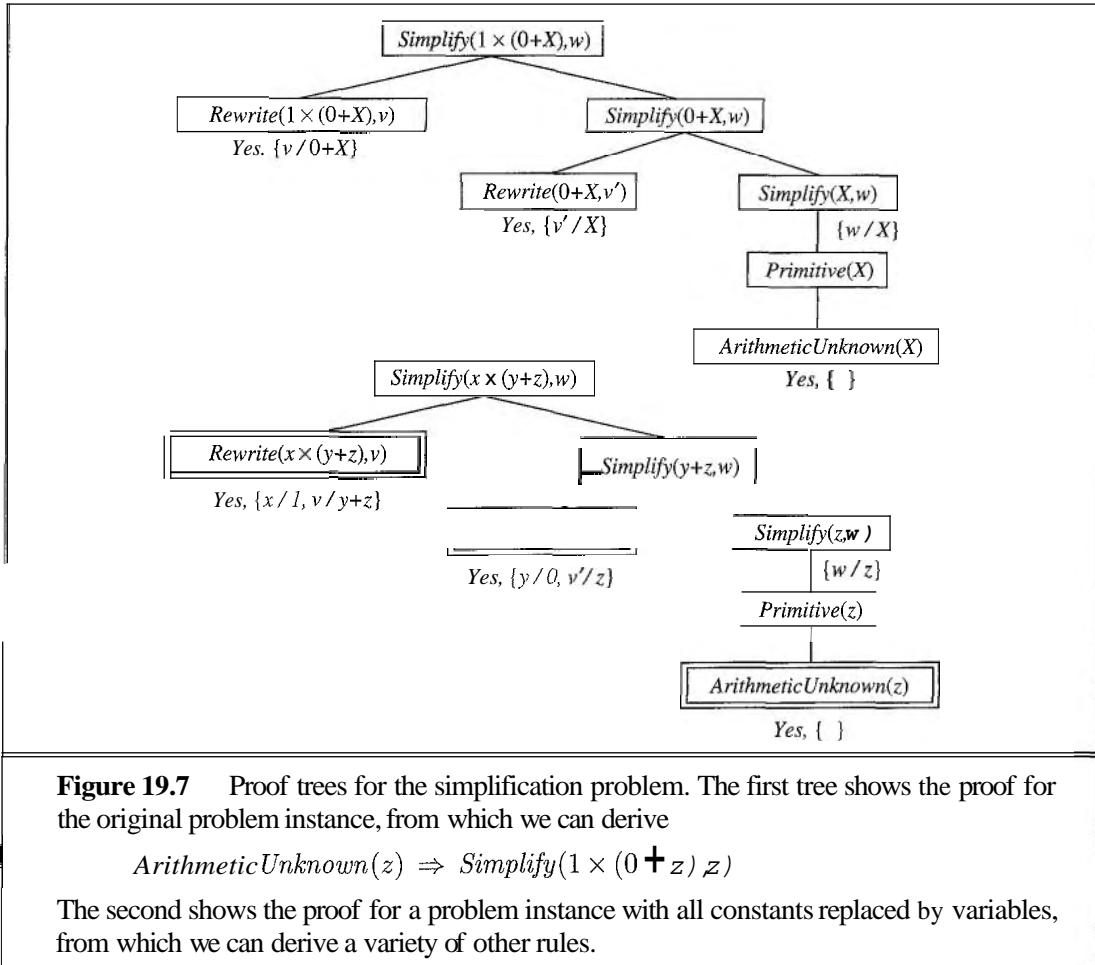


Figure 19.7 Proof trees for the simplification problem. The first tree shows the proof for the original problem instance, from which we can derive

$$\text{ArithmeticUnknown}(z) \Rightarrow \text{Simplify}(1 \times (0 + z), z)$$

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 $\text{ArithmeticUnknown}(z)$, 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

$$\text{Primitive}(z) \Rightarrow \text{Simplify}(1 \ x \ (0 + z), z).$$

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 $\text{Simplify}(y + z, w)$, yielding the rule

$$\text{Simplify}(y + z, w) \Rightarrow \text{Simplify}(1 \ x \ (y + x), w).$$

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.

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 $\text{Primitive}(z)$ is easy to solve, requiring at most two steps, whereas the subgoal $\text{Simplify}(y + z, w)$ 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.

OPERATIONALITY



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

$$\text{Nationality}(x, n) \wedge \text{Nationality}(y, n) \wedge \text{Language}(x, l) \Rightarrow \text{Language}(y, l). \quad (19.6)$$

(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

$$\text{Nationality}(\text{Fernando}, \text{Brazil}) \wedge \text{Language}(\text{Fernando}, \text{Portuguese}),$$

the following conclusion is entailed (see Exercise 19.1):

$$\text{Nationality}(x, \text{Brazil}) \Rightarrow \text{Language}(x, \text{Portuguese}).$$

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):

$$\text{Nationality}(x, n) \succ \text{Language}(x, l)$$

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:

$$\begin{aligned} \text{Material}(x, m) \wedge \text{Temperature}(x, t) &\succ \text{Conductance}(x, p); \\ \text{Material}(x, m) \wedge \text{Temperature}(x, t) &\succ \text{Density}(x, d). \end{aligned}$$

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  $i \leftarrow 0, \dots, n$  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

```

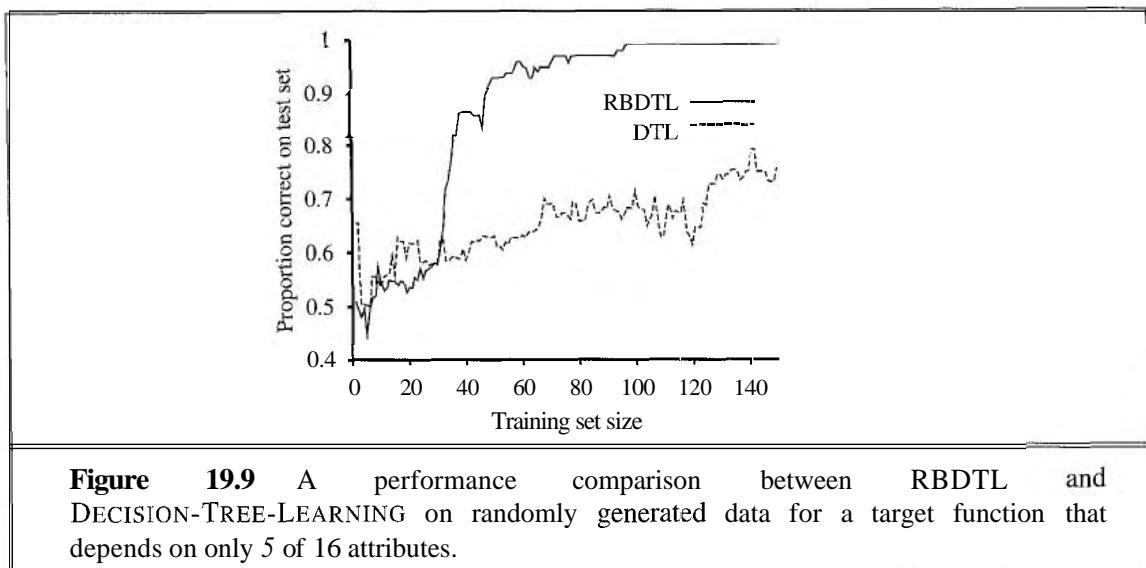
Figure 19.8 An algorithm for finding a minimal consistent determination.

The minimal consistent determination is *Material A Temperature > Conductance*. There is a nonminimal but consistent determination, namely, *Mass A Size A Temperature > 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



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.

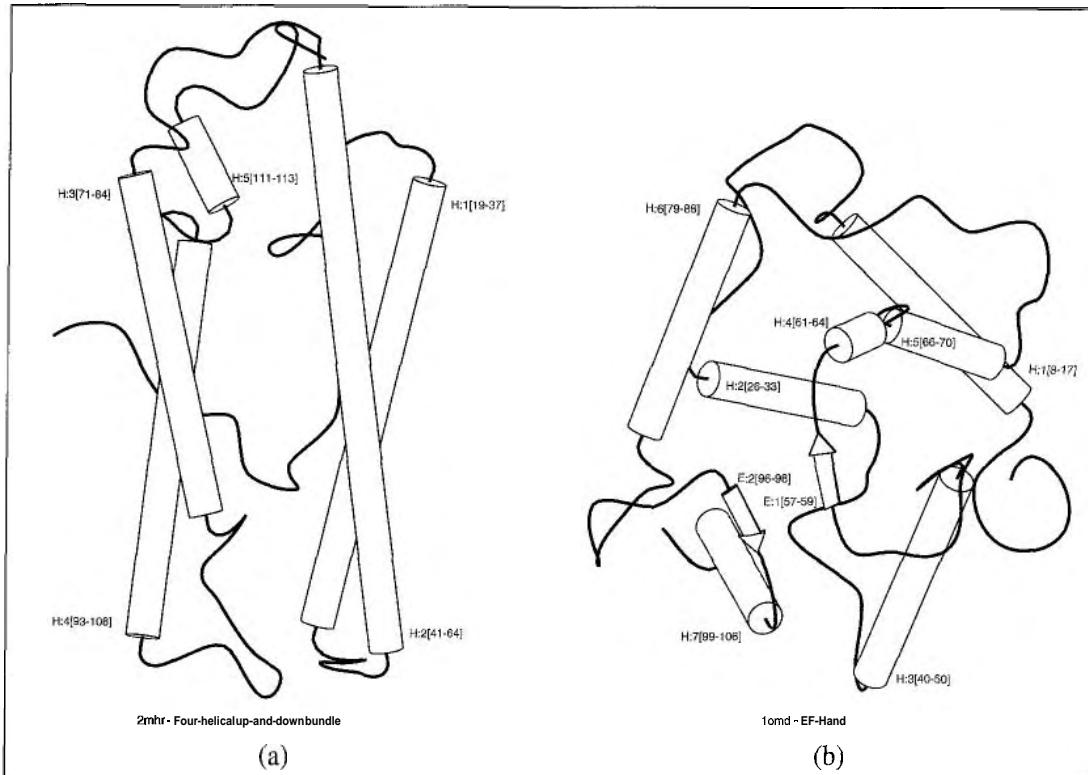


Figure 19.10 (a) and (b) show positive and negative examples, respectively, of the "four-helical up-and-down bundle" concept in the domain of protein folding. Each example structure is coded into a logical expression of about 100 conjuncts such as *TotalLength(D2mhr, 118)* *ANumberHelices(D2mhr, 6)* *A... From these descriptions and from classifications such as Fold(FOUR-HELICAL-UP-AND-DOWN-BUNDLE, D2mhr), the inductive logic programming system PROGOL (Muggleton, 1995) learned the following rule:*

$$\begin{aligned} \text{Fold}(\text{FOUR-HELICAL-UP-AND-DOWN-BUNDLE}, p) \leftarrow \\ \text{Helix}(p, h_1) \wedge \text{Length}(h_1, \text{HIGH}) \wedge \text{Position}(p, h_1, n) \\ \wedge (1 \leq n \leq 3) \wedge \text{Adjacent}(p, h_1, h_2) \wedge \text{Helix}(p, h_2). \end{aligned}$$

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

$$\text{Background } A \text{ Hypothesis } A \text{ Descriptions} \models \text{Classifications}$$

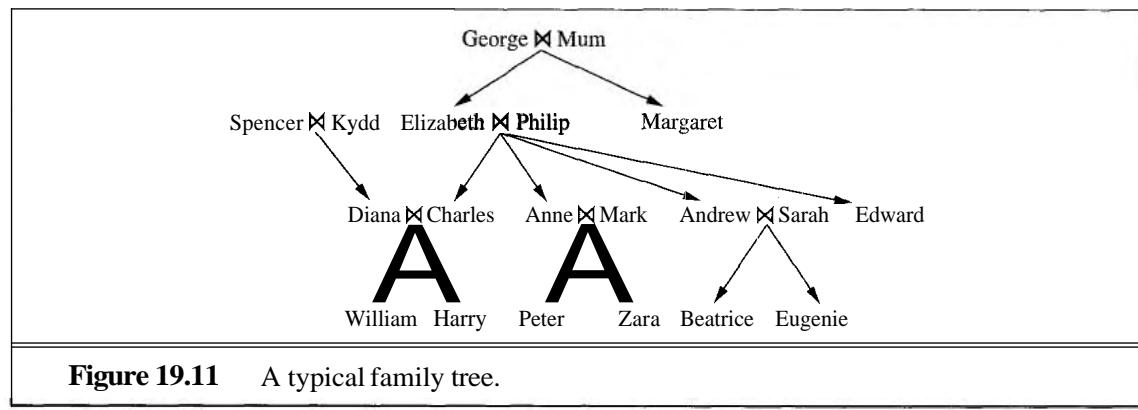
for the unknown *Hypothesis*, given the *Background* knowledge and examples described by *Descriptions* and *Classifications*. To illustrate this, we will use the problem of learning family relationships from examples. The descriptions will consist of an extended family tree, described in terms of *Mother*, *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:

<i>Father(Philip, Charles)</i>	<i>Father(Philip, Anne)</i>	...
<i>Mother(Mum, Margaret)</i>	<i>Mother(Mum, Elizabeth)</i>	...
<i>Married(Diana, Charles)</i>	<i>Married(Elizabeth, Philip)</i>	...
<i>Male(Philip)</i>	<i>Male(Charles)</i>	...
<i>Female(Beatrice)</i>	<i>Female(Margaret)</i>	...

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

$$\begin{aligned} &\text{Grandparent(Mum, Charles)} \quad \text{Grandparent(Elizabeth, Beatrice)} \dots \\ &\neg\text{Grandparent(Mum, Harry)} \quad \neg\text{Grandparent(Spencer, Peter)} \dots \end{aligned}$$

We could of course learn from a subset of this complete set.



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:

$$\begin{aligned} \text{Grandparent}(x, y) & \quad [\exists z \text{ Mother}(x, z) \wedge \text{Mother}(z, y)] \\ \vee & \quad [\exists z \text{ Mother}(x, z) \wedge \text{Father}(z, y)] \\ \vee & \quad [\exists z \text{ Father}(x, z) \wedge \text{Mother}(z, y)] \\ \vee & \quad [\exists z \text{ Father}(x, z) \wedge \text{Father}(z, y)]. \end{aligned}$$

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:

$$\text{Grandparent}(\langle \text{Mum}, \text{Charles} \rangle) \dots$$

Then we get stuck in trying to represent the example descriptions. The only possible attributes are horrible things such as

$$\text{FirstElementIsMotherOfElizabeth}(\langle \text{Mum}, \text{Charles} \rangle).$$

 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

$$\text{Parent}(x, y) \Leftrightarrow [\text{Mother}(x, y) \vee \text{Father}(x, y)],$$

then the definition of *Grandparent* would be reduced to

$$\text{Grandparent}(x, y) \Leftrightarrow [\exists z \text{ Parent}(x, z) \wedge \text{Parent}(z, y)].$$

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.

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 $\text{Grandfather}(x, y)$ 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

$(\text{George}, \text{Anne}), (\text{Philip}, \text{Peter}), (\text{Spencer}, \text{Harry}), \dots$

and negative examples are

$(\text{George}, \text{Elizabeth}), (\text{Harry}, \text{Zara}), (\text{Charles}, \text{Philzp}), \dots$

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 $\text{Grandfather}(x, y)$ as the head. The clauses must classify the 12 positive examples as instances of the $\text{Grandfather}(x, y)$ 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:

$\Rightarrow \text{Grandfather}(x, y).$

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:

$\text{Father}(x, y) \Rightarrow \text{Grandfather}(x, y).$

$\text{Parent}(x, z) \Rightarrow \text{Grandfather}(x, y).$

$\text{Father}(x, z) \Rightarrow \text{Grandfather}(x, y).$

(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 $\text{Parent}(z, y)$ gives

$\text{Father}(x, z) \wedge \text{Parent}(z, y) \Rightarrow \text{Grandfather}(x, y),$

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  $\leftarrow$  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  $\leftarrow$  examples
    while extended-examples contains negative examples do
      l  $\leftarrow$  CHOOSE-LITERAL(NEW-LITERALS(clause), extended-examples)
      append l to the body of clause
      extended-examples  $\leftarrow$  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-LITERAL 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

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 A Hypothesis A 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 $\text{Grandparent}(\text{George}, \text{Anne})$. 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 $\neg\text{Grandparent}(\text{George}, \text{Anne})$, which is the negation of the goal example. The first inverse step takes C and C_2 and generates the clause $\text{Grandparent}(\text{George}, \text{Anne})$ for C_1 . The next step takes this clause as C and the clause $\text{Parent}(\text{Elizabeth}, \text{Anne})$ as C_2 , and generates the clause

$$\neg\text{Parent}(\text{Elizabeth}, y) \vee \text{Grandparent}(\text{George}, y)$$

as C_1 . The final step treats this clause as the resolvent. With $\text{Parent}(\text{George}, \text{Elizabeth})$ as C_2 , one possible clause C_1 is the hypothesis

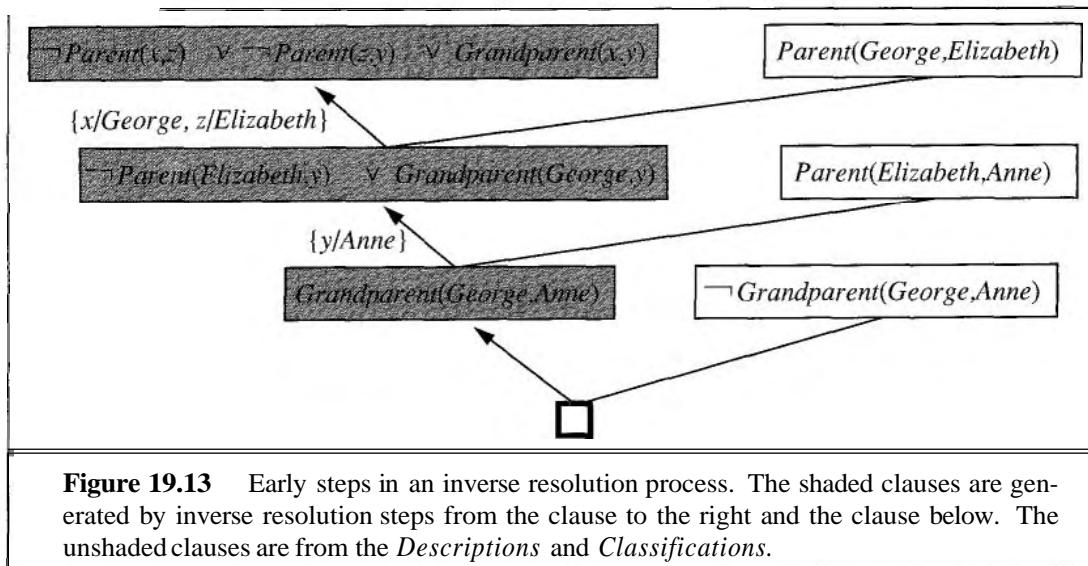
$$\text{Parent}(x, z) \wedge \text{Parent}(z, y) \Rightarrow \text{Grandparent}(x, y)$$

Now we have a resolution proof that the hypothesis, descriptions, and background knowledge entail the classification $\text{Grandparent}(\text{George}, \text{Anne})$.

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 $\neg\text{Parent}(\text{Elizabeth}, y) \vee \text{Grandparent}(\text{George}, y)$ for C_1 in the last step of Figure 19.13, the inverse resolution step might have chosen any of the following sentences:

- $\neg\text{Parent}(\text{Elizabeth}, \text{Anne}) \vee \text{Grandparent}(\text{George}, \text{Anne})$.
- $\neg\text{Parent}(z, \text{Anne}) \vee \text{Grandparent}(\text{George}, \text{Anne})$.
- $\neg\text{Parent}(z, y) \vee \text{Grandparent}(\text{George}, y)$.

(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



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 $\neg\text{Parent}(z, y) \vee \text{Grandparent}(\text{George}, y)$, 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

$$\text{Background } A \text{ Hypothesis } A \text{ Descriptions } \models \text{Classifications}$$

to the logically equivalent form

$$\text{Background } A \text{ Descriptions } A \neg \text{Classifications} \models \neg \text{Hypothesis}.$$

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.

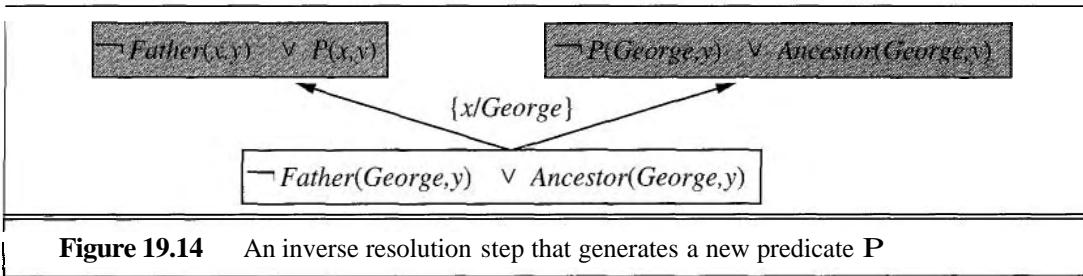


Figure 19.14 An inverse resolution step that generates a new predicate P

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:

- o 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.
- o **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

GRUE

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.

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).

ANALOGICAL
REASONING

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).

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 (Rouveiro 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 (Džeroski *et al.*, 1992). Learnability results for ILP are surveyed by Kietz and Džeroski (1994) and Cohen and Page (1995).

⁵ The inverse resolution method also appears in (Russell, 1986), with a simple algorithm given in a footnote.

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. $C = \text{True} \Rightarrow P(A,B)$ $C_1 = P(x,y) \Rightarrow Q(x,y)$, $C_2 = ??$.
- b. $C = \text{True} \Rightarrow P(A,B)$ $C_1 = ??$, $C_2 = ??$.
- 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 $\text{Resolve}(c_1, c_2, c)$ 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 \mathbf{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, \dots, 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 d ; then the probability of each hypothesis is obtained by Bayes' rule:

$$P(h_i|\mathbf{d}) = \alpha P(\mathbf{d}|h_i)P(h_i). \quad (20.1)$$

Now, suppose we want to make a prediction about an unknown quantity X . Then we have

$$\mathbf{P}(X|\mathbf{d}) = \sum_i \mathbf{P}(X|\mathbf{d}, h_i)\mathbf{P}(h_i|\mathbf{d}) = \sum_i \mathbf{P}(X|h_i)P(h_i|\mathbf{d}), \quad (20.2)$$

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

I.I.D.

For our candy example, we will assume for the time being that the prior distribution over h_1, \dots, 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

$$P(\mathbf{d}|h_i) = \prod_j P(d_j|h_i). \quad (20.3)$$

For example, suppose the bag is really an all-lime bag (h_5) and the first 10 candies are all lime; then $P(\mathbf{d}|h_5)$ 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.

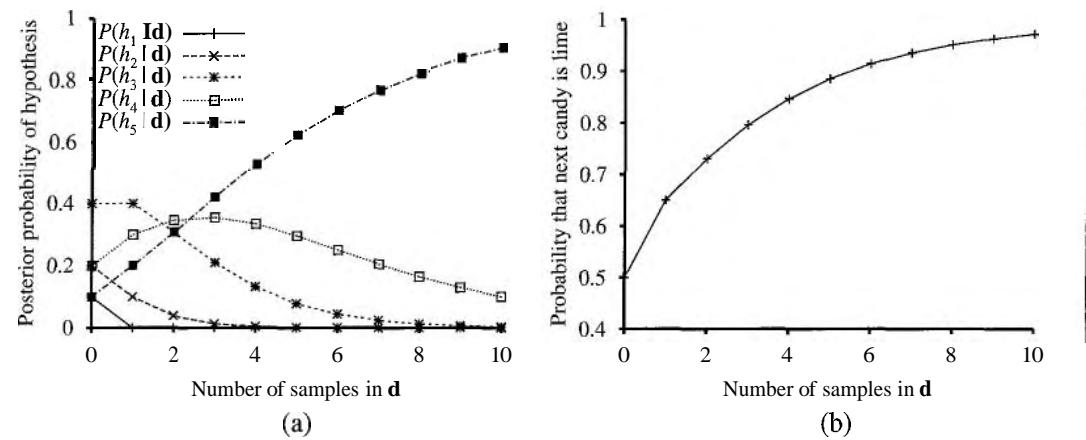


Figure 20.1 (a) Posterior probabilities $P(h_i|d_1, \dots, d_N)$ from Equation (20.1). The number of observations N ranges from 1 to 10, and each observation is of a lime candy. (b) Bayesian prediction $P(d_{N+1} = \text{lime}|d_1, \dots, d_N)$ from Equation (20.2).

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-

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.

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.



MINIMUM
DESCRIPTION
LENGTH

MAXIMUM-
LIKELIHOOD

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 8 (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

$$P(\mathbf{d}|h_\theta) = \prod_{j=1}^N P(d_j|h_\theta) = \theta^c \cdot (1-\theta)^\ell.$$

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:

$$\frac{dL(\mathbf{d}|h_\theta)}{d\theta} = \frac{c}{\theta} - \frac{\ell}{1-\theta} = 0 \quad \Rightarrow \quad \theta = \frac{c}{c+\ell} = \frac{c}{N}.$$

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.

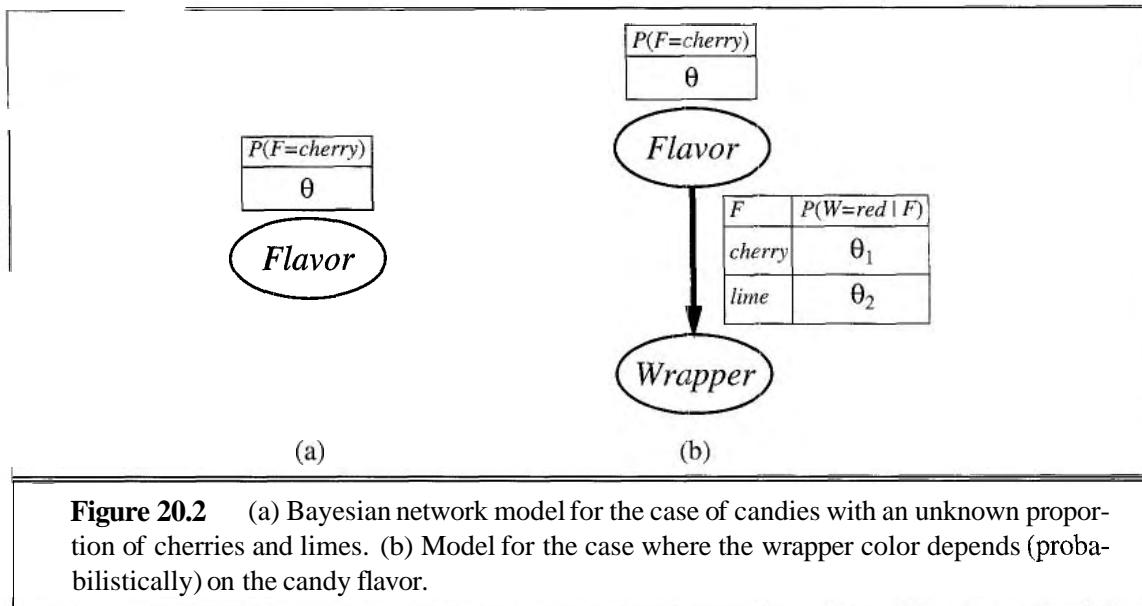


Figure 20.2 (a) Bayesian network model for the case of candies with an unknown proportion of cherries and limes. (b) Model for the case where the wrapper color depends (probabilistically) on the candy flavor.

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):

$$\begin{aligned} P(\text{Flavor} = \text{cherry}, \text{Wrapper} = \text{green} | h_{\theta, \theta_1, \theta_2}) \\ = P(\text{Flavor} = \text{cherry} | h_{\theta, \theta_1, \theta_2}) P(\text{Wrapper} = \text{green} | \text{Flavor} = \text{cherry}, h_{\theta, \theta_1, \theta_2}) \\ = \theta \cdot (1 - \theta_1) . \end{aligned}$$

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

$$P(\mathbf{d} | h_{\theta, \theta_1, \theta_2}) = O^c (l - \theta)^\ell \cdot \theta_1^{r_c} (1 - \theta_1)^{g_c} \cdot \theta_2^{r_\ell} (1 - \theta_2)^{g_\ell} .$$

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:

$$\begin{aligned}\frac{\partial L}{\partial \theta} &= \frac{c}{\theta} - \frac{\ell}{1-\theta} = 0 & \Rightarrow \theta &= \frac{c}{c+\ell} \\ \frac{\partial L}{\partial \theta_1} &= \frac{r_c}{\theta_1} - \frac{g_c}{1-\theta_1} = 0 & \Rightarrow \theta_1 &= \frac{r_c}{r_c+g_c} \\ \frac{\partial L}{\partial \theta_2} &= \frac{r_\ell}{\theta_2} - \frac{g_\ell}{1-\theta_2} = 0 & \Rightarrow \theta_2 &= \frac{r_\ell}{r_\ell+g_\ell}.\end{aligned}$$

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

$$\theta = P(C = \text{true}), \theta_{i1} = P(X_i = \text{true}|C = \text{true}), \theta_{i2} = P(X_i = \text{true}|C = \text{false}).$$

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, \dots, x_n , the probability of each class is given by

$$\mathbf{P}(C|x_1, \dots, x_n) = \alpha \mathbf{P}(C) \prod_i \mathbf{P}(x_i|C).$$

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.

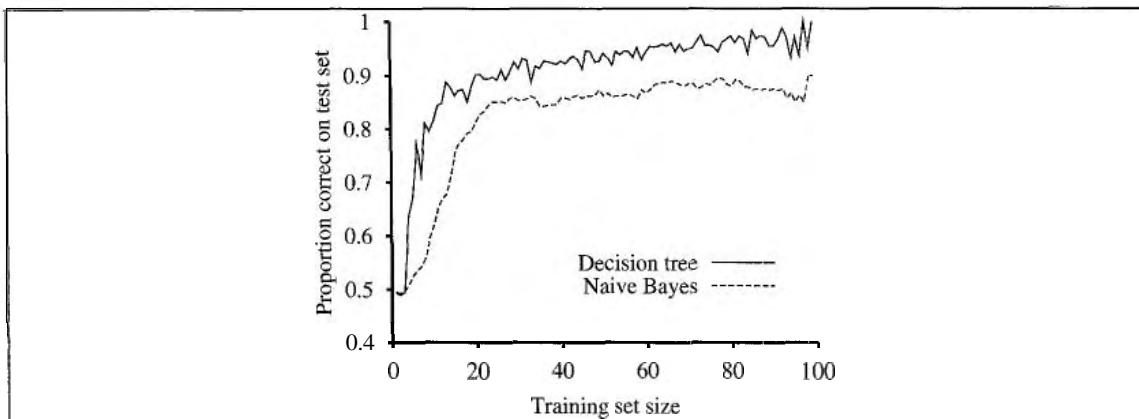


Figure 20.3 The learning curve for naive Bayes learning applied to the restaurant problem from Chapter 18; the learning curve for decision-tree learning is shown for comparison.

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, \dots, x_N . Then the log likelihood is

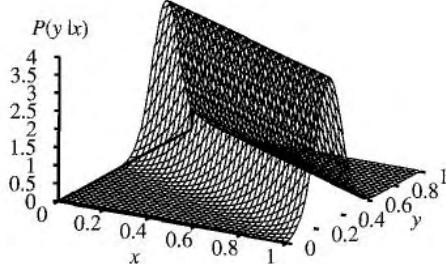
$$L = \sum_{j=1}^N \log \frac{1}{\sqrt{2\pi}\sigma} e^{-\frac{(x_j-\mu)^2}{2\sigma^2}} = N(-\log \sqrt{2\pi} - \log \sigma) - \sum_{j=1}^N \frac{(x_j - \mu)^2}{2\sigma^2}$$

Setting the derivatives to zero as usual, we obtain

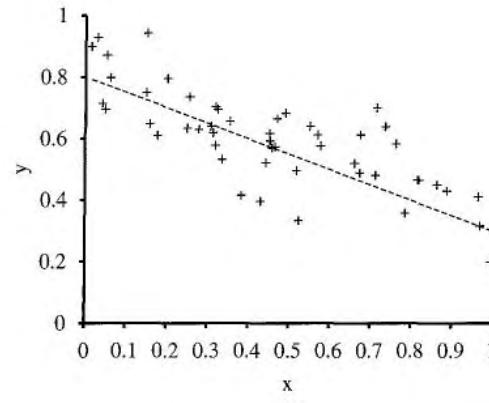
$$\begin{aligned} \frac{\partial L}{\partial \mu} &= -\frac{1}{\sigma^2} \sum_{j=1}^N (x_j - \mu) = 0 & \Rightarrow \quad \mu &= \frac{\sum_j x_j}{N} \\ \frac{\partial L}{\partial \sigma} &= -\frac{N}{\sigma} + \frac{1}{\sigma^3} \sum_{j=1}^N (x_j - \mu)^2 = 0 & \Rightarrow \quad \sigma &= \sqrt{\frac{\sum_j (x_j - \mu)^2}{N}}. \end{aligned} \tag{20.4}$$

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



(a)



(b)

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

$$P(y|x) = \frac{1}{\sqrt{2\pi}\sigma} e^{-\frac{(y-(\theta_1 x + \theta_2))^2}{2\sigma^2}} \quad (20.5)$$

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

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 $(\theta_1 x_j + \theta_2)$ —SOE 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*.

SUM OF SQUARED ERRORS
LINEAR REGRESSION

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.

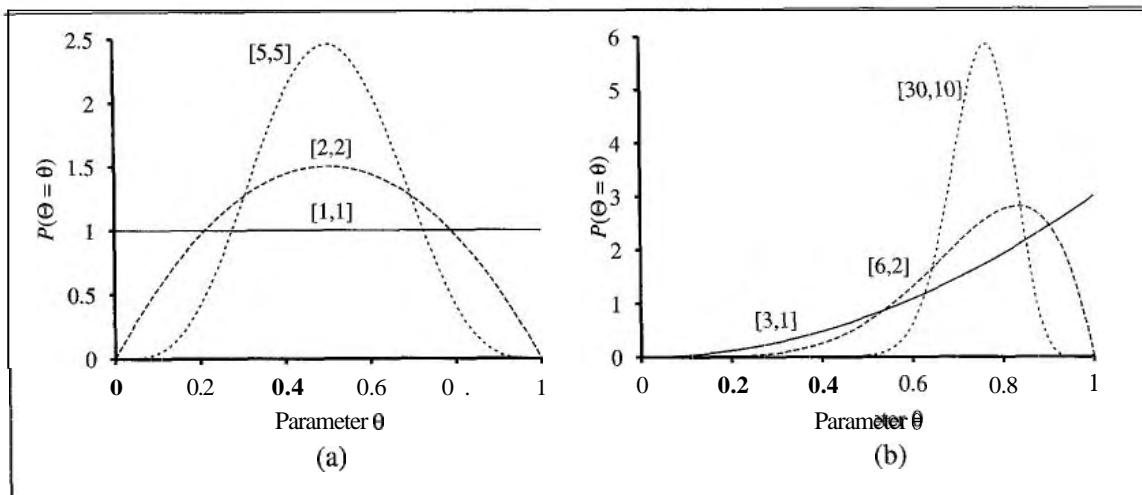


Figure 20.5 Examples of the beta $[a, b]$ distribution for different values of $[a, b]$.

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 $P(\Theta)$. Thus, $P(O=8)$ 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 $P(\Theta)$ 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

$$\text{beta}[a, b](\theta) = a \theta^{a-1} (1 - \theta)^{b-1}, \quad (20.6)$$

for θ in the range $[0, 1]$. The normalization constant a 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

$$\begin{aligned} P(\theta | D_1 = \text{cherry}) &= a P(D_1 = \text{cherry} | \theta) P(\theta) \\ &= a' \theta \cdot \text{beta}[a, b](\theta) = a' \theta \cdot \theta^{a-1} (1 - \theta)^{b-1} \\ &= a' \theta^a (1 - \theta)^{b-1} = \text{beta}[a+1, b](\theta). \end{aligned}$$

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.

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**:

$$\mathbf{P}(\Theta, \Theta_1, \Theta_2) = \mathbf{P}(\Theta)\mathbf{P}(\Theta_1)\mathbf{P}(\Theta_2)$$

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:

$$P(Flavor_i = \text{cherry} | \Theta = \theta) = \theta .$$

Similarly, the wrapper probabilities depend on Θ_1 and Θ_2 , For example,

$$P(Wrapper_i = \text{red} | Flavor_i = \text{cherry}, \Theta_1 = \theta_1) = \theta_1 .$$

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

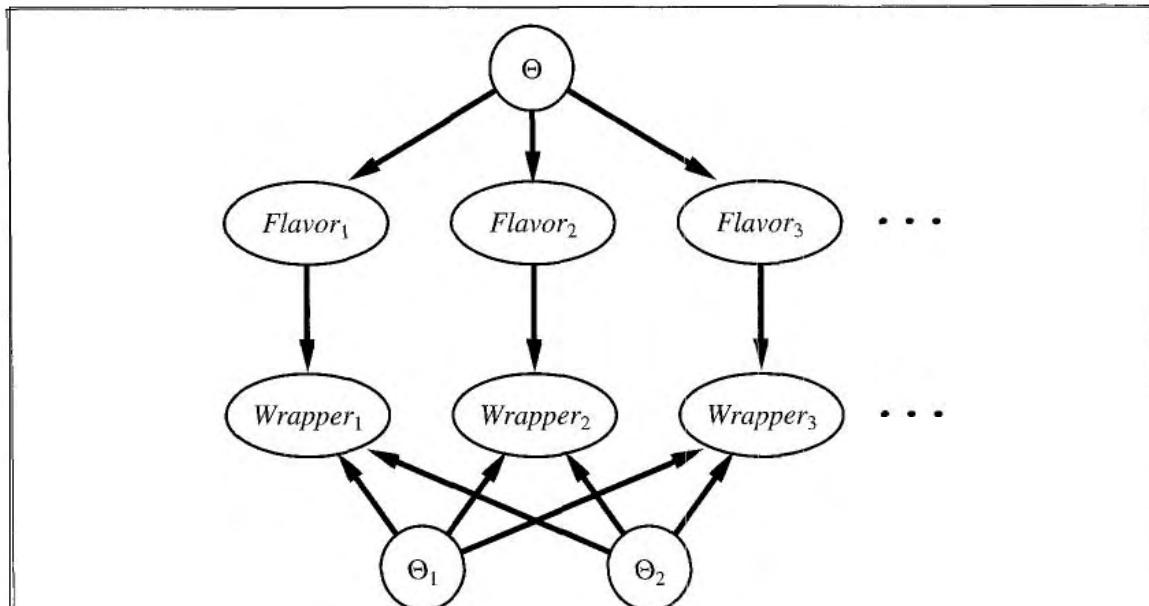


Figure 20.6 A Bayesian network that corresponds to a Bayesian learning process. Posterior distributions for the parameter variables Θ , Θ_1 , and Θ_2 can be inferred from their prior distributions and the evidence in the *Flavor*, and *Wrapper_i* variables.

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

$$\mathbf{P}(Fri/Sat, Bar | WillWait) = \mathbf{P}(Fri/Sat | WillWait)\mathbf{P}(Bar | WillWait)$$

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

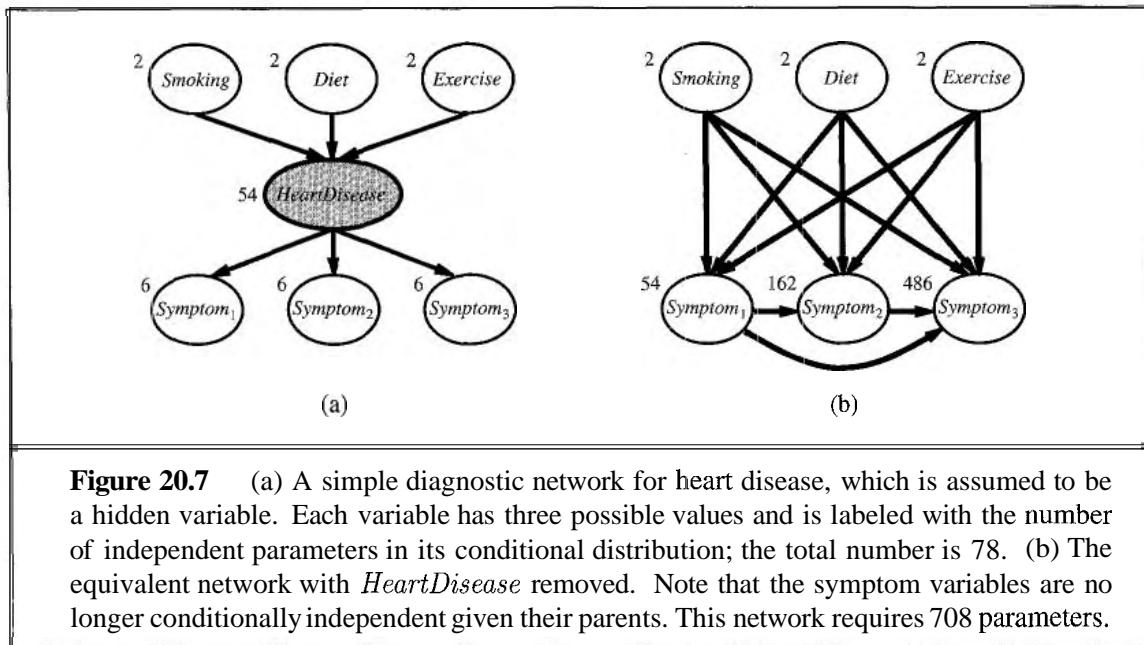
LATENT VARIABLES

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.



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 Linnaean 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 $1, \dots, k$; then the mixture

MIXTURE DISTRIBUTION COMPONENT

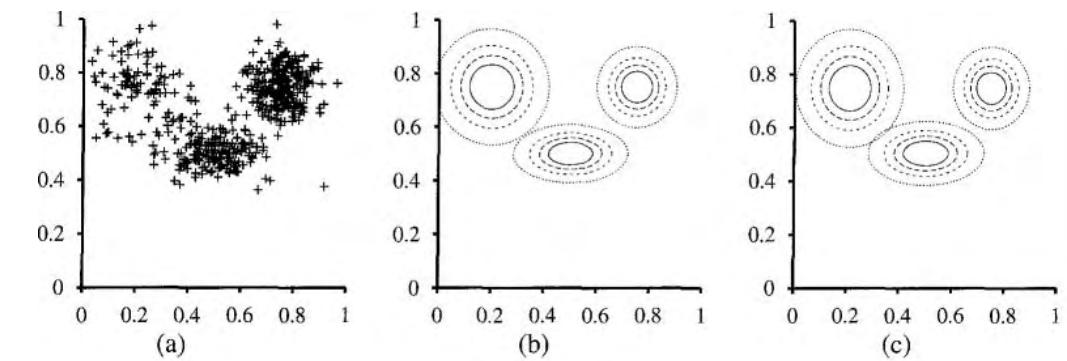


Figure 20.8 (a) 500 data points in two dimensions, suggesting the presence of three clusters. (b) A Gaussian mixture model with three components; the weights (left-to-right) are 0.2, 0.3, and 0.5. The data in (a) were generated from this model. (c) The model reconstructed by EM from the data in (b).

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), μ_i (the mean of each component), and Σ_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).

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 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 i th Gaussian, and the term $P(C=i)$ is just the weight parameter for the i th Gaussian. Define $p_i = \sum_j p_{ij}$.
2. M-step: Compute the new mean, covariance, and component weights as follows:

$$\begin{aligned}\boldsymbol{\mu}_i &\leftarrow \sum_j p_{ij} \mathbf{x}_j / p_i \\ \boldsymbol{\Sigma}_i &\leftarrow \sum_j p_{ij} (\mathbf{x}_j - \boldsymbol{\mu}_i)(\mathbf{x}_j - \boldsymbol{\mu}_i)^\top / p_i \\ w_i &\leftarrow p_i.\end{aligned}$$

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 i th 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.

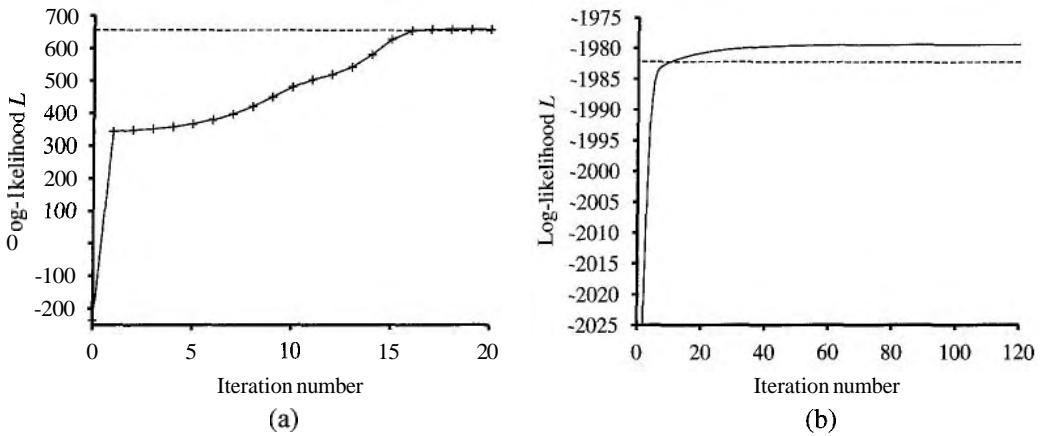


Figure 20.9 Graphs showing the log-likelihood of the data, L , as a function of the EM iteration. The horizontal line shows the log-likelihood according to the true model. (a) Graph for the Gaussian mixture model in Figure 20.8. (b) Graph for the Bayesian network in Figure 20.10(a).

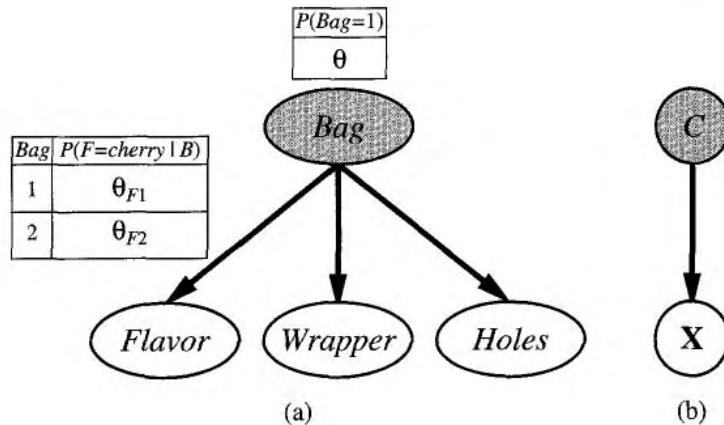


Figure 20.10 (a) A mixture model for candy. The proportions of different flavors, wrappers, and numbers of holes depend on the bag, which is not observed. (b) Bayesian network for a Gaussian mixture. The mean and covariance of the observable variables X depend on the component C .

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 rmodel is a mixture model. (In fact, we can also rmodel the mixture of Gaussians as a Bayesian network, as shown in Figure 20.10(b).) In the figure, the bag 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

$$\theta = 0.5, \theta_{F1} = \theta_{W1} = \theta_{H1} = 0.8, \theta_{F2} = \theta_{W2} = \theta_{H2} = 0.3. \quad (20.7)$$

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:

	$W = \text{red}$		$W = \text{green}$	
	$H = 1$	$H = 0$	$H = 1$	$H = 0$
$F = \text{cherry}$	273	93	104	90
$F = \text{lime}$	79	100	94	167

We start by initializing the parameters. For numerical simplicity, we will choose⁷

$$\theta^{(0)} = 0.6, \theta_{F1}^{(0)} = \theta_{W1}^{(0)} = \theta_{H1}^{(0)} = 0.6, \theta_{F2}^{(0)} = \theta_{W2}^{(0)} = \theta_{H2}^{(0)} = 0.4. \quad (20.8)$$

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

$$\frac{273}{1000} \cdot \frac{\theta_{F1}^{(0)}\theta_{W1}^{(0)}\theta_{H1}^{(0)}\theta^{(0)}}{\theta_{F1}^{(0)}\theta_{W1}^{(0)}\theta_{H1}^{(0)}\theta^{(0)} + \theta_{F2}^{(0)}\theta_{W2}^{(0)}\theta_{H2}^{(0)}(1 - \theta^{(0)})} \approx 0.22797.$$

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

$$\sum_{j: \text{Flavor}_j = \text{cherry}} P(\text{Bag} = 1 | \text{Flavor}_j = \text{cherry}, \text{wrapper}_j, \text{holes}_j).$$

Again, these probabilities can be calculated by any Bayes net algorithm. Completing this process, we obtain the new values of all the parameters:

$$\begin{aligned}\theta^{(1)} &= 0.6124, \theta_{F1}^{(1)} = 0.6684, \theta_{W1}^{(1)} = 0.6483, \theta_{H1}^{(1)} = 0.6558, \\ \theta_{F2}^{(1)} &= 0.3887, \theta_{W2}^{(1)} = 0.3817, \theta_{H2}^{(1)} = 0.3827.\end{aligned}\quad (20.9)$$

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:

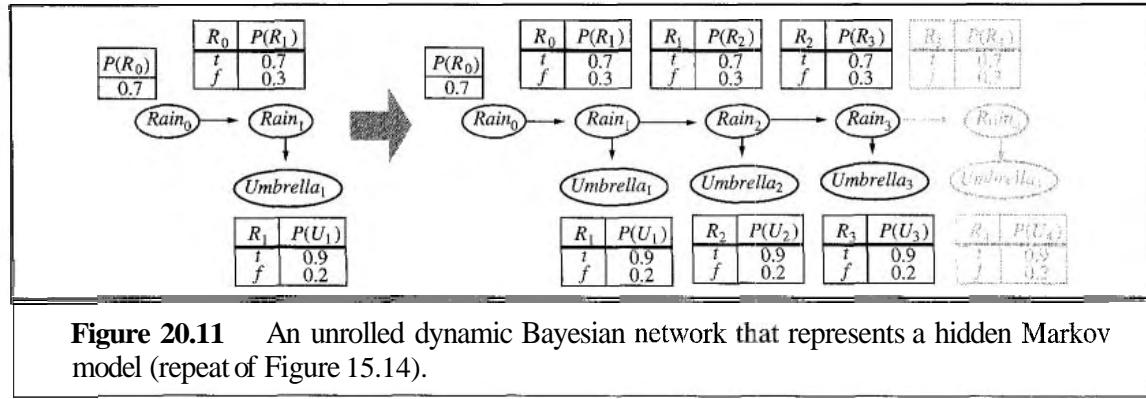
$$\theta_{ijk} \leftarrow \hat{N}(X_i = x_{ij}, Pa_i = pa_{ik}) / \hat{N}(Pa_i = pa_{ik}).$$

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



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

$$\boldsymbol{\theta}^{(i+1)} = \operatorname{argmax}_{\boldsymbol{\theta}} \sum_{\mathbf{z}} P(\mathbf{Z}=\mathbf{z}|\mathbf{x}, \boldsymbol{\theta}^{(i)}) L(\mathbf{x}, \mathbf{z}|\boldsymbol{\theta})$$

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

postiors 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

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 of 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!

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

The key idea of **nearest-neighbor** models is that the properties of any particular input point x are likely to be similar to those of points in the neighborhood of x . For example, if we want to do **density estimation**—that is, estimate the value of an unknown probability density at x —then we can simply measure the density with which points are scattered in the neighborhood of 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

PARAMETRIC
LEARNING

NONPARAMETRIC
LEARNING

INSTANCE-BASED
LEARNING

NEAREST-NEIGHBOR

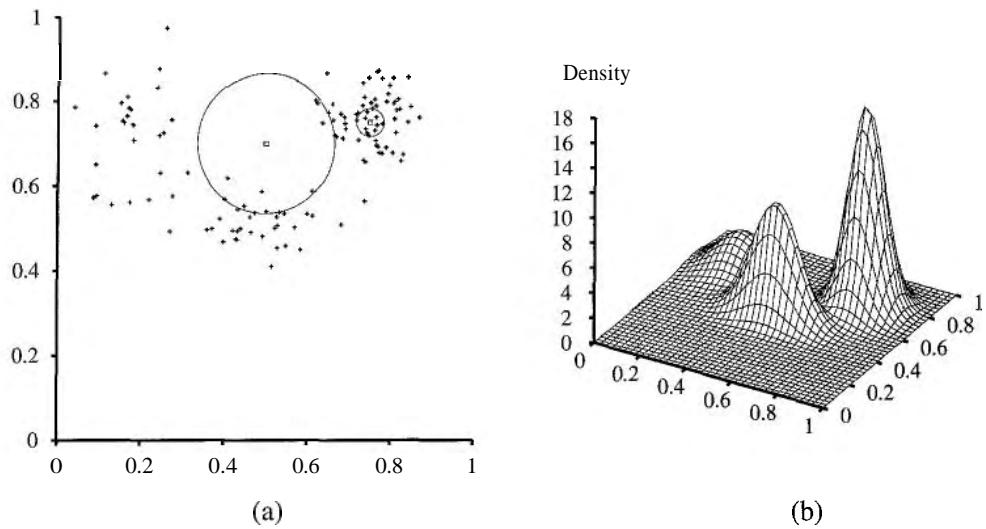


Figure 20.12 (a) A 128-point subsample of the data shown in Figure 20.8(a), together with two query points and their 10-nearest-neighborhoods. (b) A 3-D plot of the mixture of Gaussians from which the data were generated.

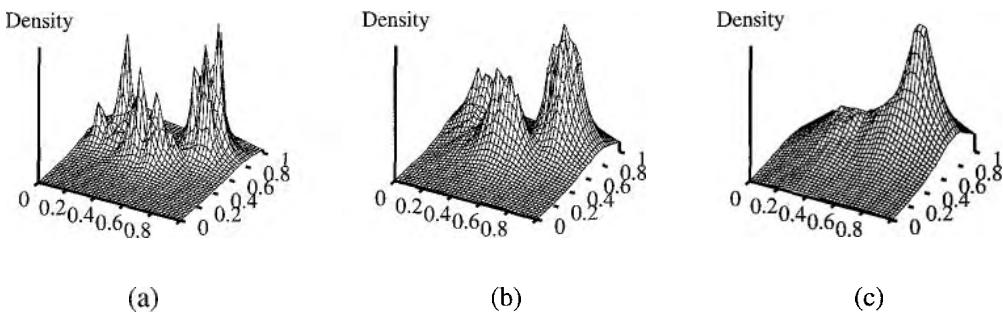


Figure 20.13 Density estimation using k-nearest-neighbors, applied to the data in Figure 20.12(a), for $k = 3, 10$, and 40 respectively.

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

MAHALANOBIS
DISTANCE
HAMMING DISTANCE

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.

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 ad \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

KERNELMODEL
KERNELFUNCTION

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

$$P(\mathbf{x}) = \frac{1}{N} \sum_{i=1}^N K(\mathbf{x}, \mathbf{x}_i) .$$

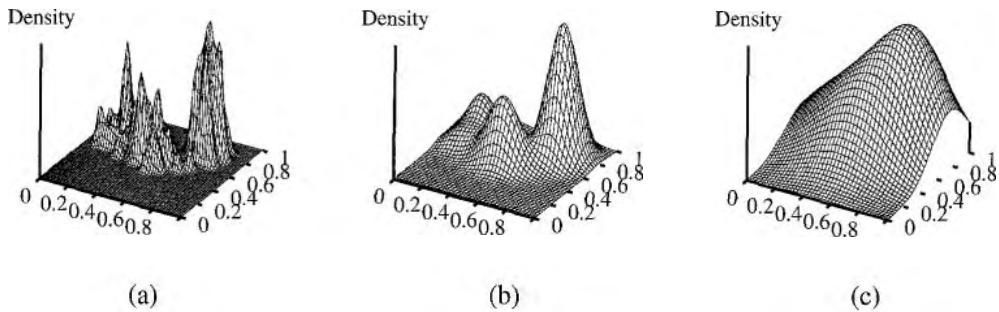


Figure 20.14 Kernel density estimation for the data in Figure 20.12(a), using Gaussian kernels with $w = 0.02, 0.07$, and 0.20 respectively.

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 i th 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-**

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.

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

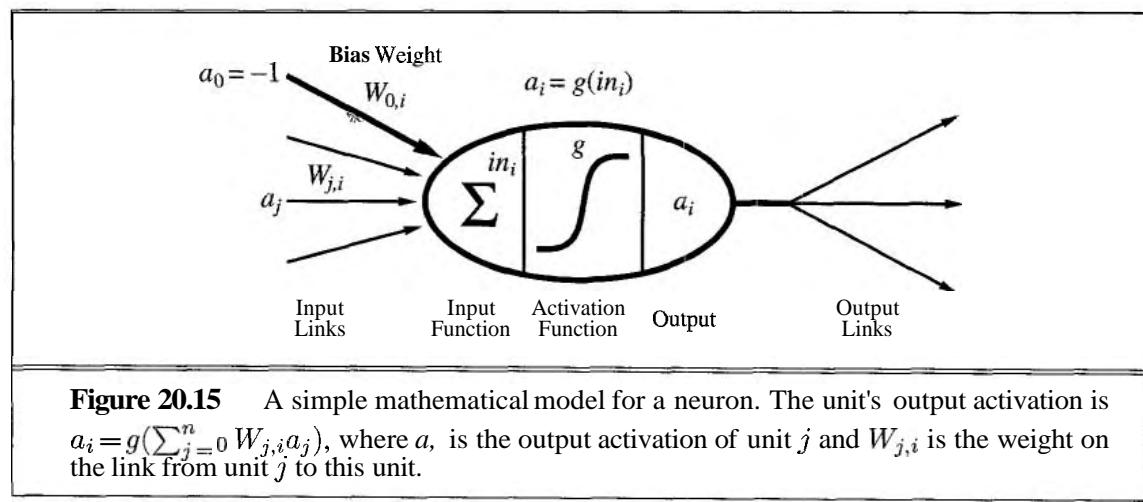
Then it applies an **activation function** g to this sum to derive the output:

$$a_i = g(in_i) = g\left(\sum_{j=0}^n W_{j,i} a_j\right) . \quad (20.10)$$

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



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 $W_{0,i}$.

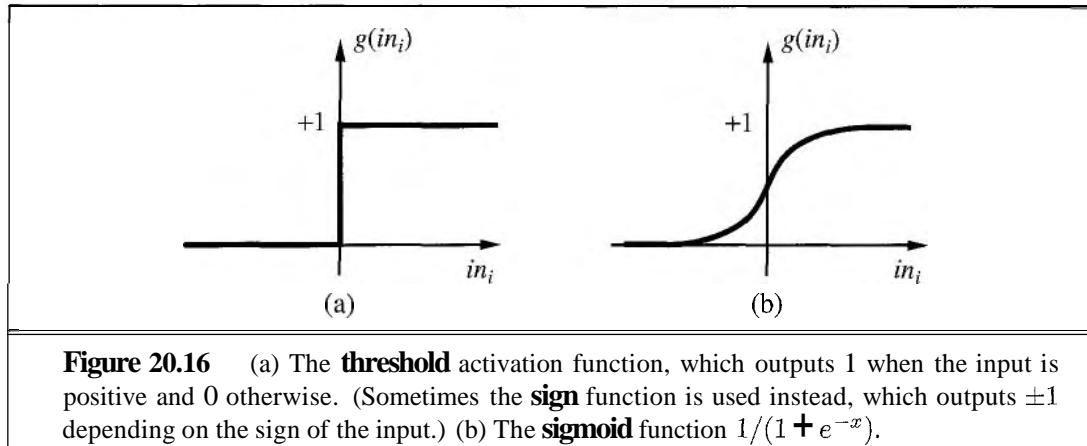


Figure 20.16 (a) The **threshold** activation function, which outputs 1 when the input is positive and 0 otherwise. (Sometimes the **sign** function is used instead, which outputs ± 1 depending on the sign of the input.) (b) The **sigmoid** function $1/(1 + e^{-x})$.

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.

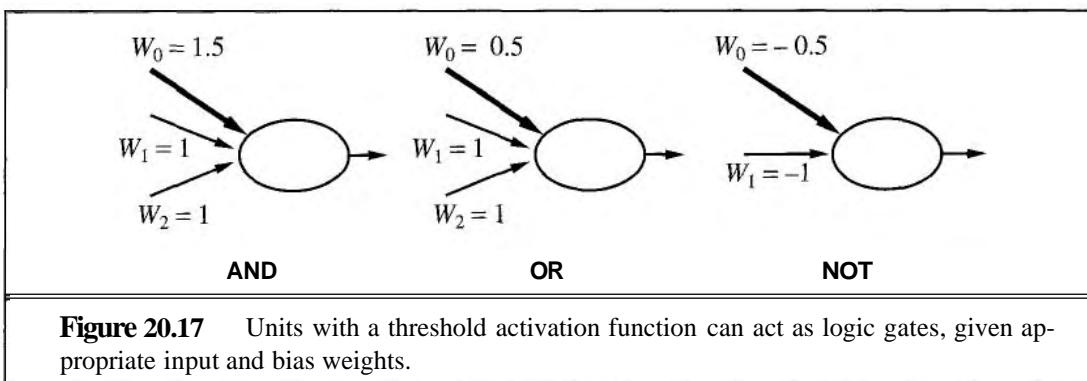


Figure 20.17 Units with a threshold activation function can act as logic gates, given appropriate input and bias weights.

Network structures

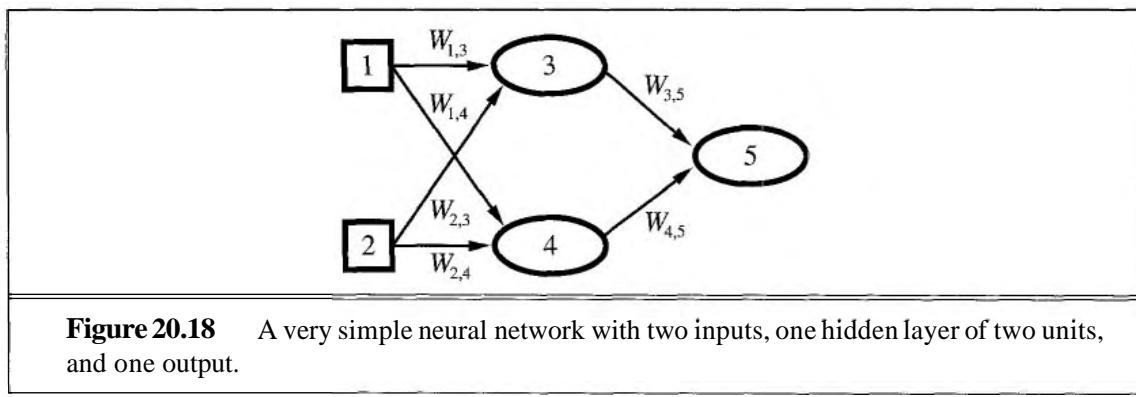
There are two main categories of neural network structures: acyclic or **feed-forward networks** and cyclic or **recurrent networks**. A feed-forward network represents a function of its current input; thus, it has no internal state other than the weights themselves. A recurrent

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 $x = (x_1, x_2)$, the activations of the input units are set to $(a_1, a_2) = (x_1, x_2)$ and the network computes

$$\begin{aligned} a_5 &= g(W_{3,5}a_3 + W_{4,5}a_4) \\ &= g(W_{3,5}g(W_{1,3}a_1 + W_{2,3}a_2) + W_{4,5}g(W_{1,4}a_1 + W_{2,4}a_2)) . \end{aligned} \quad (20.11)$$

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.



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.

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.

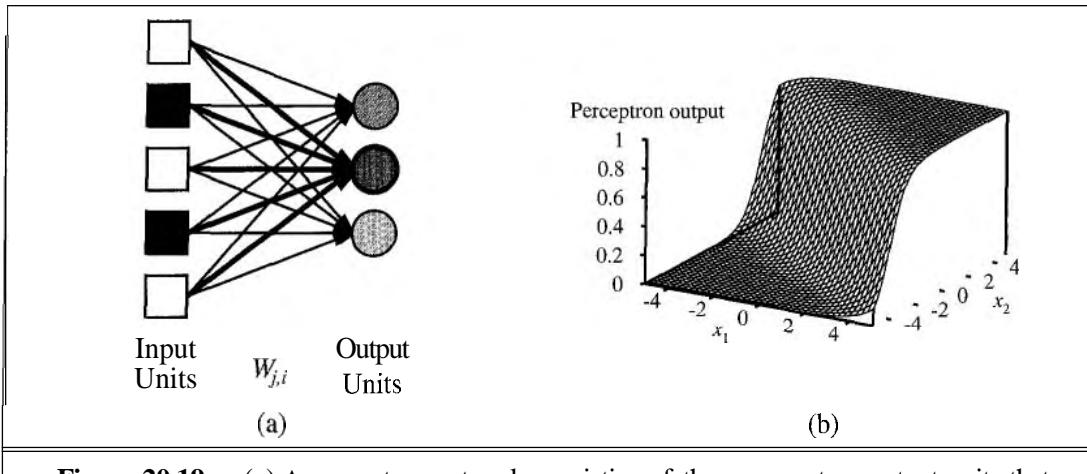


Figure 20.19 (a) A perceptron network consisting of three perceptron output units that share five inputs. Looking at a particular output unit (say the second one, outlined in bold), we see that the weights on its incoming links have no effect on the other output units. (b) A graph of the output of a 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).

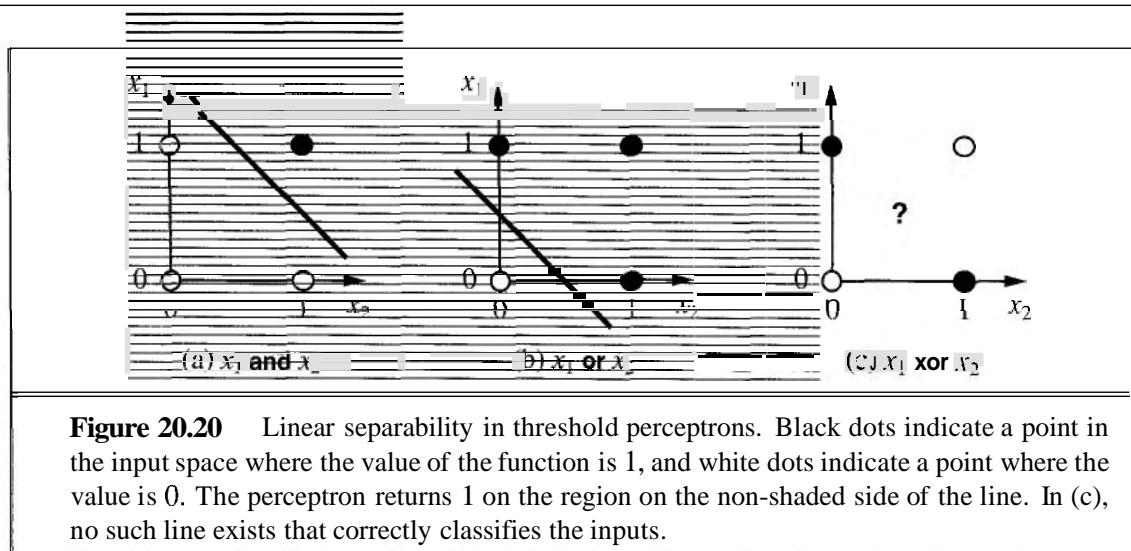
Let us begin by examining the hypothesis space that a perceptron can represent. With a threshold activation function, we can view the perceptron as representing a Boolean function. In addition to the elementary Boolean functions AND, OR, and NOT (Figure 20.17), a perceptron can represent some quite "complex" 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:

$$\sum_{j=0} W_j x_j > 0 \quad \text{or} \quad \mathbf{W} \cdot \mathbf{x} > 0$$

LINEAR SEPARATOR

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



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).)



WEIGHT SPACE

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.

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 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:

$$W_j \leftarrow W_j + \alpha \times Err \times g'(in) \times x_j , \quad (20.12)$$

where α is the **learning rate**. Intuitively, this makes a lot of sense. If the error $Err = y - h_w(\mathbf{x})$ 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  $x = x_1, \dots, x_n$  and output  $y$ 
           network, a perceptron with weights  $W_j$ ,  $j = 0 \dots n$ , and activation function  $g$ 
  repeat
    for each e in examples do
       $in \leftarrow \sum_{j=1}^n W_j x_j[e]$ 
       $Err \leftarrow y[e] - g(in)$ 
       $W_j \leftarrow W_j + \alpha \times Err \times g'(in) \times x_j[e]$ 
    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.

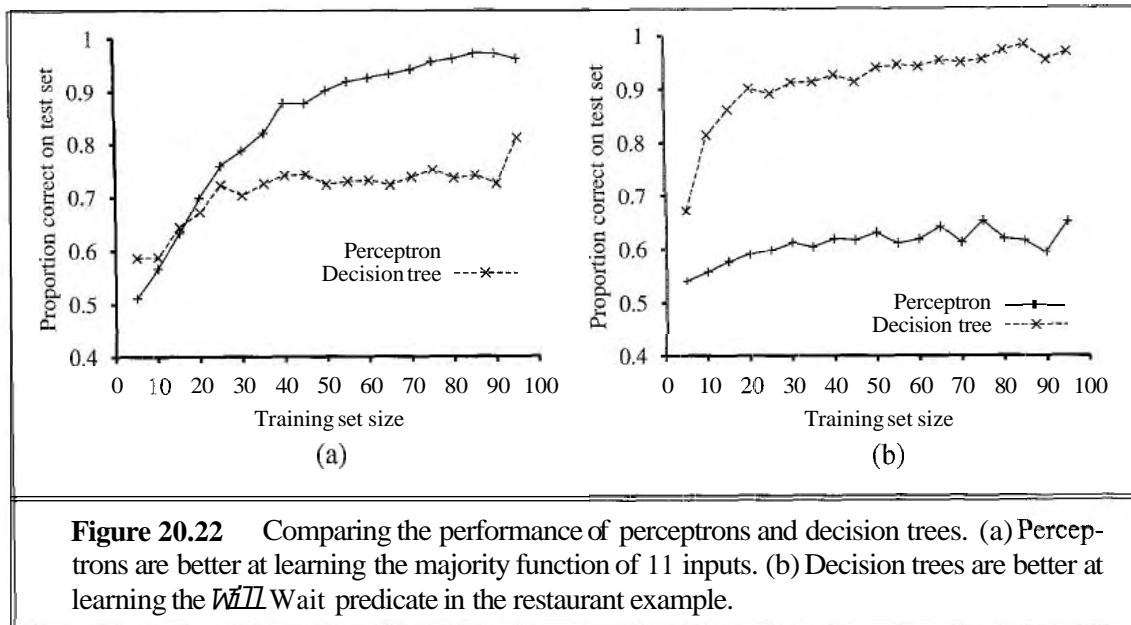


Figure 20.22 Comparing the performance of perceptrons and decision trees. (a) Perceptrons are better at learning the majority function of 11 inputs. (b) Decision trees are better at learning the *WILL Wait* predicate in the restaurant example.

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

$$L = \log p^T (1-p)^{(1-T)} = T \log p + (1-T) \log(1-p). \quad (20.13)$$

Thanks to the properties of the sigmoid function, the gradient reduces to a very simple formula (Exercise 20.16):

$$\frac{\partial L}{\partial W_i} = Err \times x_j.$$

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.

INDICATOR VARIABLE



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 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).

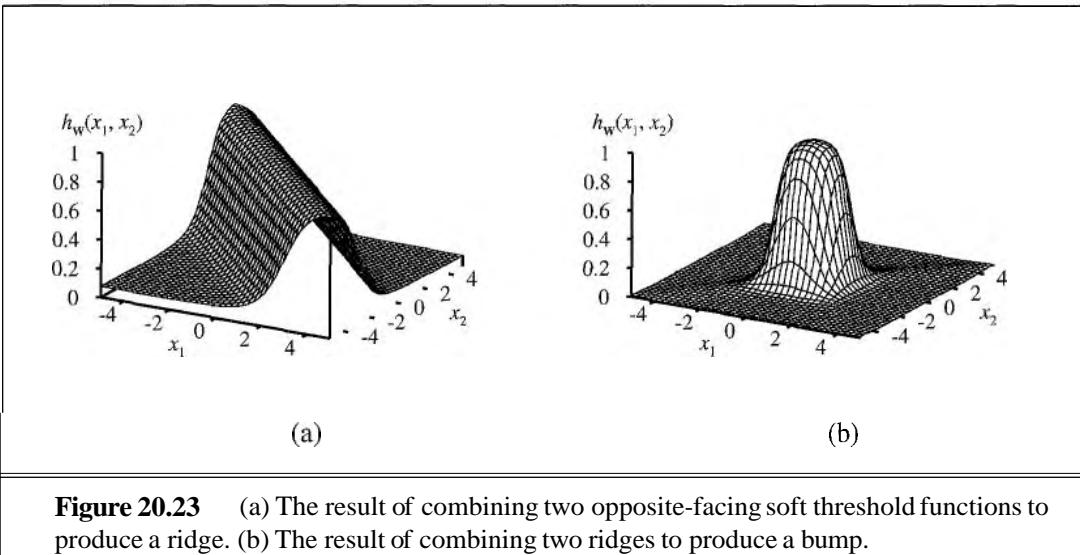


Figure 20.23 (a) The result of combining two opposite-facing soft threshold functions to produce a ridge. (b) The result of combining two ridges to produce a bump.

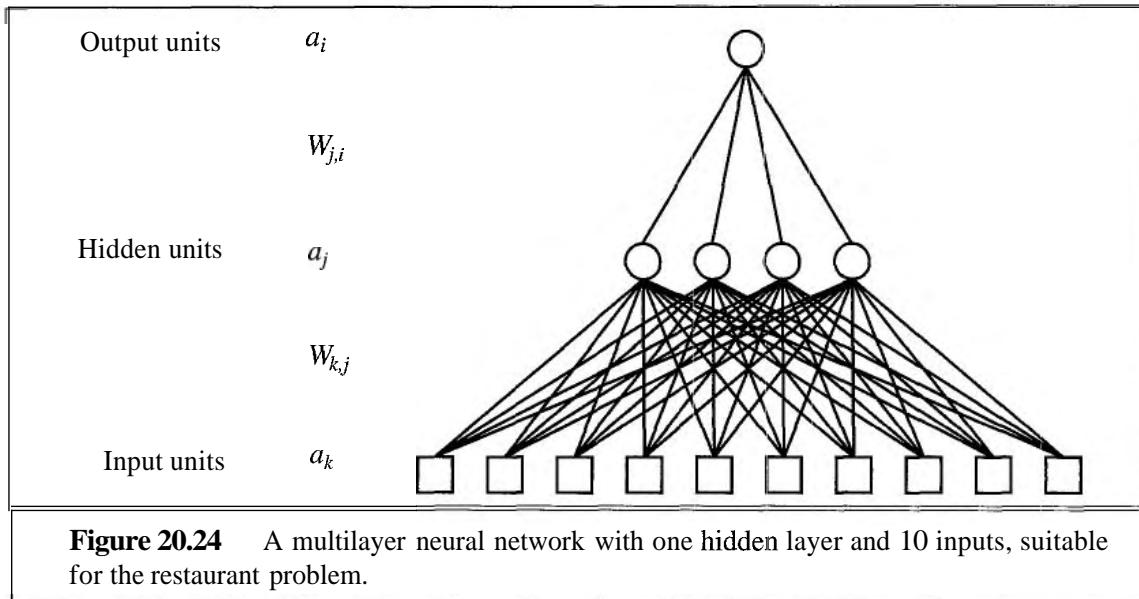
With more hidden units, we can produce more bumps of different sizes in more places. In fact, with a single, sufficiently large hidden layer, it is possible to represent any continuous function of the inputs with arbitrary accuracy; with two layers, even discontinuous functions can be represented.¹² 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 shown 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.



BACK-PROPAGATION

we have an output vector $\mathbf{h}_w(x)$ 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.

At the output layer, the weight-update rule is identical to Equation (20.12). We have multiple output units, so let Err_i be i th 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

$$W_{j,i} \leftarrow W_{j,i} + \alpha \times a_j \times \Delta_i . \quad (20.14)$$

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 A values is the following:

$$\Delta_j = g'(in_j) \sum_i W_{j,i} \Delta_i . \quad (20.15)$$

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  $a_j \leftarrow x_j[e]$ 
      for  $\ell = 2$  to L do
         $in_i \leftarrow \sum_j W_{j,i} a_j$ 
         $a_i \leftarrow g(in_i)$ 
      for each node i in the output layer do
         $\Delta_i \leftarrow g'(in_i) \times (y_i[e] - a_i)$ 
      for  $\ell = L - 1$  to 1 do
        for each node j in layer  $\ell$  do
           $\Delta_j \leftarrow g'(in_j) \sum_i W_{j,i} \Delta_i$ 
        for each node i in layer  $\ell + 1$  do
           $W_{j,i} \leftarrow W_{j,i} + \alpha \times a_j \times \Delta_i$ 
    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

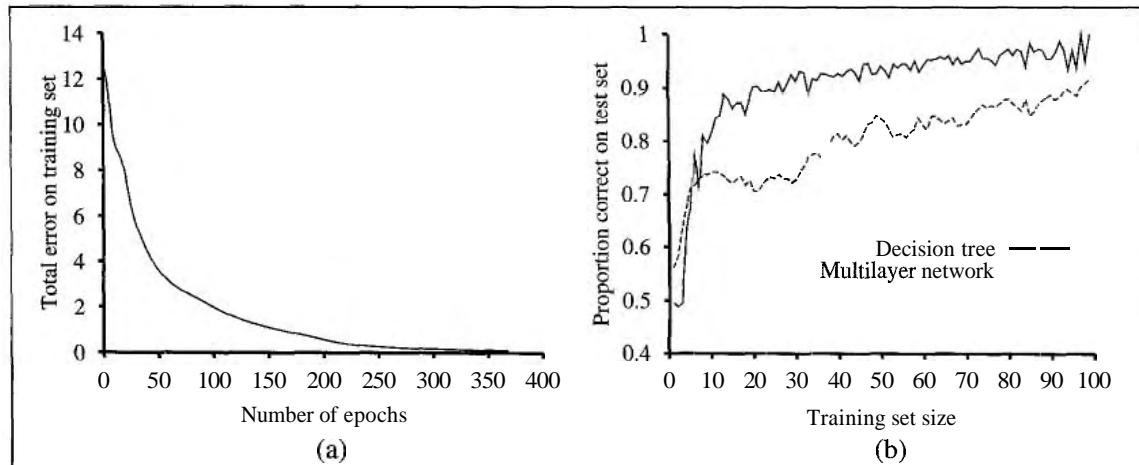


Figure 20.26 (a) Training curve showing the gradual reduction in error as weights are modified over several epochs, for a given set of examples in the restaurant domain. (b) Comparative learning curves showing that decision-tree learning does slightly better than back-propagation in a multilayer network.

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. 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. 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.

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

$$f_1 = x_1^2, \quad f_2 = x_2^2, \quad f_3 = \sqrt{2}x_1x_2. \quad (20.16)$$

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.

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

¹⁴ The reader may notice that we could have used just f_1 and f_2 , but the 3D mapping illustrates the idea better.

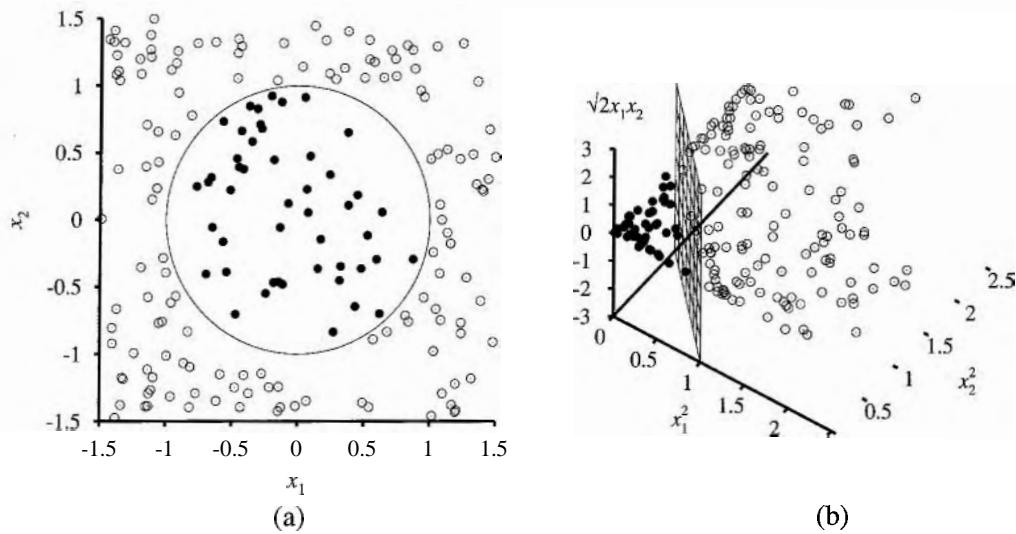


Figure 20.27 (a) A two-dimensional training set with positive examples as black circles and negative examples as white circles. The true decision boundary, $x_1^2 + x_2^2 \leq 1$, is also shown. (b) The same data after mapping into a three-dimensional input space $(x_1^2, x_2^2, \sqrt{2}x_1x_2)$. The circular decision boundary in (a) becomes a linear decision boundary in three dimensions.

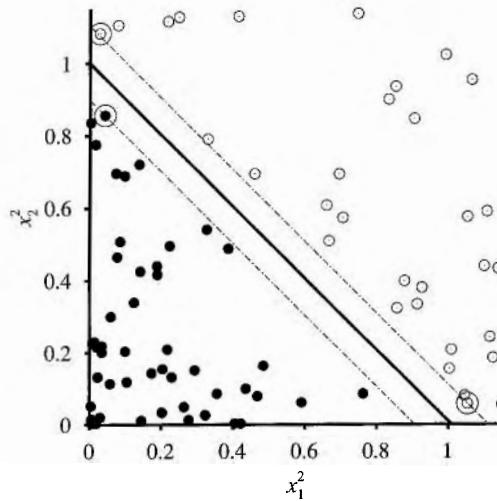


Figure 20.28 A close-up, projected onto the first two dimensions, of the optimal separator shown in Figure 20.27(b). The separator is shown as a heavy line, with the closest points—the support vectors—marked with circles. The margin is the separation between the positive and negative examples.

is to find values of the parameters α_i that maximize the expression

$$\sum_i \alpha_i - \frac{1}{2} \sum_{i,j} \alpha_i \alpha_j y_i y_j (\mathbf{x}_i \cdot \mathbf{x}_j) \quad (20.17)$$

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_i s have been calculated, it is

$$h(\mathbf{x}) = \text{sign} \left(\sum_i \alpha_i y_i (\mathbf{x} \cdot \mathbf{x}_i) \right) \quad (20.18)$$

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 .

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,

SUPPORT VECTOR

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.

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.

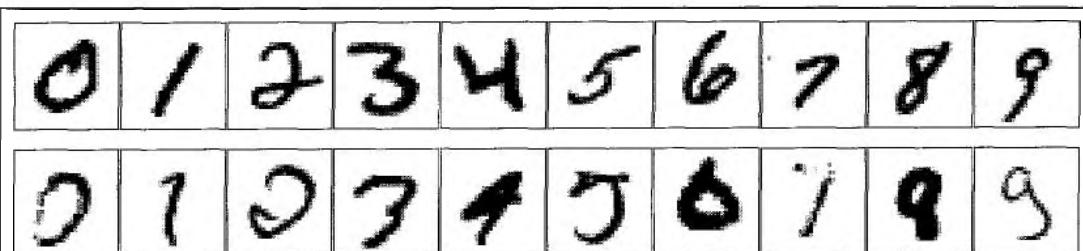


Figure 20.29 Examples from the NIST database of handwritten digits. Top row: examples of digits 0–9 that are easy to identify. Bottom row: more difficult examples of the same digits.

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:

- a **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.
- a **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.
- a 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 Chapter 13. Its surprising success 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.

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 Scholkopf (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.

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.

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

RADIAL BASIS
FUNCTION

HOPFIELD
NETWORKS

ASSOCIATIVE
MEMORY

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 Scholkopf and Smola (2002) provide rigorous introductions; a friendlier exposition appears in the *AI Magazine* article by Cristianini and Scholkopf (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

20.6 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 $P(h_i|d_1, \dots, d_m)$ and $P(D_{m+1} = \text{lime}|d_1, \dots, d_m)$. Comment on your results.

20.2 Repeat Exercise 20.1, this time plotting the values of $P(D_{m+1} = \text{lime}|h_{\text{MAP}})$ and $P(D_{m+1} = \text{lime}|h_{\text{ML}})$.

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 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_j s are generated from the x_j s 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 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^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	0	0	0	0	0	0
I_2	0	0	0	1	1	0	0	1	1	0	1	0
I_3	1	1	1	0	1	0	0	1	1	0	0	1
I_4	0	1	0	0	1	0	0	1	0	1	1	0
I_5	0	0	1	1	0	1	1	0	1	1	0	0
I_6	0	0	0	1	0	1	0	1	1	0	1	1
T	1	1	1	1	1	1	0	1	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 $\partial L / \partial W_j = Err \times x_j$.

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 W , write down equations for the value of the units in the output layer as a function of W and the input layer 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 **A utility-based agent** learns a utility function on states and uses it to select actions that maximize the expected outcome utility.
- a **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 **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.

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.

0-LEARNING
ACTION-VALUE

PASSIVE LEARNING

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.

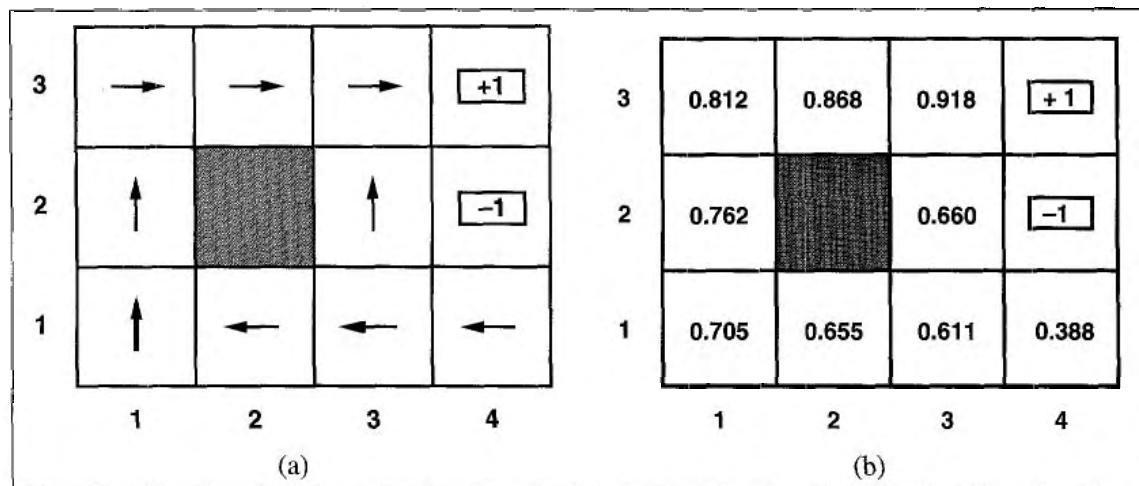


Figure 21.1 (a) A policy π for the 4×3 world; this policy happens to be optimal with rewards of $R(s) = -0.04$ in the nonterminal states and no discounting. (b) The utilities of the states in the 4×3 world, given policy π .

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:

$$\begin{aligned}
 &(1,1)\text{-}.04 \rightarrow (1,2)\text{-}.04 \rightarrow (1,3)\text{-}.04 \rightarrow (1,2)\text{-}.04 \rightarrow (1,3)\text{-}.04 \rightarrow (2,3)\text{-}.04 \rightarrow (3,3)\text{-}.04 \rightarrow (4,3)+1 \\
 &(1,1)\text{-}.04 \rightarrow (1,2)\text{-}.04 \rightarrow (1,3)\text{-}.04 \rightarrow (2,3)\text{-}.04 \rightarrow (3,3)\text{-}.04 \rightarrow (3,2)\text{-}.04 \rightarrow (3,3)\text{-}.04 \rightarrow (4,3)+1 \\
 &(1,1)\text{-}.04 \rightarrow (2,1)\text{-}.04 \rightarrow (3,1)\text{-}.04 \rightarrow (3,2)\text{-}.04 \rightarrow (4,2)-1 .
 \end{aligned}$$

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

$$U^\pi(s) = E \left[\sum_{t=0}^{\infty} \gamma^t R(s_t) \mid \pi, s_0 = s \right] \quad (21.1)$$

We will include a **discount factor** γ in all of our equations, but for the 4 x 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)):

$$U^\pi(s) = R(s) + \gamma \sum_{s'} T(s, \pi(s), s') U^\pi(s'). \quad (21.2)$$

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

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 $T((1, 3), \text{Right}, (2, 3))$ 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 x 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 $U^n(1, 3) = 0.84$ and $U^n(2, 3) = 0.92$. Now, if this transition occurred all the time, we would expect the utilities to obey

$$U^n(1, 3) = -0.04 + U^n(2, 3),$$

so $U^n(1, 3)$ 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:  $\pi$ , a fixed policy
    mdp, an MDP with model  $T$ , rewards  $R$ , discount  $\gamma$ 
     $U$ , 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  $U[s] \leftarrow r; R[s] \leftarrow r'$ 
  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
       $T[s, a, t] \leftarrow N_{sas'}[s, a, t] / N_{sa}[s, a]$ 
     $U \leftarrow \text{POLICY-EVALUATION}(\wedge, U, \text{mdp})$ 
  if TERMINAL? $[s']$  then  $s, a \leftarrow \text{null}$  else  $s, a \leftarrow s, \pi[s']$ 
  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.

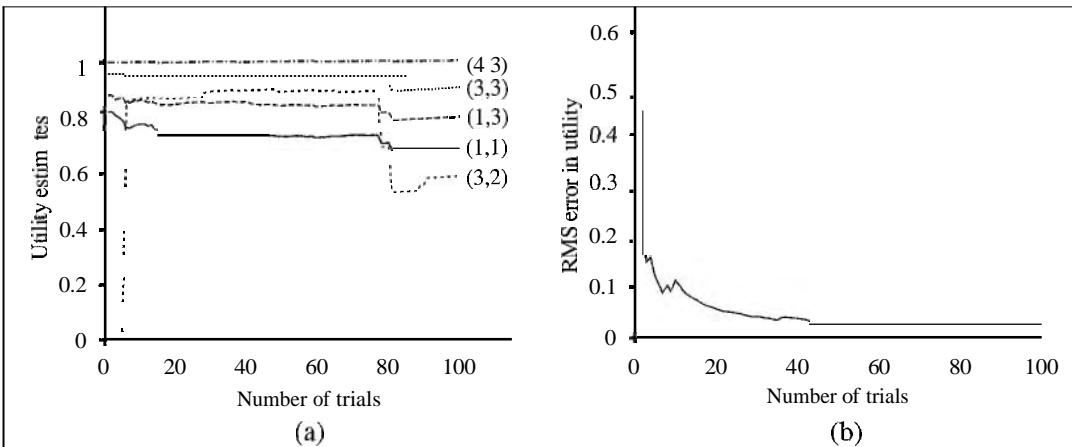


Figure 21.3 The passive ADP learning curves for the 4×3 world, given the optimal policy shown in Figure 21.1. (a) The utility estimates for a selected subset of states, as a function of the number of trials. Notice the large changes occurring around the 78th trial—this is the first time that the agent falls into the -1 terminal state at $(4,2)$. (b) The root-mean-squareerror 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:  $n$ , 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  $U[s'] \leftarrow r'$ 
  if  $s$  is not null then do
    increment  $N_s[s]$ 
     $U[s] \leftarrow U[s] + \alpha(N_s[s])(r + \gamma U[s'] - U[s])$ 
  if TERMINAL? $[s']$  then  $s, a, r \leftarrow$  null else  $s, a, r \leftarrow s, \pi[s'], r'$ 
  return  $a$ 

```

Figure 21.4 A passive reinforcement learning agent that learns utility estimates using temporal differences.

following update to $U^\pi(s)$

$$U^\pi(s) \leftarrow U^\pi(s) + \alpha(R(s) + \gamma U^\pi(s') - U^\pi(s)). \quad (21.3)$$

TEMPORAL-DIFFERENCE

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.

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 $\alpha(n) = 60/(59+n)$.



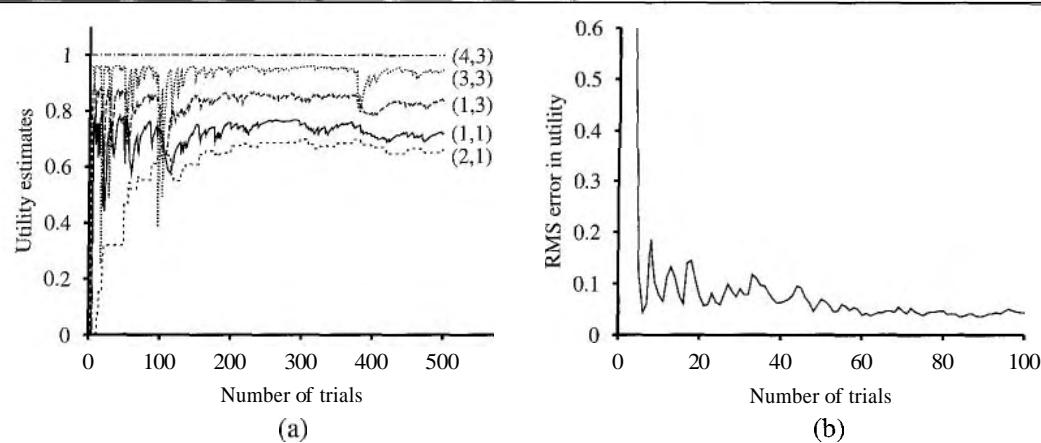


Figure 21.5 The TD learning curves for the 4×3 world. (a) The utility estimates for a selected subset of states, as a function of the number of trials. (b) The root-mean-squareerror in the estimate for $U(1, 1)$, averaged over 20 runs of 500 trials each. Only the first 100 trials are shown to enable 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. 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:

$$U(s) = R(s) + \gamma \max_{s'} \sum T(s, a, s') U(s') . \quad (21.4)$$

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.

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

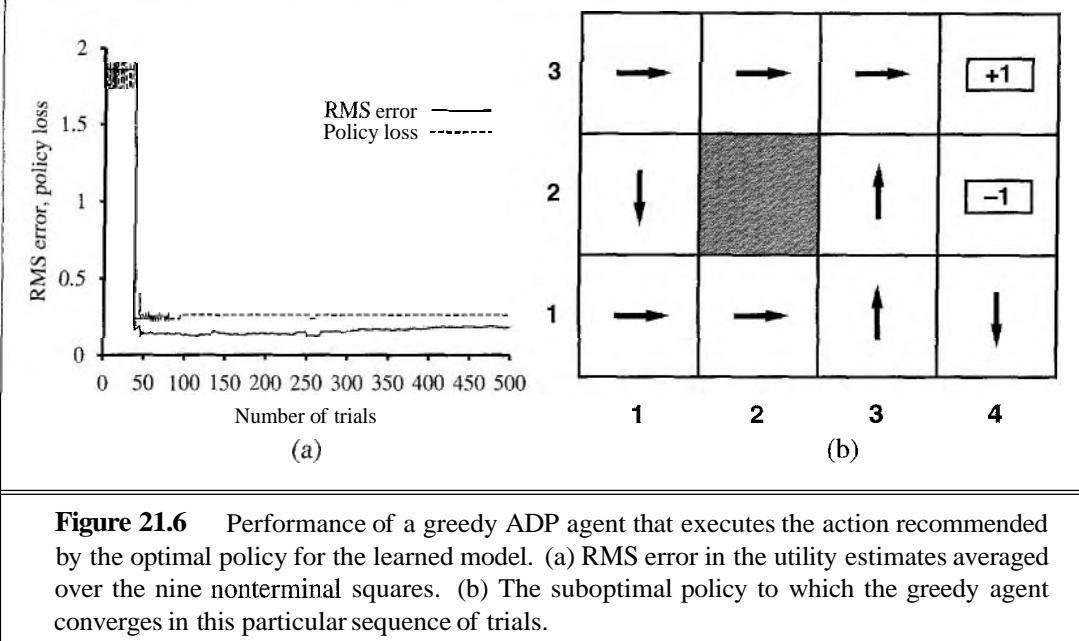


Figure 21.6 Performance of a greedy ADP agent that executes the action recommended by the optimal policy for the learned model. (a) RMS error in the utility estimates averaged over the nine nonterminal squares. (b) The suboptimal policy to which the greedy agent converges in this particular sequence 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.

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.)

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

EXPLOITATION
EXPLORATION

BANDIT PROBLEMS

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:

$$U^+(s) \leftarrow R(s) + \gamma \max_a f \left(\sum_{s'} T(s, a, s') U^+(s'), N(a, s) \right). \quad (21.5)$$

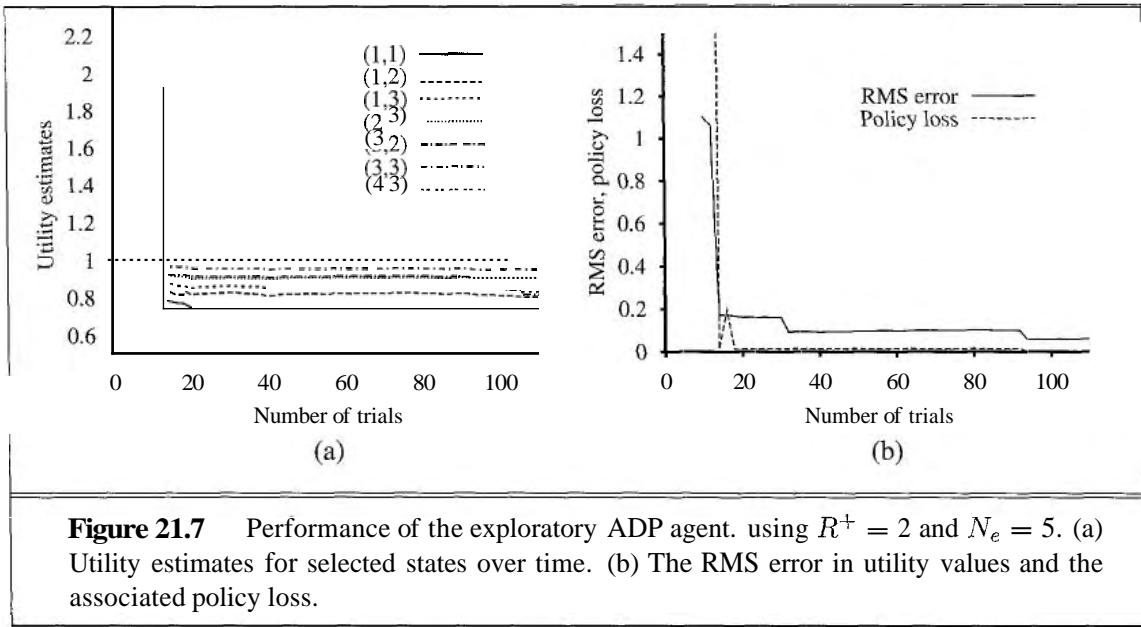
EXPLORATION
FUNCTION

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.



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:

$$U(s) = \max_a Q(a, s). \quad (21.6)$$

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  $N_{sa}[s, a]$ 
     $Q[a, s] \leftarrow Q[a, s] + \alpha(N_{sa}[s, a])(r + \gamma \max_{a'} Q[a', s'] - Q[a, s])$ 
  if TERMINAL? $[s']$  then  $s, a, r \leftarrow$  null
  else  $s, a, r \leftarrow s', \text{argmax}_{a'} f(Q[a', s'], N_{sa}[s', a']), r'$ 
  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:

$$Q(a, s) = R(s) + \gamma \sum_{s'} T(s, a, s') \max_{a'} Q(a', s') . \quad (21.7)$$

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

$$Q(a, s) \leftarrow Q(a, s) + \alpha(R(s) + \gamma \max_{a'} Q(a', s') - Q(a, s)) , \quad (21.8)$$

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

BASIS FUNCTIONS

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, \dots, f_n :

$$\hat{U}_\theta(s) = \theta_1 f_1(s) + \theta_2 f_2(s) + \dots + \theta_n f_n(s).$$

A reinforcement learning algorithm can learn values for the parameters $\theta = \theta_1, \dots, \theta_n$ such that the evaluation function \hat{U}_θ 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

$$\hat{U}_\theta(x, y) = \theta_0 + \theta_1 x + \theta_2 y . \quad (21.9)$$

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 j th 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

$$\theta_i \leftarrow \theta_i - \alpha \frac{\partial E_j(s)}{\partial \theta_i} = \theta_i + \alpha (u_j(s) - \hat{U}_\theta(s)) \frac{\partial \hat{U}_\theta(s)}{\partial \theta_i} . \quad (21.10)$$

WIDROW-HOFF RULE This is called the **Widrow-Hoff rule**, or the **delta rule**, for online least-squares. For the DELTA RULE linear function approximator $\hat{U}_\theta(s)$ in Equation (21.9), we get three simple update rules:

$$\begin{aligned} \theta_0 &\leftarrow \theta_0 + \alpha (u_j(s) - \hat{U}_\theta(s)) , \\ \theta_1 &\leftarrow \theta_1 + \alpha (u_j(s) - \hat{U}_\theta(s)) x , \\ \theta_2 &\leftarrow \theta_2 + \alpha (u_j(s) - \hat{U}_\theta(s)) y . \end{aligned}$$

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_i s also changes the values of \hat{U}_θ 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

$$\theta_i \leftarrow \theta_i + \alpha [R(s) + \gamma \hat{U}_\theta(s') - \hat{U}_\theta(s)] \frac{\partial \hat{U}_\theta(s)}{\partial \theta_i} \quad (21.11)$$

for utilities and

$$\theta_i \leftarrow \theta_i + \alpha [R(s) + \gamma \max_{a'} \hat{Q}_\theta(a', s') - \hat{Q}_\theta(a, s)] \frac{\partial \hat{Q}_\theta(a, s)}{\partial \theta_i} \quad (21.12)$$

for Q-values. These update rules can be shown to converge to the closest possible⁵ 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

BANG-BANG
CONTROL

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 0 are continuous. The actions are usually discrete: jerk left or jerk right, the so-called **bang-bang control** regime.

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

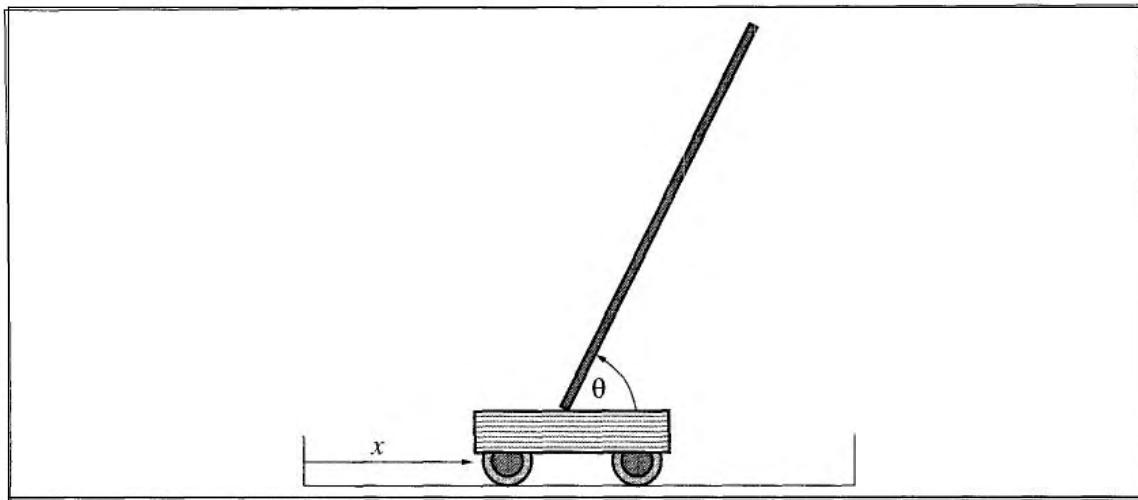


Figure 21.9 Setup for the problem of balancing a long pole on top of a moving cart. The cart can be jerked left or right by a controller that observes x , θ , \dot{x} , and $\dot{\theta}$.

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:

$$\pi(s) = \text{rmax } \hat{Q}_\theta(a, s) . \quad (21.13)$$

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}_θ 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}_θ . The value of θ that gives good play may be a long way from making \hat{U}_θ 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**:

$$\pi_\theta(s, a) = \exp(\hat{Q}_\theta(a, s)) / \sum_{a'} \exp(\hat{Q}_\theta(a', s)).$$

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.

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 j th 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 j th trial and $R_j(s)$ is the total reward received from state s onwards in the j th 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 blackjack⁸ 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 *have each program play the same 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.

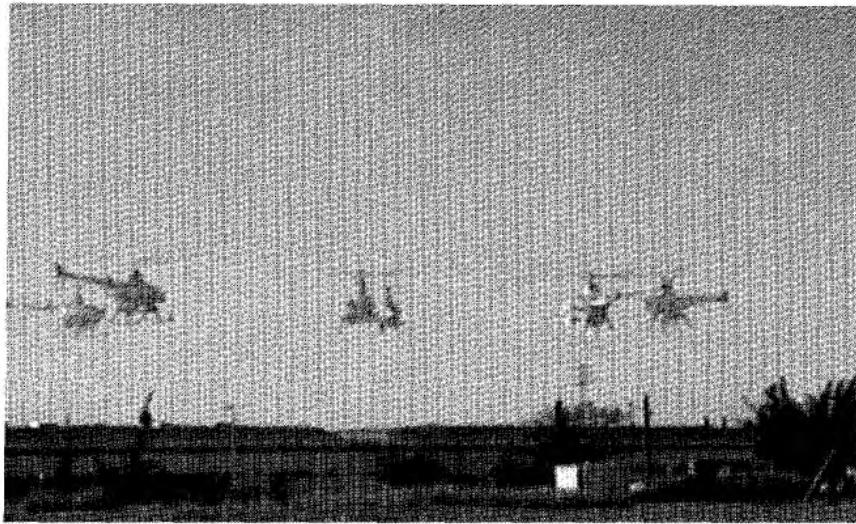


Figure 21.10 Superimposed time-lapse images of an autonomous helicopter performing a very difficult "nose-in circle" maneuver. The helicopter is under the control of a policy developed by the PEGASUS policy search algorithm. A simulator model was developed by observing the effects of various control 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 cat–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 $TD(\lambda)$ algorithm. $TD(\lambda)$ 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 $TD(\lambda)$ and related algorithms make inefficient use of experiences; essentially, they are online regression algorithms that converge much more slowly than offline regression. His $LSTD(\lambda)$ 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 Tennenholz (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).

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.

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.

REWARD SHAPING

HIERARCHICAL
REINFORCEMENT
LEARNING

PARTIALPROGRAM

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**.
- **Acknowledge** requests: **OK.**
- **Promise** or commit to a plan: *I'll shoot the wumpus; you grab the gold.*

INDIRECT SPEECH ACT

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

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 A and P, and a typical string is "P A Q." The string "P Q A" 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

STRINGS

TERMINALSYMBOLS

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.

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*.

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.

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. 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. 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 b s and then c s).

In **context-free** grammars (or CFGs), 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 non-terminal on the left-hand side and a terminal symbol optionally followed by a non-terminal 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 a s followed by any number of b s.

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).

SYNTHESIS

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. 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.

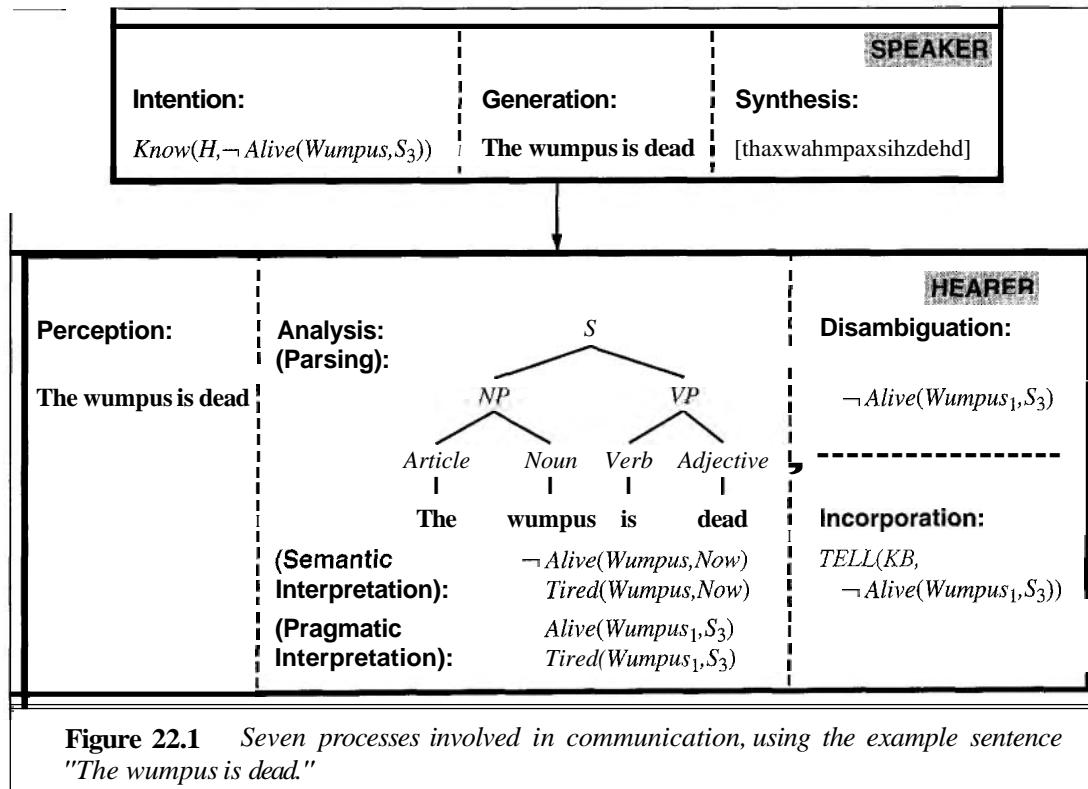
PERCEPTION

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, \dots, 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
PARSETREE
SEMANTIC
INTERPRETATION

PRAGMATIC
INTERPRETATION

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. \mathbf{H} infers that S intended to convey P_i (where ideally $P_i = \mathbf{P}$). Most speakers are not intentionally ambiguous, but most utterances have several feasible interpretations. Communication works because the hexer 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. \mathbf{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 \mathbf{P} , 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 12.
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  $\leftarrow$  UPDATE-STATE(state, action, percept)
    words  $\leftarrow$  SPEECH-PART(percept)
    semantics  $\leftarrow$  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  $\leftarrow$  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),

<i>Noun</i>	\rightarrow	stench breeze glitter nothing agent wumpus pit pits gold east ...
<i>Verb</i>	\rightarrow	is see smell shoot feel stinks go grab carry kill turn ...
<i>Adjective</i>	\rightarrow	right left east dead back smelly ...
<i>Adverb</i>	\rightarrow	here there nearby ahead right left east south back ...
<i>Pronoun</i>	\rightarrow	me you I it ...
<i>Name</i>	\rightarrow	John Mary Boston Aristotle ...
<i>Article</i>	\rightarrow	the a an ...
<i>Preposition</i>	\rightarrow	to in on near ...
<i>Conjunction</i>	\rightarrow	and or but ...
<i>Digit</i>	\rightarrow	0 1 2 3 4 5 6 7 8 9

Figure 22.3 The lexicon for \mathcal{E}_0 .

<i>S</i>	\rightarrow	<i>NP VP</i>	<i>I</i> + feel a breeze
		<i>S Conjunction S</i>	I feel a breeze + and + I smell a wampus

<i>NP</i>	\rightarrow	<i>Pronoun</i>	<i>I</i>
		<i>Name</i>	John
		<i>Noun</i>	pits
		<i>Article Noun</i>	the + wampus
		<i>Digit Digit</i>	3 4
		<i>NP PP</i>	the wampus + to the east
		<i>NP RelClause</i>	the wampus + that is smelly

<i>VP</i>	\rightarrow	<i>Verb</i>	stinks
		<i>VP NP</i>	feel + a breeze:
		<i>VP Adjective</i>	is + smelly
		<i>VP PP</i>	turn + to the east
		<i>VP Adverb</i>	go + ahead

<i>PP</i>	\rightarrow	<i>Preposition NP</i>	to + the east
<i>RelClause</i>	\rightarrow	that <i>VP</i>	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

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 **under-generates**: 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.)

UNDERGENERATION

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]]].



TOP-DOWN PARSING

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:

- 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 $X \rightarrow X\dots$. With a depth-first search, such a rule would lead us to keep replacing X with $[X: X\dots]$ 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	<i>Article</i> \rightarrow the
2	Article wumpus is dead	wumpus	<i>Noun</i> \rightarrow wumpus
3	Article Noun is dead	<i>Article Noun</i>	<i>NP</i> \leftarrow <i>Article Noun</i>
4	NP is dead	is	<i>Verb</i> \rightarrow is
5	NP Verb dead	dead	<i>Adjective</i> \rightarrow dead
6	NP Verb Adjective	Verb	<i>VP</i> \rightarrow Verb
7	NP VP Adjective	<i>VP Adjective</i>	<i>VP</i> \rightarrow <i>VP Adjective</i>
8	NP VP	<i>NP VP</i>	<i>S</i> \rightarrow <i>NP VP</i>
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.

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

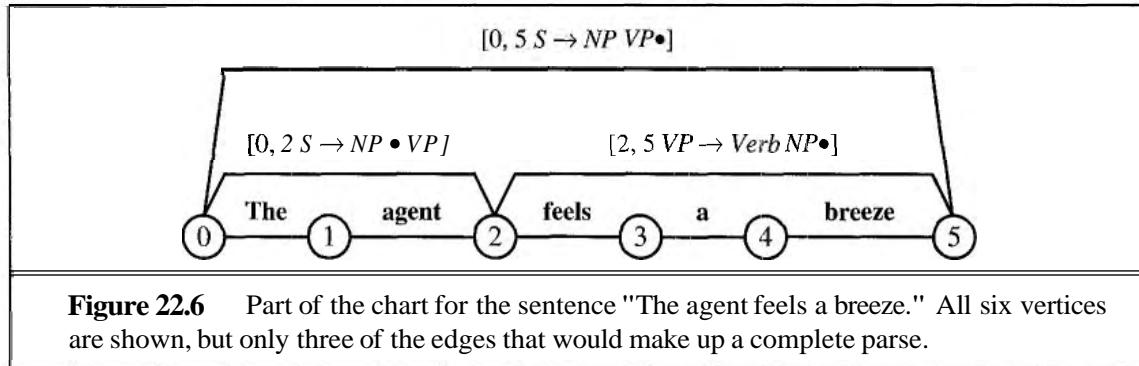
$$[0, 5, S \rightarrow NP VP \bullet]$$

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**.



```

function CHART-PARSE(words, grammar) returns chart
  chart  $\leftarrow$  array[0... LENGTH(words)] of empty lists
  ADD-EDGE([0, 0,  $S' \rightarrow \cdot S$ ])
  for i  $\leftarrow$  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  $\rightarrow a \cdot B \beta$ ] in chart[j] do
    if word is of category B then
      ADD-EDGE([i, j+1, A  $\rightarrow \alpha B \beta$ ])
  procedure PREDICTOR([i, j, A  $\rightarrow a \cdot B \beta$ ])
  /* Add to chart any rules for B that could help extend this edge */
  for each (B  $\rightarrow y$ ) in REWRITES-FOR(B, grammar) do
    ADD-EDGE([j, j, B  $\rightarrow \cdot \gamma$ ])
  procedure EXTENDER([j, k, B  $\rightarrow \gamma \bullet$ ])
  /* See what edges can be extended by this edge */
  eB  $\leftarrow$  the edge that is the input to this procedure
  for each [i, j, A  $\rightarrow a \cdot B' \beta$ ] in chart[j] do
    if B = B' then
      ADD-EDGE@(k, A  $\rightarrow a e_B \beta$ )

```

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 $[0, 0, S^t \rightarrow ■ S]$ to the chart, where S is the grammar's start symbol, and S^t 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, $[S \rightarrow NP VP J]$. 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^t 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.

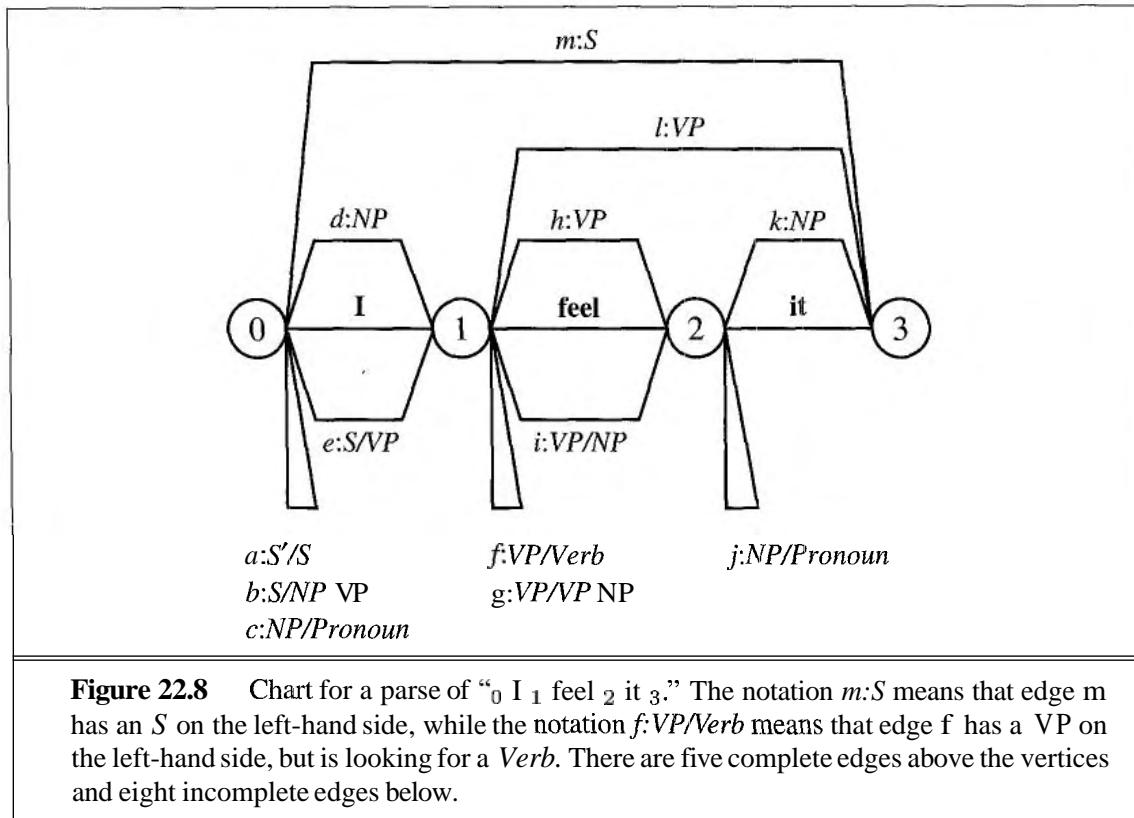
The other bottom-up component, EXTENDER,⁴ takes a complete edge with left hand side B and uses it to extend any incomplete rule in the chart that ends where the complete edge starts if the incomplete rule is looking for a B .

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.)

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.



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)	$[0, 1, NP \rightarrow Pronoun^*]$
e	EXTENDER(b,d)	$[0, 1, S \rightarrow NP \bullet VP]$
f	PREDICTOR(e)	$[1, 1, VP \rightarrow \bullet Verb]$
g	PREDICTOR(e)	$[1, 1, VP \rightarrow \blacksquare VP\ NP]$
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 " $_0 I _1 \text{feel} _2 \text{it} _3$." 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.

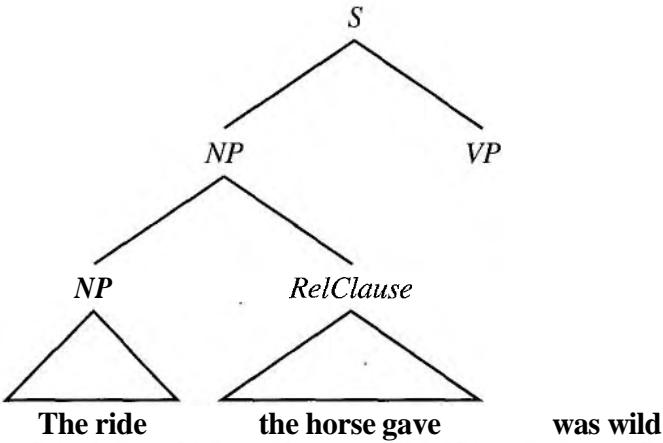


Figure 22.10 A left-corner parsing algorithm avoids predicting a *VP* starting with "ride," but does predict a *VP* starting with "was" because the grammar expects a *VP* following an *NP*. The triangle over "the horse gave" means that the words have a parse as a *RelClause*, but with additional intermediate 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 $[0, n, S \rightarrow a \bullet]$ 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, $(X \text{ and } (Y \text{ and } Z))$ versus $((X \text{ and } Y) \text{ and } Z)$. 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 $[ij, X \rightarrow \alpha \bullet]$ 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 $[ij, X \rightarrow \alpha \bullet]$ into the tree $[X : \alpha]$, we'll also look to see whether there are any other edges of the form $[a, j, X \rightarrow \beta \bullet]$. 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:

$$[S: [S: \left\{ \begin{array}{l} [NP: Fall leaves] [VP: fall] \\ [NP: Fall] [VP: leaves fall] \end{array} \right\}] \text{ and } [S: \left\{ \begin{array}{l} [NP: spring leaves] [VP: spring] \\ [NP: spring] [VP: leaves spring] \end{array} \right\}]]$$

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.

$\begin{aligned} S &\rightarrow NP_S VP \mid \dots \\ NP_S &\rightarrow Pronoun_S \mid Name \mid Noun \mid \dots \\ NP_O &\rightarrow Pronoun_O \mid Name \mid Noun \mid \dots \\ VP &\rightarrow VP NP_O \mid \dots \\ PP &\rightarrow Preposition NP_O \\ Pronouns &\rightarrow I \mid you \mid he \mid she \mid it \mid \dots \\ Pronoun_O &\rightarrow me \mid you \mid him \mid her \mid it \mid \dots \end{aligned}$
$\begin{aligned} S &\rightarrow NP(\text{Subjective}) VP \mid \dots \\ NP(\text{case}) &\rightarrow Pronoun(\text{case}) \mid Name \mid Noun \mid \dots \\ VP &\rightarrow VP NP(\text{Objective}) \mid \dots \\ PP &\rightarrow Preposition NP(\text{Objective}) \\ Pronoun(\text{Subjective}) &\rightarrow I \mid you \mid he \mid she \mid it \mid \dots \\ Pronoun(\text{Objective}) &\rightarrow me \mid you \mid him \mid her \mid it \mid \dots \end{aligned}$

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 .

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 *Pronouns* (which includes "I") and *PronounO* (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 .

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.

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.

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 .

AGREEMENT

AUGMENT

DEFINITE CLAUSE
GRAMMAR

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

$$NP(s_1) \ A \ VP(s_2) \Rightarrow S(s_1 + s_2)$$

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 $X \rightarrow Y Z \dots$ translates as $Y(s_1) \ A \ Z(s_2) \ A \ \dots \Rightarrow X(s_1 + s_2 + \dots)$.
- The notation $X \rightarrow Y \mid Z \mid \dots$ translates as $Y(s) \vee Z(s) \vee \dots \Rightarrow X(s)$.
- 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 $\{P(\dots)\}$ can appear on the right-hand side of a rule and translates verbatim into $P(\dots)$. This allows the grammar writer to insert a test for $P(\dots)$ without having the automatic string argument added.
- The notation $X \rightarrow \text{word}$ translates as $X([\text{word}])$.

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.

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

$$\text{Verb}([NP, PP]) \rightarrow \text{give} \mid \text{hand} \mid \dots$$

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 :

$$\begin{aligned} VP(subcat) &\rightarrow Verb(subcat) \\ &\quad | \quad VP(subcat + [NP]NP(Objective)) \\ &\quad | \quad VP(subcat + [Adjective]Adjective) \\ &\quad | \quad VP(subcat + [PP]PP) . \end{aligned}$$

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 :

$$\begin{aligned} VP(subcat) &\rightarrow VP(subcat) PP \\ &\quad | \quad VP(subcat) Adverb \end{aligned}$$

Generative capacity of augmented grammars

RULESCHEMA

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:

$$\begin{array}{ll} S(n) \rightarrow A(n) B(n) C(n) \\ A(1) \rightarrow a & A(n+1) \rightarrow a A(n) \\ B(1) \rightarrow b & B(n+1) \rightarrow b B(n) \\ C(1) \rightarrow c & C(n+1) \rightarrow c C(n) \end{array}$$

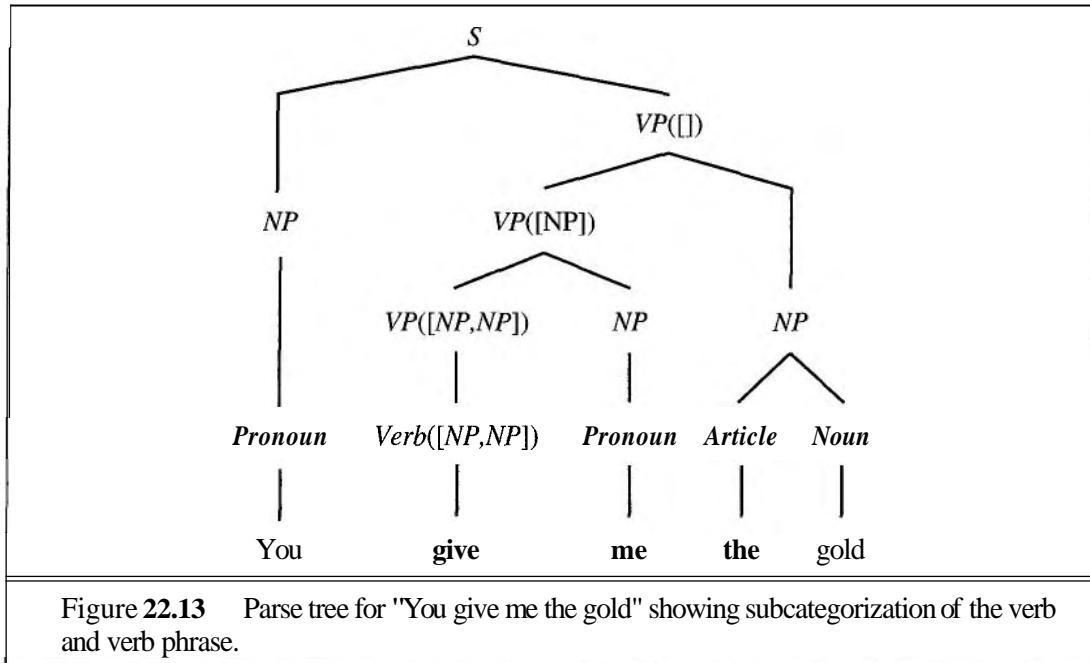


Figure 22.13 Parse tree for "You give me the gold" showing subcategorization of the verb and verb phrase.

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 *[xy]*, which is our notation for a square in the grid.

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 A Q* is determined by the meaning of *P*, *Q*, and *A*; 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

```

 $Exp(x) \rightarrow Exp(x_1) \text{ Operator}(op) Exp(x_2) \{x = Apply(op, x_1, x_2)\}$ 
 $Exp(x) \rightarrow (Exp(x))$ 
 $Exp(x) \rightarrow Number(x)$ 
 $Number(x) \rightarrow Digit(x)$ 
 $Number(x) \rightarrow Number(x_1) Digit(x_2) \{x = 10 \times x_1 + x_2\}$ 
 $Digit(x) \rightarrow x \{0 \leq x \leq 9\}$ 
 $Operator(x) \rightarrow x \{x \in \{+, -, \div, \times\}\}$ 

```

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.

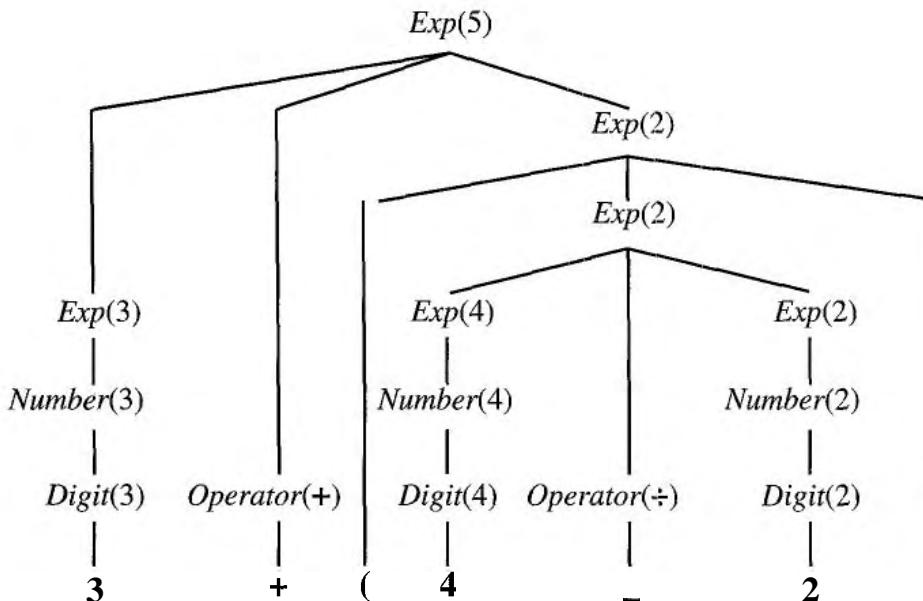


Figure 22.15 Parse tree with semantic interpretations for the string " $3 + (4 \div 2)$ ".

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 A-notation (see page 248), we can represent "loves Mary" as the predicate

$$\text{Ax } \text{Loves}(x, \text{Mary}) .$$

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*:"

$$S(\text{rel}(\text{obj})) \rightarrow \text{NP}(\text{obj}) \text{ VP}(\text{rel}) .$$

The rule tells us that the semantic interpretation of "John loves Mary" is

$$(\text{Ax } \text{Loves}(x, \text{Mary}))(\text{John}),$$

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 $\lambda y \text{ Ax } \text{Loves}(x, y)$, the predicate that, when given the argument *Mary*, returns the predicate *Ax 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.

$$\begin{aligned} S(\text{rel}(\text{obj})) &\rightarrow \text{NP}(\text{obj}) \text{ VP}(\text{rel}) \\ \text{VP}(\text{rel}(\text{obj})) &\rightarrow \text{Verb}(\text{rel}) \text{ NP}(\text{obj}) \\ \text{NP}(\text{obj}) &\rightarrow \text{Name}(\text{obj}) \\ \\ \text{Name}(\text{John}) &\rightarrow \text{John} \\ \text{Name}(\text{Mary}) &\rightarrow \text{Mary} \\ \text{Verb}(\lambda y \text{ Ax } \text{Loves}(x, y)) &\rightarrow \text{loves} \end{aligned}$$

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:

$$\begin{aligned} e \in \text{Loves}(\text{John}, \text{Mary}) \text{ A During}(\text{Now}, e); \\ e \in \text{Loves}(\text{John}, \text{Mary}) \text{ A After}(\text{Now}, e). \end{aligned}$$

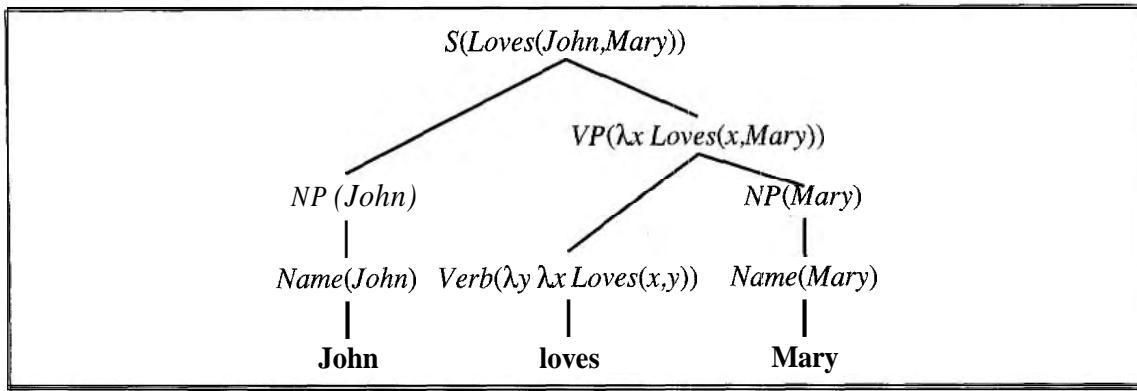


Figure 22.17 A parse tree with semantic interpretations for the string "John loves Mary".

This suggests that our two lexical rules for the words "loves" and "loved" should be these:

$$\begin{aligned} \text{Verb}(\lambda y \text{ Ax } e \in \text{Loves}(John, Mary) \wedge \text{During}(Now, e)) &\rightarrow \text{loves} ; \\ \text{Verb}(\lambda y \lambda x e \in \text{Loves}(x, y) \wedge \text{After}(Now, e)) &\rightarrow \text{loved} . \end{aligned}$$

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:

$$\begin{aligned} \forall a \ a \in \text{Agents} \Rightarrow \\ \exists w \ w \in \text{Wumpuses} \wedge \exists e \ e \in \text{Smell}(a, w) \wedge \text{During}(Now, e) ; \\ \exists w \ w \in \text{Wumpuses} \forall a \ a \in \text{Agents} \Rightarrow \\ \exists e \ e \in \text{Smell}(a, w) \wedge \text{During}(Now, e) . \end{aligned}$$

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:

$$\begin{aligned} \text{Every agent} \quad NP(\forall a \ a \in \text{Agents} \Rightarrow P) \\ \text{smells a wumpus} \quad VP(\exists w \ w \in \text{Wumpuses} \ A \\ \quad \quad \quad \exists e \ (e \in \text{Smell}(a, w) \wedge \text{During}(Now, e))) . \end{aligned}$$

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.'

INTERMEDIATE FORM

QUASI-LOGICAL FORM

QUANTIFIED TERM

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.

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

$$[\forall a a \in Agents].$$

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

$$\exists e (e \in Smell([\forall a a \in Agents] [\exists w w \in Wumpuses]) A During(Now, e))$$

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 *A* 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
<i>S</i>	<i>sentence</i>	I sleep.	$\exists e \ e \in Sleep(Speaker) \wedge During(Now, e)$
<i>NP</i>	<i>object</i>	a dog	$\exists d Dog(d)$
<i>PP</i>	$object^2 \rightarrow sentence$	in [2,2]	$\lambda x In(x, [2, 2])$
<i>RelClause</i>	$object \rightarrow sentence$	that sees me	$\lambda x \exists e \ e \in Sees(x, Speaker) \wedge During(Now, e)$
<i>VP</i>	$object^n \rightarrow sentence$	sees me	$\lambda x \exists e \ e \in Sees(x, Speaker) \wedge During(Now, e)$
<i>Adjective</i>	$object \rightarrow sentence$	smelly	$\lambda x Smelly(x)$
<i>Adverb</i>	$event \rightarrow sentence$	today	$\lambda e During(e, Today)$
<i>Article</i>	<i>quantifier</i>	the	$\exists!$
<i>Conjunction</i>	$sentence^2 \rightarrow sentence$	and	$\lambda p, q (p \wedge q)$
<i>Digit</i>	<i>object</i>	7	7
<i>Noun</i>	$object \rightarrow sentence$	wumpus	$\lambda x x \in Wumpuses$
<i>Preposition</i>	$object^2 \rightarrow sentence$	in	$\lambda x \forall y In(x, y)$
<i>Pronoun</i>	<i>object</i>	I	<i>Speaker</i>
<i>Verb</i>	$object^n \rightarrow sentence$	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:

$$\begin{aligned} \forall d \ d \in Dogs \Rightarrow \exists a \ a \in Days \wedge \exists e \ e \in Has(d, a, Now); \\ \exists a \ a \in Days \wedge \forall d \ d \in Dogs \Rightarrow \exists e \ e \in Has(d, a, Now). \end{aligned}$$

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.

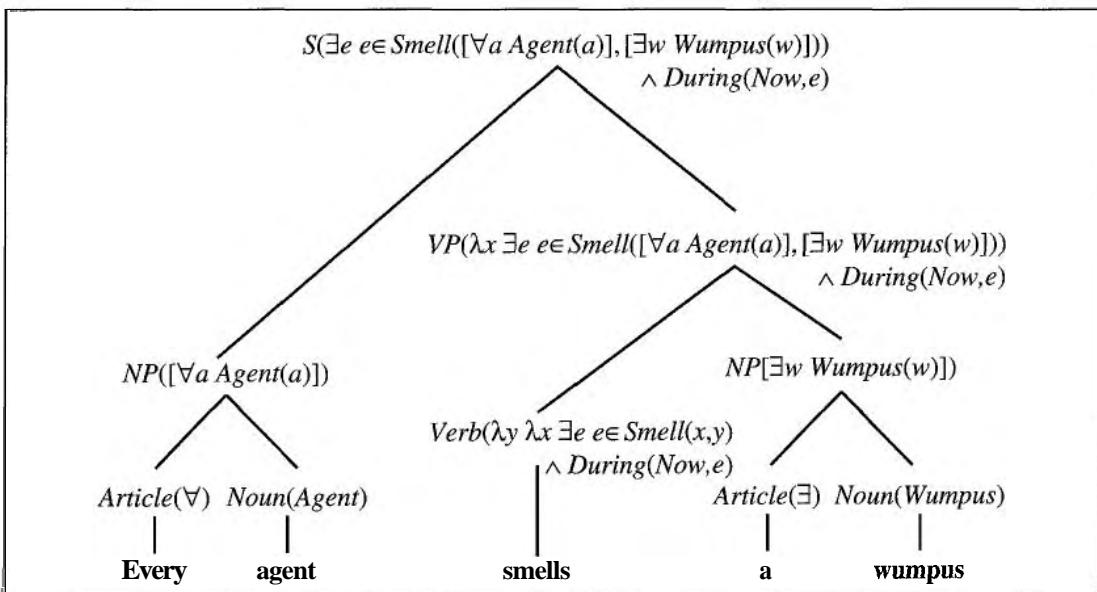


Figure 22.19 Parse tree for the sentence "Every agent smells a wumpus," showing both syntactic structure and semantic interpretations.

$$S(rel(obj)) \rightarrow NP(obj) VP(rel)$$

$$S(conj(sem_1, sem_2)) \rightarrow S(sem_1) Conjunction(conj) S(sem_2)$$

$$NP(sem) \rightarrow Pronoun(sem)$$

$$NP(sem) \rightarrow Name(sem)$$

$$NP([q x sem(x)]) \rightarrow Article(q) Noun(sem)$$

$$NP([q x obj A rel(x)]) \rightarrow NP([q x obj]) PP(rel)$$

$$NP([q x obj \wedge rel(x)]) \rightarrow NP([q x obj]) RelClause(rel)$$

$$NP([sem_1, sem_2]) \rightarrow Digit(sem_1) Digit(sem_2)$$

$$VP(sem) \rightarrow Verb(sem)$$

$$VP(rel(obj)) \rightarrow VP(rel) NP(obj)$$

$$VP(sem_1(sem_2)) \rightarrow VP(sem_1) Adjective(sem_2)$$

$$VP(sem_1(sem_2)) \rightarrow VP(sem_1) PP(sem_2)$$

$$RelClause(sem) \rightarrow \text{that } VP(sem)$$

$$PP(\lambda x rel(x, obj)) \rightarrow Preposition(rel) NP(obj)$$

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 $T((\text{Speaker} = \text{Agent}_B), \text{Now})$.

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:

$$\begin{aligned} S(\text{Statement}(\text{Speaker}, \text{rel}(\text{obj}))) &\rightarrow NP(\text{obj}) \text{ VP}(\text{rel}) \\ S(\text{Command}(\text{Speaker}, \text{rel}(\text{Hearer}))) &\rightarrow VP(\text{rel}). \end{aligned}$$

So the quasi-logical form for "Go to 2 2" is¹³

$$\text{Command}(\exists e \ e \in Go(\text{Hearer}, [2, 2])).$$

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 $\text{Append}([1, 2], [3], x)$ we have $x = [1, 2, 3]$ and to enumerate the values of x and y that make $\text{Append}(x, y, [1, 2, 3])$ 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 $VP(\text{rel, untensed})$ 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.



LEXICAL AMBIGUITY

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).

SYNTACTIC AMBIGUITY

SEMANTIC 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 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** 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

$$\exists x, e \ x = \text{Chrysler} \wedge e \in \text{Announce}(x) \wedge \text{After}(\text{Now}, e).$$

We need to change that to

$$\begin{aligned} \exists m, x, e \ x = \text{Chrysler} \wedge e \in \text{Announce}(m) \wedge \text{After}(\text{Now}, e) \\ \wedge \text{Metonymy}(m, x). \end{aligned}$$

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:

$$\forall m, x \ (m = x) \Rightarrow \text{Metonymy}(m, x).$$

For the Chrysler example, a reasonable generalization is that an organization can be used to stand for a spokesperson of that organization:

$$\forall m, x \ x \in \text{Organizations} \wedge \text{Spokesperson}(m, x) \Rightarrow \text{Metonymy}(m, x).$$

Other metonymies include the author for the works (I read Shakespeare) or more generally the produces 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.

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

$$\operatorname{argmax}_{\text{intent}} \text{Likelihood}(\text{intent} | \text{words}, \text{situation}),$$

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

$$\text{Assert}(\text{Speaker}, \neg(\text{Speaker} \in \text{Criminals}))$$

because this is a more likely thing to say.

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:

$$\begin{aligned} \text{Segment}(x) &\rightarrow S(x) \\ \text{Segment}(\text{CoherenceRelation}(x, y)) &\rightarrow \text{Segment}(x) \text{ Segment}(y) . \end{aligned}$$

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 "abcdcbcabcd." 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 $abcdbc$	$S \rightarrow abcdbc$ $S \rightarrow aAdA; A \rightarrow bc$	bc twice
7 $abcdcba$	$S \rightarrow aAdAa; A \rightarrow bc$	
8 $abcdcab$	$S \rightarrow aAdAab; A \rightarrow bc$	
9 $abcdcbcabc$	$S \rightarrow aAdAabc; A \rightarrow bc$ $S \rightarrow aAdAaA; A \rightarrow bc$ $S \rightarrow BdAB; A \rightarrow bc; B \rightarrow aA$	bc twice aA twice
10 $abcdcbcabcd$	$S \rightarrow BdABd; A \rightarrow bc; B \rightarrow aA$ $S \rightarrow CAC; A \rightarrow bc; B \rightarrow aA; C \rightarrow Bd$ $S \rightarrow CAC; A \rightarrow bc; C \rightarrow aAd$	Bd twice B only once

Figure 22.22 A trace of SEQUITUR inducing a grammar for the input text "abcdcbcabcd." 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, aAd .

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?

$\text{Answer}(c, \text{Capital}(s, c) \wedge \text{Largest}(p, \text{State}(s)) \wedge \text{Population}(s, p))$

CHILL's task is to learn a predicate $\text{Parse}(\text{words}, \text{query})$ 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.

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

ATTRIBUTE GRAMMAR

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** (Melčuk 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 Willts'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 $s_1, s_2 \dots s_n$, the correct grammar could be all-inclusive ($S \rightarrow \text{word}^*$), or it could be a copy of the input ($S \rightarrow s_1 \mid s_2 \mid \dots \mid s_n$), 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, Hornung 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 Schiitze (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).

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 $\exists x \ x \in \text{Wumpuses}$? 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:

<i>Pronoun</i> → someone	<i>V</i> → walked
<i>Adv</i> → slowly	<i>Prep</i> → to
<i>Det</i> → the	<i>Noun</i> → supermarket

Which of the following three grammars, combined with the lexicon, generates the given sentence? Show the corresponding parse tree(s).

(A):

$$S \rightarrow NP \ VP$$

$$NP \rightarrow Pronoun$$

$$NP \rightarrow Article \ Noun$$

$$VP \rightarrow VP \ PP$$

$$VP \rightarrow VP \ Adv \ Adv$$

$$VP \rightarrow Verb$$

$$PP \rightarrow Prep \ NP$$

$$NP \rightarrow Noun$$

(B):

$$S \rightarrow NP \ VP$$

$$NP \rightarrow Pronoun$$

$$NP \rightarrow Noun$$

$$NP \rightarrow Article \ NP$$

$$VP \rightarrow Verb \ Vmod$$

$$VP \rightarrow Verb \ Adv$$

$$Vmod \rightarrow Adv \ Vmod$$

$$Vmod \rightarrow Adv$$

$$Adv \rightarrow PP$$

$$PP \rightarrow Prep \ NP$$

$$PP \rightarrow Prep \ NP$$

(C):

$$S \rightarrow NP \ VP$$

$$NP \rightarrow Pronoun$$

$$NP \rightarrow Article \ NP$$

$$VP \rightarrow Verb \ Adv$$

$$Adv \rightarrow Adv \ Adv$$

$$Adv \rightarrow PP$$

$$PP \rightarrow Prep \ NP$$

$$NP \rightarrow Noun$$

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 *Buffaloⁿ*, 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.

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 *Buffaloⁿ*. 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 *Buffaloⁿ* for n up to 10. Extra credit: Carl de Marcken calculated that there are 121,030,872,213,055,159,681,184,485 *Buffaloⁿ* 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 ap-
proach 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 $P(w_i|w_{i-(n-1)} \dots w_{i-1})$. 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.

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

SMOOTHING
ADD-ONE
SMOOTHING

LINEAR
INTERPOLATION
SMOOTHING

probability the better. One problem with this approach is that $P(\text{words})$ 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:

$$\text{Perplexity}(\text{words}) = 2^{-\log_2(P(\text{words}))/N},$$

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.

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

Itiseasytoreadwordswithoutspace

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, $P(\text{word})$, and a string. Then, for each position i in the string, it stores in $\text{best}[i]$ the probability of the most probable string spanning from the start up to i . It also stores in $\text{words}[i]$ 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

$$\begin{aligned} P(\text{"without"}) &= 0.0004; \\ P(\text{"with"}) &= 0.005; P(\text{"out"}) = 0.0008; \\ P(\text{"with out"}) &= 0.005 \times 0.0008 = 0.000004. \end{aligned}$$

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

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  $\leftarrow$  LENGTH(text)
  words  $\leftarrow$  empty vector of length n + 1
  best  $\leftarrow$  vector of length n + 1, initially all 0.0
  best[0]  $\leftarrow$  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  $\leftarrow$  text[j:i]
      w  $\leftarrow$  LENGTH(word)
      if P[word]  $\times$  best[i - w]  $\geq$  best[i] then
        best[i]  $\leftarrow$  P[word]  $\times$  best[i - w]
        words[i]  $\leftarrow$  word
    /* Now recover the sequence of best words */
    sequence  $\leftarrow$  the empty list
    i  $\leftarrow$  n
    while i > 0 do
      push words[i] onto front of sequence
      i  $\leftarrow$  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, $P(\text{words})$, 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 $P(\text{"eat a banana"})$ and $P(\text{"eat a bandanna"})$ depends only on $P(\text{"banana"})$ versus $P(\text{"bandanna"})$ 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 phrase² 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.

$S \rightarrow NP VP [1.00]$
$NP \rightarrow Pronoun [0.10]$
$Name [0.10]$
$Noun [0.20]$
$Article\ Noun [0.50]$
$NP\ PP [0.10]$
$VP \rightarrow Verb [0.60]$
$VP\ NP [0.20]$
$VP\ PP [0.20]$
$PP \rightarrow Preposition\ NP [1.00]$
$Noun \rightarrow \text{breeze} [0.10] \mid \text{wumpus} [0.15] \mid \text{agent} [0.05] \mid \dots$
$Verb \rightarrow \text{sees} [0.15] \mid \text{smells} [0.10] \mid \text{goes} [0.25] \mid \dots$
$Pronoun \rightarrow \text{me} [0.05] \mid \text{you} [0.10] \mid \text{I} [0.25] \mid \text{it} [0.20] \mid \dots$
$Article \rightarrow \text{the} [0.30] \mid \text{a} [0.35] \mid \text{every} [0.05] \mid \dots$
$Preposition \rightarrow \text{to} [0.30] \mid \text{in} [0.25] \mid \text{on} [0.05] \mid \dots$

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.

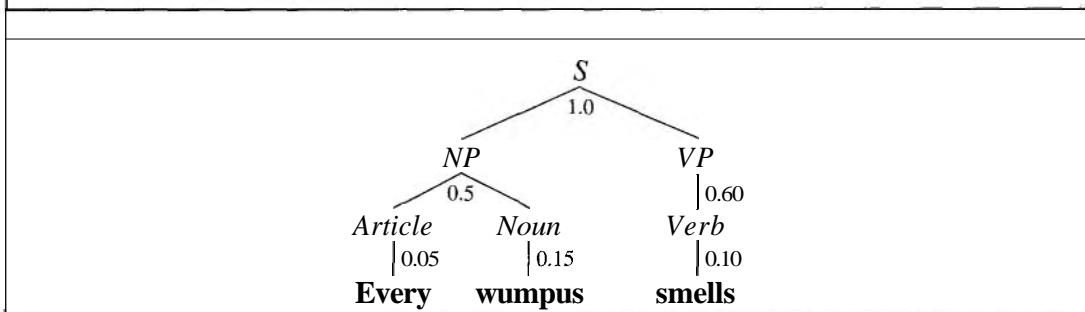


Figure 23.3 Parse tree for the sentence "Every wumpus smells," showing the probabilities of each subtree. The probability of the tree as a whole is $1.0 \times 0.5 \times 0.05 \times 0.15 \times 0.60 \times 0.10 = 0.000225$. Since this tree is the only parse of the sentence, that number is also the probability 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 $w_i \dots w_j$ is or is not generated by a rule $X \rightarrow a$. 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.

The inside–outside algorithm seems magical in that it induces a grammar from unparsed text. But it has several drawbacks. First, it is slow: $O(n^3 t^3)$, 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

$$\begin{aligned} X &\rightarrow Y Z \\ X &\rightarrow t, \end{aligned}$$

where X , Y and Z are nonterminals and 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.

23.2 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].
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.
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.

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

$$P(R = \text{true} | D, Q)$$

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 $P(R = \text{true} | D, Q)$. 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 $R = \text{true}$, 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 j th word in the query, we have

$$P(Q|D,r) = \prod_j P(Q_j|D,r).$$

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 i th 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 (2.3) 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 relevant 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
Not relevant	10	10

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.

ROC CURVE

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 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.

RECIPROCAL RANK

TIME TO ANSWER

CASE FOLDING

STEMMING

SYNONYMS

SPELLING CORRECTION

METADATA

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.

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 "Lebensversicherungsgesellschaft-sangestellter" (life insurance company employee). Languages such as Finnish, Turkish, Inuit, and Yupik have recursive morphological rules that in principle generate words of unbounded length.

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.

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.

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.

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

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.

DOCUMENT CLUSTERING

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.

AGGLOMERATIVE CLUSTERING

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 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 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."

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.

When the query is a single word (26% of the time, according to Silverstein *et al.* (1998)), processing is very fast. We make a single lookup in the lexicon to get the address of the hit list, and then we create an empty priority queue. After that, we go through the hit list one document at a time and check the count for the document. If the priority queue has fewer than R elements (where R is the size of the desired result set), we add the (document, count) pair to the queue. Otherwise, if the count is larger than that of the lowest entry in the priority queue, we delete the lowest entry and add the new (document, count) pair. Thus, answering the query takes time $O(H + R \log R)$, where H is the number of documents in the hit list. When the query has n words, we have to merge n hit lists, which can be done in time $O(nH + R \log R)$.

We have presented our theoretical overview of IR using the probabilistic model because that model makes use of the ideas we have already covered for other topics. But actual IR systems in practice are more likely to use a different approach called the **vector space model**. This model uses the same bag-of-words approach as the probability model. Each document is represented as a vector of unigram word frequencies. The query too is represented in the exact same way; the query [Bayes information retrieval model] is represented as the vector

$$[0, \dots, 1, 0, \dots, 1, 0, \dots, 1, 0, \dots, 1, 0, \dots]$$

where the idea is that there is one dimension for every word in the document collection and the query gets a score of 0 on every dimension except the four that actually appear in the query. Relevant documents are selected by finding the document vectors that are nearest neighbors to the query vector in vector space. One measure of similarity is the dot product between query vector and document vector; the larger this is, the closer the two vectors. Algebraically, this gives high scores for words that appear frequently in both document and query. Geometrically, the dot product between two vectors is equal to the cosine of the angle between the vectors; maximizing the cosine of two such vectors (in the same quadrant) means that the angle between them is close to 0.

There is much more to the vector space model than this. In practice, it has grown to accommodate a wide variety of extra features, refinements, corrections, and additions. The basic idea of ranking documents by their similarity in a vector space makes it possible to fold in new ideas into the numeric ranking system. Some argue that a probabilistic model would allow these same manipulations to be done in a more principled way, but IR researchers are unlikely to change unless they can see a clear performance advantage to another model.

To get an idea of the magnitude of the indexing problem for a typical IR task, consider a standard document collection from the TREC (Text REtrieval Conference) collection consisting of 750,000 documents totaling 2 GB (gigabytes) of text. The lexicon contains roughly 500,000 words after stemming and case folding; these words can be stored in 7 to 10 MB. The inverted index with (document, count) pairs takes 324 MB, although one can use compression techniques to get it down to 83 MB. Compression saves space, at the cost of slightly increased processing requirements. However, if compression allows you to keep the whole index in memory rather than storing it on disk, then it will yield a substantial net increase in performance. Support for phrase queries increases the size to about 1,200 MB uncompressed or 600 MB with compression. Web search engines work on a scale about 3000 times larger than this. Many of the issues are the same, but because it is impractical to deal with terabytes

of data on a single computer, the index is divided into k segments, with each segment stored on a different computer. A query is sent to all of the computers in parallel, and then the k result sets are merged into a single result set that is shown to the user. Web search engines also have to deal with thousands of queries per second, so they need n copies of the k computers. Values of k and n continue to grow over time.

23.3 INFORMATION EXTRACTION

INFORMATION EXTRACTION

Information extraction is the process of creating database entries by skimming a text and looking for occurrences of a particular class of object or event and for relationships among those objects and events. We could be trying to extract instances of addresses from web pages, with database fields for street, city, state, and zip code; or instances of storms from weather reports, with fields for temperature, wind speed, and precipitation. Information extraction systems are mid-way between information retrieval systems and full-text parsers, in that they need to do more than consider a document as a bag of words, but less than completely analyze every sentence.

The simplest type of information extraction system is called an *attribute-based* system because it assumes that the entire text refers to a single object and the task is to extract attributes of that object. For example, we mentioned in Section 10.5 the problem of extracting from the text “17in SXGA Monitor for only \$249.99” the database relations given by

$$\begin{aligned} \exists m \quad m \in \text{ComputerMonitors} \quad &A \text{ Size}(m, \text{Inches}(17)) \quad A \text{ Price}(m, \$249.99) \\ &A \text{ Resolution}(m, 1280 \times 1024). \end{aligned}$$

REGULAR EXPRESSIONS

Some of this information can be handled with the help of regular expressions, which define a regular grammar in a single text string. Regular expressions are used in Unix commands such as grep, in programming languages such as Perl, and in word processors such as Microsoft Word. The details vary slightly from one tool to another and so are best learned from the appropriate manual, but here we show how to build up a regular expression for prices in dollars, demonstrating common subexpressions:

[0-9]	matches any digit from 0 to 9
[0-9] +	matches one or more digits
. [0-9] [0-9]	matches a period followed by two digits
(. [0-9] [0-9]) ?	matches a period followed by two digits, or nothing
\$ [0-9] + (. [0-9] [0-9]) ?	matches \$249.99 or \$1.23 or \$1000000 or ...

Attribute-based extraction systems can be built as a series of regular expressions, one for each attribute. If a regular expression matches the text exactly once, then we can pull out the portion of the text that is the value of the attribute. If there is no match there's nothing more we can do, but if there are several matches, we need a process to choose among them. One strategy is to have several regular expressions for each attribute, ordered by priority. So, for example, the top priority regular expression for price might look for the string “our price:” immediately preceding the dollar sign; if that is not found, we fall back on a less reliable regular expression. Another strategy is to take all the matches and find some way

to choose between them. For example, we could take the lowest price that is within 50% of the highest price. This will handle texts like "List price \$99.00, special sale price \$78.00, shipping \$3.00."

One step up from attribute-based extraction systems are *relational-based* extraction systems, which have to worry about more than one object and the relations between them. Thus, when these systems see the text "\$249.99," they need to determine not just that it is a price, but also which object has that price. A typical relational-based extraction system is FASTUS, which handles news stories about corporate mergers and acquisitions. It can read the story:

Bridgestone Sports Co. said Friday it has set up a joint venture in Taiwan with a local concern and a Japanese trading house to produce golf clubs to be shipped to Japan.

and generate a database record like

$$\begin{aligned} e \in \text{Joint Ventures A Product}(e, \text{"golf clubs"}) \wedge \text{A Date}(e, \text{"Friday"}) \\ \wedge \text{Entity}(e, \text{"Bridgestone Sports Co"}) \wedge \text{Entity}(e, \text{"a local concern"}) \\ \wedge \text{Entity}(e, \text{"a Japanese trading house"}). \end{aligned}$$

CASCADED
FINITE-STATE
TRANSDUCERS

Relational extraction systems often are built by using **cascaded finite-state transducers**. That is, they consist of a series of finite-state automata (FSAs), where each automaton receives text as input, transduces the text into a different format, and passes it along to the next automaton. This is appropriate because each FSA can be efficient and because together they can extract the necessary information. A typical system is FASTUS, which consists of the following five stages:

1. Tokenization
2. Complex word handling
3. Basic group handling
4. Complex phrase handling
5. Structure merging

FASTUS's first stage is **tokenization**, which segments the stream of characters into tokens (words, numbers, and punctuation). For English, tokenization can be fairly simple; just separating characters at white space or punctuation does a fairly good job. For Japanese, tokenization would need to do segmentation, using something like the Viterbi segmentation algorithm. (See Figure 23.1.) Some tokenizers also deal with markup languages such as HTML, SGML, and XML.

The second stage handles **complex words**, including collocations such as "set up" and "joint venture," as well as proper names such as "Prime Minister Tony Blair" and "Bridgestone Sports Co." These are recognized by a combination of lexical entries and finite-state grammar rules. For example, a company name might be recognized by the rule

`CapitalizedWord+ ("Company" | "Co" | "Inc" | "Ltd")`

These rules should be constructed with care and tested for recall and precision. One commercial system recognized "Intel Chairman Andy Grove" as a place rather than a person because of a rule of the form

`CapitalizedWord+ ("Grove" | "Forest" | "Village" | ...)`

The third stage handles **basic groups**, meaning noun groups and verb groups. The idea is to chunk these into units that will be managed by the later stages. A noun group consists of a head noun, optionally preceded by determiners and other modifiers. Because the noun group does not include the full complexity of the *NP* in \mathcal{E}_1 we do not need recursive context-free grammar rules: the regular grammar rules allowed in finite state automata suffice. The verb group consists of a verb and its attached auxiliaries and adverbs, but without the direct and indirect object and prepositional phrases. The example sentence would emerge from this stage as follows:

1 NG: Bridgestone Sports Co.)	10 NG: a local concern
2 VG: said	11 CJ: and
3 NG: Friday	12 NG: a Japanese trading house
4 NG: it	13 VG: to produce
5 VG: had set up	14 NG: golf clubs
6 NG: a joint venture	15 VG: to be shipped
7 PR: in	16 PR: to
8 NG: Taiwan	17 NG: Japan
9 PR: with	

Here NG means noun group, VG is verb group, PR is preposition, and CJ is conjunction.

The fourth stage combines the basic groups into **complex phrases**. Again, the aim is to have rules that are finite-state and thus can be processed quickly, and that result in unambiguous (or nearly unambiguous) output phrases. One type of combination rule deals with domain-specific events. For example, the rule

Company+ SetUp JointVenture ("with" Company+)?

captures one way to describe the formation of a joint venture. This stage is the first one in the cascade where the output is placed into a database template as well as being placed in the output stream.

The final stage **merges structures** that were built up in the previous step. If the next sentence says "The joint venture will start production in January," then this step will notice that there are two reference to a joint venture, and that they should be merged into one.

In general, information extraction works well for a restricted domain in which it is possible to predetermine what subjects will be discussed, and how they will be mentioned. It has proven useful in a number of domains, but is not a substitute for full-scale natural language processing.

23.4 MACHINE TRANSLATION

Machine translation is the automatic translation of text from one natural language (the source) to another (the target). This process has proven to be useful for a number of tasks, including the following:

1. **Rough translation**, in which the goal is just to get the gist of a passage. Ungrammatical and inelegant sentences are tolerated as long as the meaning is clear. For example, in

Web surfing, a user is often happy with a rough translation of a foreign web page. Sometimes a monolingual human can post-edit the output without having to read the source. This type of machine-assisted translation saves money because such editors can be paid less than bilingual translators.

2. Restricted-source translation, in which the subject matter and format of the source text are severely limited. One of the most successful examples is the TAUM-METEO system, which translates weather reports from English to French. It works because the language used in weather reports is highly stylized and regular.
3. Preedited translation, in which a human preeditsthe source document to make it conform to a restricted subset of English (or whatever the original language is) before machine translation. This approach is particularly cost-effective when there is a need to translate one document into many languages, as is the case for legal documents in the European Community or for companies that sell the same product in many countries. Restricted languages are sometimes called "Caterpillar English," because Caterpillar Corp. was the first firm to try writing its manuals in this form. Xerox defined a language for its maintenance manuals which was simple enough that it could be translated by machine into all the languages Xerox deals with. As an added benefit, the original English manuals became clearer as well.
4. Literary translation, in which all the nuances of the source text are preserved. This is currently beyond the state of the art for machine translation.

As an example of rough translation, the SYSTRAN translation service translated the first paragraph of this chapter into Italian and back to English as follows:

Italian: In capitolo 22 abbiamo visto come un agente potrebbe comunicare con un altro agente (essere umano o software) che usando le espressioni in un linguaggio reciprocamente accordato. Completare sintattico e l'analisi semantica delle espressioni. necessaria da estrarre il significato completo del utterances ed è possibile perch6 le espressioni sono corte e limitate ad un settore limitato.

English: In chapter 22 we have seen as an agent could communicate with an other agent (to be human or software) that using the expressions in a language mutual come to an agreement. Complete syntactic and the semantic analysis of the expressions is necessary to extract the complete meant one of the utterances and is possible because the expressions short and are limited to a dominion.

Translation is difficult because, in the general case, it requires in-depth understanding of the text, and that in turn requires in-depth understanding of the situation that is being communicated. This is true even for very simple texts—even "texts" of one word. Consider the word "Open" on the door of a store.⁴ It communicates the idea that the store is accepting customers at the moment. Now consider the same word "Open" on a large banner outside a newly constructed store. It means that the store is now in daily operation, but readers of this sign would not feel misled if the store closed at night without removing the banner. The two

⁴ This example is due to Martin Kay.

signs use the identical word to convey different meanings. In German, on the other hand, the sign on the door would be "Offen" while the banner would read "Neu Eroffnet."

The problem is that different languages categorize the world differently. For example, the French word "doux" covers a wide range of meanings corresponding approximately to the English words "soft," "sweet," and "gentle." Similarly, the English word "hard" covers virtually all uses of the German word "hart" (physically recalcitrant, cruel) and some uses of the word "schwierig" (difficult). The German verb "heilen" covers the medical uses of the English word "cure," as well as the transitive and intransitive uses of "heal." Therefore, representing the meaning of a sentence is more difficult for translation than it is for single-language understanding. A single-language parsing system could use predicates like $Open(x)$, but for translation, the representation language would have to make more distinctions, perhaps with $Open_1(x)$ representing the "Offen" sense and $Open_2(x)$ representing the "Neu Eroffnet" sense. A representation language that makes all the distinctions necessary for a set of languages is called an **interlingua**.

To do fluent translation, a translator (human or machine) must read the original text, understand the situation to which it is referring, and find a corresponding text in the target language that does a good job of describing the same or a similar situation. Often, this involves a choice. For example, the English word "you," when referring to a single person, can be translated into French as either the formal "vous" or the informal "tu." There is just no way that one can refer to the concept of "you" in French without also making a choice of formal or informal. Translators (both machine and human) sometimes find it difficult to make this choice.

Machine translation systems

Machine translation systems vary in the level to which they analyze the text. Some systems attempt to analyze the input text all the way into an interlingua representation (as we did in Chapter 22) and then generate sentences in the target language from that representation. This is difficult because it includes the complete language understanding problem as a subproblem, to which is added the difficulty of dealing with an interlingua. It is brittle because if the analysis fails, there is no output. It does have the advantage that there is no part of the system that relies on knowledge of two languages at once. That means that one can build an interlingua system to translate among n languages with $O(n^2)$ work instead of $O(n^2)$.

Other systems are based on a **transfer**. They keep a data base of translation rules (or examples), and whenever the rule (or example) matches, they translate directly. Transfer can occur at the lexical, syntactic, or semantic level. For example, a strictly syntactic rule maps English [Adjective Noun] to French [Noun Adjective]. A mixed syntactic and lexical rule maps French [S_1 "et puis" S_2] to English [S_1 "and then" S_2]. A transfer that goes directly from one sentence to another is called a **memory-based translation** method, because it relies on memorizing a large set of (English, French) pairs. The transfer method is robust in that it will always generate *some* output, and at least some of the words are bound to be right. Figure 23.5 diagrams the various transfer points.

INTERLINGUA

TRANSFER

MEMORY-BASED
TRANSLATION

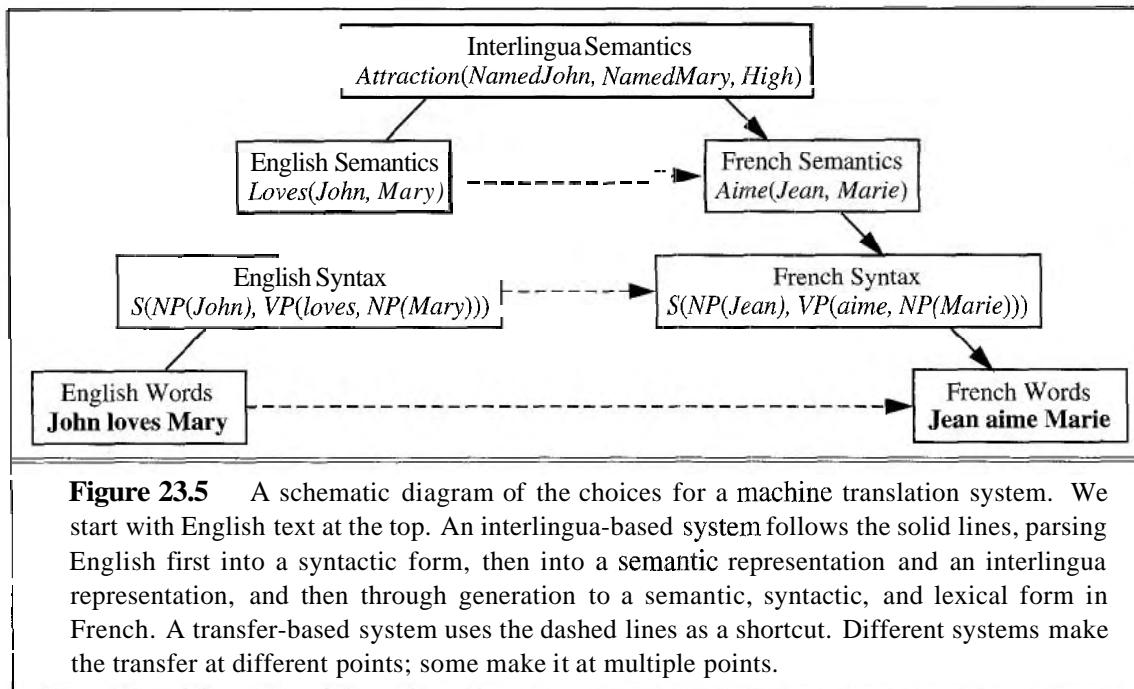


Figure 23.5 A schematic diagram of the choices for a machine translation system. We start with English text at the top. An interlingua-based system follows the solid lines, parsing English first into a syntactic form, then into a semantic representation and an interlingua representation, and then through generation to a semantic, syntactic, and lexical form in French. A transfer-based system uses the dashed lines as a shortcut. Different systems make the transfer at different points; some make it at multiple points.

Statistical machine translation

In the early 1960s, there was great hope that computers would be able to translate from one natural language to another, just as Turing's project "translated coded German messages into intelligible German. By 1966, it became clear that fluent translation requires an understanding of the meaning of the message, whereas code breaking does not.

In the last decade, there has been a move towards statistically based machine translation systems. Of course, any of the steps in Figure 23.5 could benefit from the application of statistical data and from a clear probabilistic model of what constitutes a good analysis or transfer. But "statistical machine translation" has come to denote an approach to the whole translation problem that is based on finding the most probable translation of a sentence, using data gathered from a bilingual corpus. As an example of a bilingual corpus, **Hansard**⁵ is a record of parliamentary debate. Canada, Hong Kong, and other countries produce bilingual Hansards, the European Union publishes its official documents in 11 languages, and the United Nations publishes multilingual documents. These have proven to be invaluable resources for statistical machine translation.

We can express the problem of translating an English sentence E into, say, a French⁶ sentence F by the following application of Bayes' rule::

$$\begin{aligned} \operatorname{argmax}_F P(F|E) &= \operatorname{argmax}_F P(E|F)P(F)/P(E) \\ &= \operatorname{argmax}_F P(E|F)P(F). \end{aligned}$$

⁵ Named after William Hansard, who first published the British parliamentary debates in 1811.

⁶ Throughout this section we consider the problem of translating from English to French. Do not be confused by the fact that Bayes' rule leads us to consider $P(E|F)$ rather than $P(F|E)$, making it seem as if we were translating French to English.

This rule says we should consider all possible French sentences F and choose the one that maximizes the product $P(E|F)P(F)$. The factor $P(E)$ can be ignored because it is the same for every F . The factor $P(F)$ is the **language model** for French; it says how probable a given sentence is in French. $P(E|F)$ is the **translation model**; it says how probable an English sentence is as a translation, given a French sentence.

The astute reader will wonder what we have gained from defining $P(F|E)$ in terms of $P(E|F)$. In other applications of Bayes' rule, we made this reversal because we wanted to use a causal model. For example, we use the causal model $P(Symptoms|Disease)$ to compute $P(Disease|Symptoms)$. With translation, however, neither direction is more causal than the other. The reason for applying Bayes' rule in this case is that we believe we will be able to learn a language model $P(F)$ that is more accurate than the translation model $P(E|F)$ (and more accurate than estimating $P(F|E)$ directly). Essentially, we have divided the problem into two parts: first we use the translation model $P(E|F)$ to find candidate French sentences that mention the right concepts from the English sentence, but that might not be fluent French; then we use the language model $P(F)$ (for which we have much better probability estimates) to pick out the best candidate.

The **language model** $P(F)$ can be any model that gives a probability to a sentence. With a very large corpus, we could estimate $P(F)$ directly by counting how many times each sentence appears in the corpus. For example, we use the Web to collect 100 million French sentences, and if the sentence "Clique ici" appears 50,000 times, then $P(\text{Clique ici})$ is .0005. But even with 100 million examples, most sentence counts will be zero.⁷ Therefore, we will use the familiar **bigram** language model, in which the probability of a French sentence consisting of the words $f_1 \dots f_n$ is

$$P(f_1 \dots f_n) = \prod_{i=1}^n P(f_i|f_{i-1}).$$

We will need to know bigram probabilities such as $P(\text{Eiffel}|\text{tour}) = .02$. This captures only a very local notion of syntax, where a word depends on just the previous word. However, for rough translation that is often sufficient.⁸

The **translationmodel**, $P(E|F)$, is more difficult to come by. For one thing, we don't have a ready collection of (English, French) sentence pairs from which to train. For another, the complexity of the model is greater, because it considers the cross product of sentences rather than just individual sentences. We will start with an overly simplistic translation model and build up to something approximating the "IBM Model 3" (Brown *et al.*, 1993) which might still seem overly simplistic, but has proven to generate acceptable translations roughly half the time.

The overly simplistic model is "to translate a sentence, just translate each word individually and independently, in left-to-right order." This is a unigram word choice model. It

⁷ If there are just 100,000 words in the lexicon, then 99.99999% of the three-word sentences have a count of zero in the 100 million sentence corpus. It gets worse for longer sentences.

⁸ For the finer points of translation, $P(f_i|f_{i-1})$ is clearly not enough. As a famous example, Marcel Proust's 3500 page novel *A la recherche du temps perdu* begins and ends with the same word, so some translators have decided to do the same, thus basing the translation of a word on one that appeared roughly 2 million words earlier.

makes it easy to compute the probability of a translation:

$$P(E|F) = \prod_{i=1}^n P(E_i|F_i).$$

In a few cases this model works fine. For example, consider

$$P(\text{the dog}| \text{le chien}) = P(\text{the}| \text{le}) \times P(\text{dog}| \text{chien}).$$

Under any reasonable set of probability values, "the dog" would be the maximum likelihood translation of "le chien." In most cases, however, the model fails. One problem is word order. In French, "dog" is "chien" and "brown" is "brun," but "brown dog" is "chien brun." Another problem is that word choice is not a one-to-one mapping. The English word "home" is often translated as "a la maison," a one-to-three mapping (or three-to-one in the other direction). Despite these problems, IBM Model 3 stubbornly sticks to a basically unigram model, but adds a few complications to patch it up.

To handle the fact that words are not translated one for one, the model introduces the notion of the **fertility** of a word. A word with fertility n gets copied over n times, and then each of those n copies gets translated independently. The model contains parameters for $P(\text{Fertility} = n | \text{word})$ for each French word. To translate "à la maison" to "home," the model would choose fertility 0 for "a" and "la" and fertility 1 for "maison" and then use the unigram translation model to translate "maison" to "home." This seems reasonable enough: "à" and "la" are low-content words that could reasonably translate to nothing. Translating in the other direction is more dubious. The word "home" would be assigned fertility 3, giving us "home home home." The first "home" would translate to "à," the second to "la" and the third to "maison." In terms of the translation model, "a la maison" would get the exact same probability as "maison la à." (That's the dubious part.) It would be up to the language model to decide which is better. It might seem to make more sense to make "home" translate directly to "à la maison," rather than indirectly via "home home home," but that would require many more parameters, and they would be hard to obtain from the available corpus.

The final part of the translation model is to permute the words into the right positions. This is done by a model of the offsets by which a word moves from its original position to its final position. For example, in translating "chien brim" to "brown dog," the word "brown" gets an offset of +1 (it is moved one position to the right) and "dog" gets an offset of -1. You might imagine that the offset should be dependent on the word: adjectives like "brown" would tend to have a positive offset because French tends to put the adjectives after the noun. But IBM Model 3 decided that making offsets dependant on the word would require too many parameters, so the offset is independent of the word and dependent only on the position within the sentence, and the length of the sentences in both languages. That is, the model estimates the parameters

$$P(\text{Offset} = o | \text{Position} = p, \text{EngLen} = m, \text{FrLen} = n).$$

So to determine the offset for "brown" in "brown dog," we consult $P(\text{Offset}|1, 2, 2)$, which might give us, say, +1 with probability .3 and 0 with probability .7. The offset model seems even **more** dubious, as if it had been concocted by someone more familiar with moving magnetic words around on a refrigerator than with actually speaking a natural language. We will see shortly that it was designed that way not because it is a good model of language, but be-

cause it makes reasonable use of the available data. In any case, it serves to remind us vividly that a mediocre translation model can be saved by a good French language model. Here is an example showing all the steps in translating a sentence:

Source French:	Le	chien	brun	n'	est	pas	all6	à	la	maison
Fertility model:	1	1	1	1	1	0	1	0	0	1
Transformed French:	Le	chien	brun	n'	est		all6			maison
Word choice model:	The	dog	brown	not	did		go			home
Offset model:	0	+1	-1	+1	-1	0		0		
Target English:	The	brown	dog	did	not		go			home

Now, we know how to compute the probability $P(F|E)$ for any pair of (French, English) sentences. But what we really want to do is, given an English sentence, find the French sentence which maximizes that probability. We can't just enumerate sentences; with 10^5 words in French, there are 10^{5n} sentences of length n , and many alignments for each one. Even if we consider only the 10 most frequent word-to-word translations for each word, and only consider offsets of 0 or ± 1 , we still get about $2^{n/2}10^n$ sentences, which means that we could enumerate them all for $n = 5$, but not for $n = 10$. Instead, we need to *search* for the best solution. A version of A* search has proven effective; see Germann *et al.* (2001).

Learning probabilities for machine translation

We have outlined a model for $P(F|E)$ that involves four sets of parameters:

- Language model: $P(\text{word}_i|\text{word}_{i-1})$
- Fertility model: $P(\text{Fertility} = n|\text{word}_F)$
- Word choice model: $P(\text{word}_E|\text{word}_F)$
- Offset model: $P(\text{Offset} = o|\text{pos}, \text{len}_E, \text{len}_F)$

Even with a modest vocabulary of 1,000 words, this model requires millions of parameters. Obviously, we will have to learn them from data. We will assume that the only data available to us is a bilingual corpus. Here is how to use it:

Segment into sentences: The unit of translation is a sentence, so we will have to break the corpus into sentences. Periods are strong indicators of the end of a sentence, but consider "Dr. J. R. Smith of Rodeo Dr. arrived."; only the final period ends a sentence. Sentence segmentation can be done with about 98% accuracy.

Estimate the French language model $P(\text{word}_i|\text{word}_{i-1})$: Considering just the French half of the corpus, count the frequencies of word pairs and do smoothing to give an estimate of $P(\text{word}_i|\text{word}_{i-1})$. For example we might have $P(\text{Eiffel}|\text{tour}) = .02$.

Align sentences: For each sentence in the English version, determine what sentence(s) it corresponds to in the French version. Usually, the next sentence of English corresponds to the next sentence of French in a 1:1 match, but sometimes there is variation: one sentence in one language **will** be split into a 2:1 match, or the order of two sentences will be swapped, resulting in a **2:2** match. By looking at the sentence lengths alone, it is possible to align them (1:1, 1:2, or 2:2, etc.) with accuracy in the 90% to 99% range using a variation on the Viterbi segmentation algorithm (Figure 23.1). Even better alignment can be achieved by

using landmarks that are common to both languages, such as numbers or proper names, or words that we know have an unambiguous translation from a bilingual dictionary.

Now we are ready to estimate the parameters of the translation model. We will do that by first making a poor initial guess and then improving it.

Estimate the initial fertility model $P(Fertility = n | word_F)$: Given a French sentence of length m that is aligned to an English sentence of length n , consider this as evidence that each French word has fertility n/m . Consider all the evidence over all sentences to get a fertility probability distribution for each word.

Estimate the initial word choice model $P(word_E | word_F)$: Look at all the French sentences that contain, say, "brun." The words that appear most frequently in the English sentences aligned with these sentences are the likely word-to-word translations of "brun."

Estimate the initial offset model $P(Offset = o | pos, len_E, len_F)$: Now that we have the word choice model, use it to estimate the offset model. For an English sentence of length n that is aligned to a French sentence of length m , look at each French word in the sentence (at position i) and at each English word in the sentence (at position j) that is a likely word choice for the French word, and consider that as evidence for $P(Offset = i - j | i, n, m)$.

Improve all the estimates: Use EM (expectation–maximization) to improve the estimates. The hidden variable will be a **word alignment vector** between sentence-aligned sentence pairs. The vector gives, for each English word, the position in the French sentence of the corresponding French word. For example, we might have the following:

Source French: Le chien brun n' est pas allé à la maison

Target English: The brown dog did not go home

Word alignment: 1 3 2 5 4 7 10

First, using the current estimates of the parameters, create a word alignment vector for each sentence pair. This will allow us to make better estimates. The fertility model is estimated by counting how many times a member of the word alignment vector goes to multiple words or to zero words. The word choice model now can look only at words that are aligned to each other, rather than at all words in the sentence, and the offset model can look at each position in the sentence to see how often it moves according to the word alignment vector. Unfortunately, we don't know for sure what the correct alignment is, and there are too many of them to enumerate. So we are forced to search for a few high-probability alignments and weight them by their probabilities when we collect evidence for the new parameter estimates. That is all we need for the EM algorithm. From the initial parameters we compute alignments and from the alignments we improve the parameter estimates. Repeat until convergence.

WORD ALIGNMENT VECTOR

23.5 SUMMARY

The main points of this chapter are:

1. Probabilistic language models based on n-grams recover a surprising amount of information about a language.

2. CFGs can be extended to probabilistic CFGs, making it easier to learn them from data and easier to do disambiguation.
 3. **Information retrieval** systems use a very simple language model based on bags of words, yet still manage to perform well in terms of **recall** and **precision** on very large corpora of text.
 4. **Information extraction** systems use a more complex model that includes limited notions of syntax and semantics. They are often implemented using a cascade of finite state automata.
 5. **Machine translation** systems have been implemented using a range of techniques, from full syntactic and semantic analysis to statistical techniques based on word frequencies.
 6. In building a statistical language system, it is best to devise a model that can make good use of available data, even if the model seems overly simplistic.
-

BIBLIOGRAPHICAL AND HISTORICAL NOTES

n-gram letter models for language modeling were proposed by Markov (1913). Claude Shannon (Shannon and Weaver, 1949) was the first to generate n-gram word models of English. Chomsky (1956, 1957) pointed out the limitations of finite-state models compared with context-free models, concluding "Probabilistic models give no particular insight into some of the basic problems of syntactic structure." This is true, but it ignores the fact that probabilistic models do provide insight into some *other* basic problems—problems that CFGs do not address. Chomsky's remarks had the unfortunate effect of scaring many people away from statistical model for two decades, until these models reemerged for use in speech recognition (Jelinek, 1976).

Add-one smoothing is due to Jeffreys (1948), and deleted interpolation smoothing is due to Jelinek and Mercer (1980), who used it for speech recognition. Other techniques include Witten–Bell smoothing (1991) and Good–Turing smoothing (Church and Gale, 1991). The later also arises frequently in bioinformatics problems. Biostatistics and probabilistic NLP are coming closer together, as each deals with long, structured sequences chosen from an alphabet of constituents.

Simple n-gram letter and word models are not the only possible probabilistic models. Blei *et al.* (2001) describe a probabilistic text model called **latent Dirichlet allocation** that views a document as a mixture of topics, each with its own distribution of words. This model can be seen as an extension and rationalization of the **latent semantic indexing** model of (Deerwester *et al.*, 1990) (see also Papadimitriou *et al.* (1998)) and is also related to the multiple cause mixture model of (Sahami *et al.*, 1996).

Probabilistic context-free grammars (PCFGs) answer all of Chomsky's objections about probabilistic models, and have advantages over CFGs. PCFGs were investigated by Booth (1969) and Salomaa (1969). Jelinek (1969) presents the stack decoding algorithm, a variation of Viterbi search that can be used to find the most probable parse with a PCFG.

Baker (1979) introduced the inside–outside algorithm, and Lari and Young (1990) described its uses and limitations. Charniak (1996) and Klein and Manning (2001) discuss parsing with **treebank** grammars. Stolcke and Omohundro (1994) show how to learn grammar rules with Bayesian model merging. Other algorithms for PCFGs are presented by Charniak (1993) and by Manning and Schütze (1999). Collins (1999) offers a survey of the field and an explanation of one of the most successful programs for statistical parsing.

Unfortunately, PCFGs perform worse than simple n-gram models on a variety of tasks, because PCFGs cannot represent information associated with individual words. To correct for that deficiency, several authors (Collins, 1996; Charniak, 1997; Hwa, 1998) have introduced versions of **lexicalized probabilistic grammars**, which combine context-free and word-based statistics.

The Brown Corpus (Francis and Kucera, 1967) was the first effort to collect a balanced corpus of text for empirical linguistics. It contained about a million words, tagged with part of speech. It was originally stored on 100,000 punched cards. The Penn treebank (Marcus *et al.*, 1993) is a collection of about 1.6 million words, hand-parsed into trees. It is stored on a CD. The British National Corpus (Leech *et al.*, 2001) extends that to 100 million words. The World Wide Web has over a trillion words. It is stored on over 10 million servers.

The field of **information retrieval** is experiencing a regrowth in interest, sparked by the wide usage of Internet searching. Robertson (1977) gives an early overview, and introduces the probability ranking principle. Manning and Schütze (1999) give a short introduction to IR in the context of statistical approaches to NLP. Baeza-Yates and Ribeiro-Neto (1999) is a general-purpose overview, replacing older classics by Salton and McGill (1983) and by Frakes and Baeza-Yates (1992). The book *Managing Gigabytes* (Witten *et al.*, 1999) does just what the title says: explains how to efficiently index, compress, and make queries on corpora in the gigabyte range. The TREC conference, organized by the U.S. government's National Institute of Standards and Technology (NIST), hosts an annual competition for IR systems and publishes proceedings with results. In the first seven years of the competition performance roughly doubled.

The most popular model for IR is the **vector space model** (Salton *et al.*, 1975). Salton's work dominated the early years of the field. There are two alternative probabilistic models. The one we presented is based on the work of Ponte and Croft (1998). It models the joint probability distribution $P(D, Q)$ in terms of $P(Q|D)$. An alternative model (Maron and Kuhns, 1960; Robertson and Sparck Jones, 1976) uses $P(D|Q)$. Lafferty and Zhai (2001) show that the models are based on the same joint probability distribution, but that the choice of model has implications for training the parameters. Our presentation is derived from theirs. Turtle and Croft (1992) compare the various IR models.

Brin and Page (1998) describe the implementation of a search engine for the World Wide Web, including the PAGERANK algorithm, a query-independent measure of document quality based on an analysis of Web links. Kleinberg (1999) describes how to find authoritative sources on the Web using link analysis. Silverstein *et al.* (1998) investigate a log of a billion Web searches. Kukich (1992) surveys the literature on spelling correction. Porter (1980) describes the classic rule-based stemming algorithm, and Krovetz (1993) describes a dictionary-based version.

Manning and Schütze (1999) provide a good overview of document classification and clustering. Joachims (2001) uses statistical learning theory and support vector machines to give a theoretical analysis of when classification will be successful. Apté et al. (1994) report an accuracy of 96% in classifying Reuters news articles into the "Earnings" category. Koller and Sahami (1997) report accuracy up to 95% using a naive Bayes classifier, and up to 98.6% using a Bayes classifier that accounts for some dependencies among features. Lewis (1998) surveys forty years of application of naive Bayes techniques to text classification and retrieval.

The journal *Information Retrieval* and the proceedings of the annual *SIGIR* conference cover recent developments in the field.

Early information extraction programs include GUS (Bobrow et al., 1977) and FRUMP (DeJong, 1982). Some of the design of modern information extraction systems can be traced to work on semantic grammars in the 1970s and 1980s. For example, an interface to an airline reservation system with a semantic grammar would have categories like *Location* and *FlyTo* instead of *NP* and *VP*. See Birnbaum and Selfridge (1981) for an implementation of a system based on semantic grammars.

Recent information extraction has been pushed forward by the annual Message Understand Conferences (MUC), sponsored by the U.S. government. The FASTUS system was done by Hobbs et al. (1997); the collection of papers in which it appears (Roche and Schabes, 1997) lists other systems using finite state models.

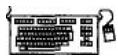
In the 1930s Petr Troyanskii applied for a patent for a "translating machine," but there were no computers available to implement his ideas. In March 1947, the Rockefeller Foundation's Warren Weaver wrote to Norbert Wiener, suggesting machine translation might be possible. Drawing on work in cryptography and information theory, Weaver wrote, "When I look at an article in Russian, I say: 'This is really written in English, but it has been coded in strange symbols. I will now proceed to decode.'" For the next decade, the community tried to decode in this way. IBM exhibited a rudimentary system in 1954. Bar-Hillel (1960) and Locke and Booth (1955) describe the enthusiasm of this period. Later disillusionment with machine translation is described by Lindsay (1963), who also points out some of the obstacles to machine translation having to do with the interaction between syntax and semantics and with the need for world knowledge. The U.S. government became disappointed in the lack of progress, and a report (ALPAC, 1966) concluded "there is no immediate or predictable prospect of useful machine translation." However, limited work continued, and the SYSTRAN system was deployed by the U.S. Air Force in 1970 and by the European Community in 1976. the TAUM-METEO weather translation system was also deployed in 1976 (Quinlan and O'Brien, 1992). Starting in the 1980s, computer power had increased to the point where the ALPAC findings were no longer correct. Voorhees (1993) reports some recent translation applications based on Wordnet. A textbook introduction is given by Hutchins and Somers (1992).

Statistical machine translation harkens back to Weaver's 1947 note, but it was only in the 1980s that it became practical. Our presentation was based on the work of Brown and his colleagues at IBM (Brown et al., 1988, 1993). It is very mathematical, so the accompanying tutorial by Kevin Knight (1999) is a breath of fresh air. More recent work on statistical

machine translation goes beyond the bigram model to models that include some syntax (Yamada and Knight, 2001). Early work on sentence segmentation was done by Palmer and Hearst (1994). Michel and Plamondon (1996) cover bilingual sentence alignment.

There are two excellent books on probabilistic language processing: Charniak (1993) is brief and to the point while Manning and Schütze (1999) is comprehensive and up to date. Work on practical language processing is presented at the biennial Applied Natural Language Processing conference (ANLP), the conference on Empirical Methods in Natural Language Processing (EMNLP), and the journal *Natural Language Engineering*. SIGIR sponsors a newsletter and an annual conference on information retrieval.

EXERCISES



STYLOMETRY

23.1 (Adapted from Jurafsky and Martin (2000).) In this exercise we will develop a classifier for authorship: given a text, it will try to determine which of two candidate authors wrote the text. Obtain samples of text from two different authors. Separate them into training and test sets. Now train a unigram word model for each author on the training set. Finally, for each test set, calculate its probability according to each unigram model and assign it to the most probable model. Assess the accuracy of this technique. Can you improve its accuracy with additional features? This subfield of linguistics is called **stylometry**; its successes include the identification of the author of the *Federalist Papers* (Mosteller and Wallace, 1964) and some disputed works of Shakespeare (Foster, 1989).



23.2 This exercise explores the quality of the n-gram model of language. Find or create a monolingual corpus of about 100,000 words. Segment it into words, and compute the frequency of each word. How many distinct words are there? Plot the frequency of words versus their rank (first, second, third, ...) on a log–log scale. Also, count frequencies of bigrams (two consecutive words) and trigrams (three consecutive words). Now use those frequencies to generate language: from the unigram, bigram, and trigram models, in turn, generate a 100-word text by making random choices according to the frequency counts. Compare the three generated texts with actual language. Finally, calculate the perplexity of each model.



23.3 This exercise concerns the detection of spam email. Spam is defined as unsolicited bulk commercial email messages. Dealing with spam is an annoying problem for many email users, so a reliable way of eliminating it would be a boon. Create a corpus of spam email and one of non-spam mail. Examine each corpus and decide what features appear to be useful for classification: unigram words? bigrams? message length, sender, time of arrival? Then train a classification algorithm (decision tree, naive Bayes, or some other algorithm of your choosing) on a training set and report its accuracy on a test set.

23.4 Create a test set of five queries, and pose them to three major Web search engines. Evaluate each one for precision at 1, 3, and 10 documents returned and for mean reciprocal rank. Try to explain the differences.



23.5 Try to ascertain which of the search engines from the previous exercise are using case folding, stemming, synonyms, and spelling correction.

23.6 Estimate how much storage space is necessary for the index to a billion-page corpus of Web pages. Show the assumptions you made.

23.7 Write a regular expression or a short program to extract company names. Test it on a corpus of business news articles. Report your recall and precision.

23.8 Select five sentences and submit them to an online translation service. Translate them from English to another language and back to English. Rate the resulting sentences for grammaticality and preservation of meaning. Repeat the process; does the second round of iteration give worse results or the same results? Does the choice of intermediate language make a difference to the quality of the results?

23.9 Collect some examples of time expressions, such as "two o'clock," "midnight," and "12:46." Also think up some examples that are ungrammatical, such as "thirteen o'clock" or "half past two fifteen." Write a grammar for the time language.

23.10 (Adapted from Knight (1999).) The IBM Model 3 machine translation model assumes that, after the word choice model proposes a list of words and the offset proposes possible permutations of the words, the language model can choose the best permutation. This exercise investigates how sensible that assumption is. Try to unscramble these proposed sentences into the correct order:

- have programming a seen never I language better
- loves john mary
- is the communication exchange of intentional information brought by about the production perception of and signs from drawn a of system signs conventional shared

Which ones could you do? What type of knowledge did you draw upon? Train a bigram model from a training corpus, and use it to find the highest-probability permutation of some sentences from a test corpus. Report on the accuracy of this model.

23.11 If you look in an English–French dictionary, the translation for "hear" is the verb "entendre." But if you train the IBM Model 3 on the Canadian Hansard, the most probable translation for "hear" is "Bravo." Explain why that is, and estimate what the fertility distribution for "hear" might be. (Hint: you might want to look at some Hansard text. Try a web search for [Hansard hear].)

24 PERCEPTION

In which we connect the computer to the raw, unwashed world.

PERCEPTION
SENSORS

Perception provides agents with information about the world they inhabit. Perception is initiated by sensors. A sensor is anything that can record some aspect of the environment and pass it as input to an agent program. The sensor could be as simple as a one-bit sensor that detects whether a switch is on or off or as complex as the retina of the human eye, which contains more than a hundred million photosensitive elements. In this chapter, our focus will be on vision, because that is by far the most useful sense for dealing with the physical world.

24.1 INTRODUCTION

There are a variety of sensory modalities that are available to artificial agents. Those they share with humans include vision, hearing, and touch. Hearing, at least for speech, was covered in Section 15.6. Touch, or tactile sensing, is discussed in Chapter 25, where we examine its use in dexterous manipulation by robots, and the rest of this chapter will cover vision. Some robots can perceive modalities that are not available to the unaided human, such as radio, infrared, GPS, and wireless signals. Some robots do active sensing, meaning they send out a signal, such as radar or ultrasound, and sense the reflection of this signal off of the environment.

There are two ways that an agent can use its percepts. In the feature extraction approach, agents detect some small number of features in their sensory input and pass them directly to their agent program, which can act reactively to the features, or can combine them with other information. The wumpus agent worked in this mode, with five sensors each extracting a one-bit feature. It is now known that a fly extracts features from the optical flow and feeds them directly to muscles that help it steer, allowing it to react and change direction within 30 milliseconds.

The alternative is a model-based approach, wherein the sensory stimulus is used to reconstruct a model of the world. In this approach we start with a function f that maps from the state of the world, W , to the stimulus, S , that the world will produce:

$$S = f(W).$$

The function f is defined by physics and optics, and is fairly well understood. Generating S from f and a real or imaginary world W is the problem addressed by **computer graphics**. Computer vision is in some sense the inverse of computer graphics: given f and S , we try to compute W with

$$W = f^{-1}(S).$$

Unfortunately, f does not have a proper inverse. For one thing, we cannot see around corners, so we cannot recover all aspects of the world from the stimulus. Moreover, even the part we can see is enormously ambiguous: without additional information we can not tell if S is an image of a toy Godzilla destroying a two-foot tall model building, or a real monster destroying a two-hundred foot building. We can address some of these issues by building a probability distribution over worlds, rather than trying to find a unique world:

$$P(W) = P(W|S)P(S).$$

A more important drawback with this type of modeling is that it is solving too difficult a problem. Consider that in computer graphics it can take several hours of computation to render a single frame of a movie, that 24 frames are needed per second, and that computing f^{-1} is more difficult than computing f . Clearly this is too much computation for a supercomputer, let alone a fly, to react in real time. Fortunately, the agent does not need a model of the level of detail used in photorealistic computer graphics. The agent need only know whether there is a tiger hiding in the brush, not the precise location and orientation of every hair on the tiger's back.

For most of this chapter, we will see how to recognize objects, such as tigers, and we will see ways to do this without representing every last detail of the tiger. In Section 24.2 we study the process of image formation, defining some aspects of the $f(W)$ function. First we look at the geometry of the process. We will see that light reflects off objects in the world, and onto points in the image plane in the sensor of an agent. The geometry explains why a large Godzilla far away looks like a small Godzilla up close. Then we look at the photometry of the process, which describes how light in the scene determines the brightness of points in the image. Together, geometry and photometry give us a model of how objects in the world will map into a two-dimensional array of pixels.

With an understanding of how images are formed, we then turn to how they are processed. The flow of information in visual processing in both humans and computers can be divided into three phases. In early or low-level vision (Section 24.3) the raw image is smoothed to eliminate noise, and features of the two-dimensional image are extracted, particularly edges between regions. In mid-level vision these edges are grouped together to form two-dimensional regions. In high-level vision (Section 24.4), the two-dimensional regions are recognized as actual objects in the world (Section 24.5). We study various cues in the image that can be harnessed to this end, including motion, stereopsis, texture, shading, and contour. Object recognition is important to an agent in the wild to detect tigers, and it is important for industrial robots to distinguish nuts from bolts. Finally, Section 24.6 describes how the recognition of objects can help us perform useful tasks, such as manipulation and navigation. Manipulation means being able to grab and use tools and other objects, and navigation means being able to move from place to place without bumping into anything. By keeping these

tasks in mind we can make sure that an agent builds only as much of a model as it needs to achieve its goals.

24.2 IMAGE FORMATION

SCENE	Vision gathers light scattered from objects in a scene and creates a two-dimensional image
IMAGE	on an image plane. The image plane is coated with photosensitive material: Rhodopsin molecules in the retina, silver halides on photographic film, and a charge-coupled device (CCD) array in a digital camera. Each site in a CCD integrates the electrons released by photon absorption for a fixed time period. In a digital camera the image plane is subdivided into a rectangular grid of a few million pixels . The eye has a similar array of pixels consisting of about 100 million rods and 5 million cones, arranged in a hexagonal array.
PIXELS	

The scene is very large and the image plane is quite small, so there needs to be some way of focusing the light onto the image plane. This can be done with or without a lens. Either way, the key is to define the geometry so that we can tell where each point in the scene will end up in the image plane.

Images without lenses: the pinhole camera

PINHOLE CAMERA	The simplest way to form an image is with a pinhole camera , which consists of a pinhole opening, O , at the front of a box, and an image plane at the back of the box (Figure 24.1). We will use a three-dimensional coordinate system with the origin at O , and will consider a point P in the scene, with coordinates (X, Y, Z) . P gets projected to the point P' in the image plane with coordinates (x, y, z) . If f is the distance from the pinhole to the image plane, then by similar triangles, we can derive the following equations:
----------------	---

$$\frac{-x}{f} = \frac{X}{Z}, \frac{-y}{f} = \frac{Y}{Z} \Rightarrow x = \frac{-fX}{Z}, y = \frac{-fY}{Z}$$

PERSPECTIVE PROJECTION	These equations define an image formation process known as perspective projection . Note that the Z in the denominator means that the farther away an object is, the smaller its image will be. Also, note that the minus signs mean that the image is inverted, both left-right and up-down, compared with the scene.
------------------------	---

Under perspective projection, parallel lines converge to a point on the horizon. (Think of railway tracks.) Let us see why this must be so. A line in the scene passing through the point (X_0, Y_0, Z_0) in the direction (U, V, W) can be described as the set of points $(X_0 + \lambda U, Y_0 + \lambda V, Z_0 + \lambda W)$, with λ varying between $-\infty$ and $+\infty$. The projection of a point P_λ from this line onto the image plane is given by

$$\left(f \frac{X_0 + \lambda U}{Z_0 + \lambda W}, f \frac{Y_0 + \lambda V}{Z_0 + \lambda W} \right).$$

VANISHING POINT	As $X \rightarrow \infty$ or $X \rightarrow -\infty$, this becomes $p_\infty = (fU/W, fV/W)$ if $W \neq 0$. We call p_∞ the vanishing point associated with the family of straight lines with direction (U, V, W) . Lines with the same direction share the same vanishing point.
-----------------	---

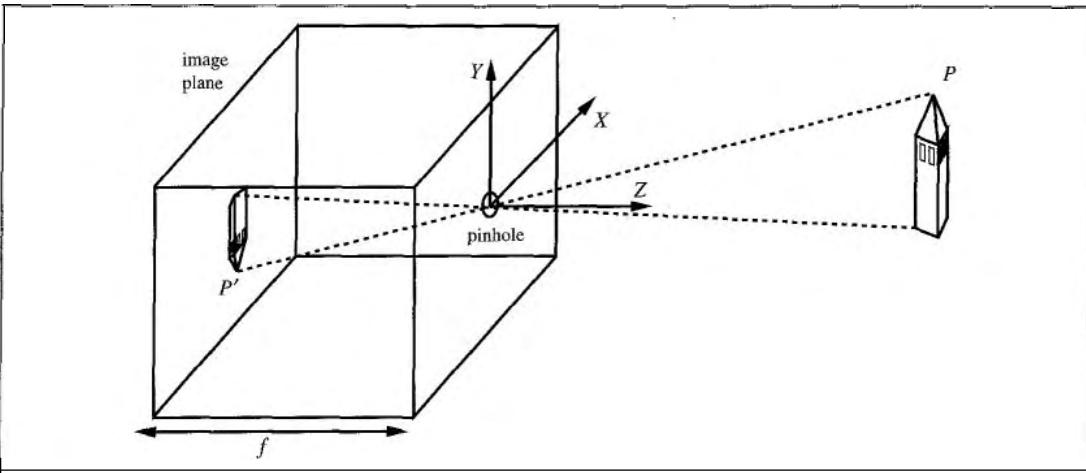


Figure 24.1 Geometry of image formation in the pinhole camera.

SCALED
ORTHOGRAPHIC
PROJECTION

If the object is relatively shallow compared with its distance from the camera, we can approximate perspective projection by **scaled orthographic projection**. The idea is as follows: If the depth Z of points on the object varies within some range $Z_0 \pm \Delta Z$, with $\Delta Z \ll Z_0$, then the perspective scaling factor f/Z can be approximated by a constant $s = f/Z_0$. The equations for projection from the scene coordinates (X, Y, Z) to the image plane become $x = sX$ and $y = sY$. Note that scaled orthographic projection is an approximation that is valid only for those parts of the scene with not much internal depth variation; it should not be used to study properties "in the large." An example to convince you of the need for caution: under orthographic projection, parallel lines stay parallel instead of converging to a vanishing point!

Lens systems

LENS

Vertebrate eyes and modern cameras use a **lens**. A lens is much wider than a pinhole, enabling it to let in more light. This is paid for by the fact that not all the scene can be in sharp focus at the same time. The image of an object at distance Z in the scene is produced at a fixed distance from the lens Z' , where the relation between Z and Z' is given by the lens equation

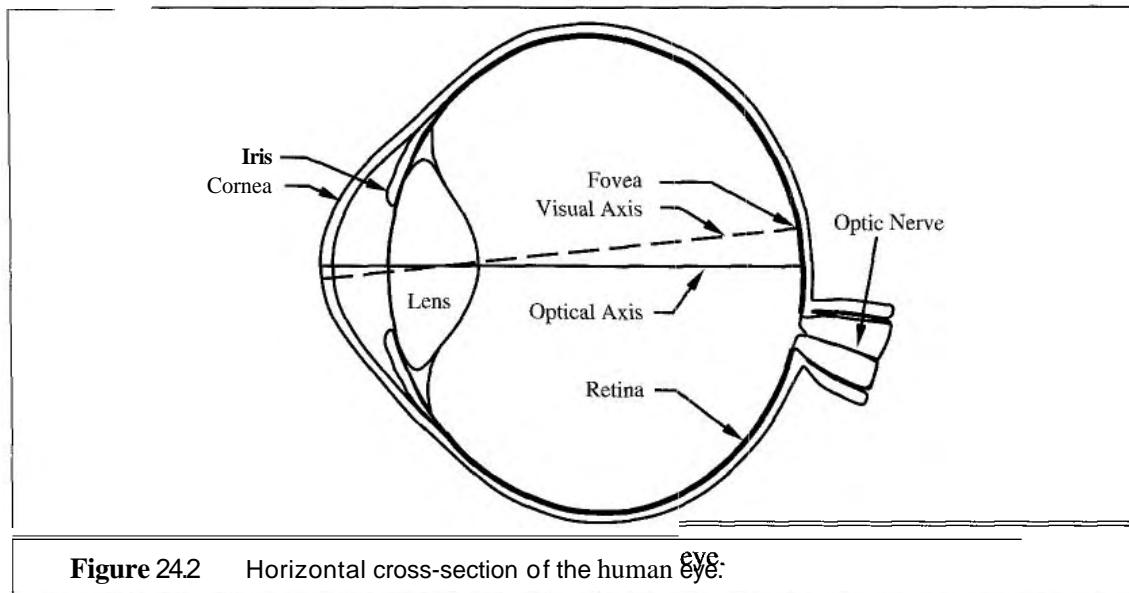
$$\frac{1}{Z} + \frac{1}{Z'} = \frac{1}{f},$$

in which f is the focal length of the lens. Given a certain choice of image distance Z'_0 between the nodal point of the lens and the image plane, scene points with depths in a range around Z_0 , where Z_0 is the corresponding object distance, will be imaged in reasonably sharp focus. This range of depths in the scene is referred to as the **depth of field**.

Note that, because the object distance Z is typically much greater than the image distance Z' or f , we often make the following approximation:

$$\frac{1}{Z} + \frac{1}{Z'} \approx \frac{1}{Z'} \Rightarrow \frac{1}{Z'} \approx \frac{1}{f}.$$

DEPTH OF FIELD



Thus, the image distance $Z' \approx f$. We can therefore continue to use the pinhole camera perspective projection equations to describe the geometry of image formation in a lens system.

In order to focus objects that are at different distances Z , the lens in the eye (see Figure 24.2) changes shape, whereas the lens in a camera moves in the 2-direction.

Light: the photometry of image formation

PHOTOMETRY

Light is a crucial prerequisite for vision; without light, all images would be uniformly dark, no matter how interesting the scene. **Photometry** is the study of light. For our purposes, we will model how the light in the scene maps into the intensity of light in the image plane over time, which we denote as $I(x, y)$.¹ A vision system uses this model backwards, going from the intensity of images to properties of the world. Figure 24.3 shows a digitized image of a stapler on a desk, and a close-up of a 12×12 block of pixels extracted from the stapler image. A computer program trying to interpret the image would have to start from a matrix of intensity values like this.

The brightness of a pixel in the image is proportional to the amount of light directed toward the camera by the surface patch in the scene that projects to the pixel. This in turn depends on the reflectance properties of the surface patch and on the position and distribution of the light sources in the scene. There is also a dependence on the reflectance properties of the rest of the scene, because other scene surfaces can serve as indirect light sources by reflecting light.

SPECULAR
REFLECTION

We can model two different kinds of reflection. **Specular reflection** means that light is reflected from the outer surface of the object, and obeys the constraint that the angle of reflection is equal to the angle of incidence. This is the behavior of a perfect mirror. **Diffuse reflection** means that the light penetrates the surface of the object, is absorbed by the object,

DIFFUSE
REFLECTION

¹ When we are concerned with changes over time we will use $I(x, y, t)$.

and is then re-emitted. For a perfectly diffusing (or **Lambertian**) surface the light scatters with equal intensity in all directions. The intensity depends only on the angle of incidence of the light source: a light source directly overhead will reflect the most light, and a light source that is almost parallel to the surface will reflect almost no light. In between those two extremes the reflected intensity, I , obeys Lambert's cosine law,

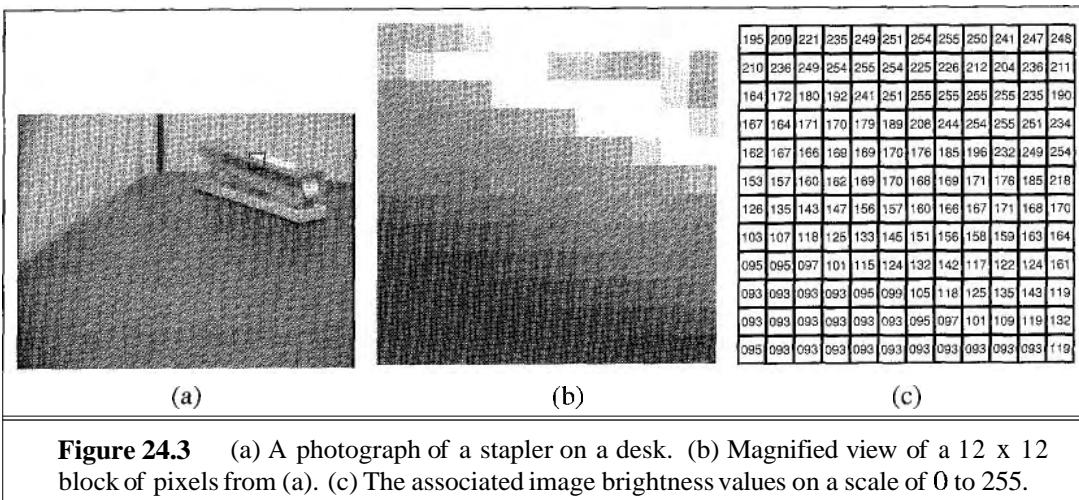
$$I = kI_0 \cos \theta ,$$

where I_0 is the intensity of the light source, θ is the angle between the light source and the surface normal, and k is a constant called the **albedo**, which depends on the reflective properties of the surface. It varies from 0 (for perfectly black surfaces) to 1 (for pure white surfaces).

In real life, surfaces exhibit a combination of diffuse and specular properties. Modeling this combination on the computer is the bread and butter of computer graphics. Rendering realistic images is usually done by ray tracing, which aims to simulate the physical process of light originating from light sources and being reflected and re-reflected multiple times.

Color: the spectrophotometry of image formation

In Figure 24.3 we showed a black-and-white picture, merrily ignoring the fact that visible light comes in a range of wavelengths—ranging from 400 nm on the violet end of the spectrum to 700 nm on the red end. Some light consists of a single wavelength, corresponding to a color of the rainbow. But other light is a mixture of different wavelengths. Does that mean we need a mixture of values for our $I(x, y)$ measure, rather than a single value? If we wanted to represent the physics of light exactly, we would indeed. But if we want only to duplicate the perception of light by humans (and many other vertebrates) we can compromise. Experiments (going back to Thomas Young in 1801) have shown that any mixture of wavelengths, no matter how complex, can be duplicated by a mixture of just three primary colors. That is, if you have a light generator that can linearly combine three wavelengths (typically, we choose red (700 nm), green (546 nm) and blue (436 nm)), then by adjusting knobs to give more of



one color and less of another you can match any combination of wavelengths, as far as human visual perception is concerned. This experimental fact means that images can be represented with a vector of just three intensity numbers per pixel: one for each of the three primary wavelengths. In practice, one byte each results in a high-fidelity reproduction of the image. This trichromatic perception of color is related to the fact that the retina has three types of cones with receptivity peaks at 650 nm, 530 nm, and 430 nm, respectively, but the exact details of the relationship is more complex than a one-to-one mapping.

24.3 EARLY IMAGE PROCESSING OPERATIONS

We have seen how light reflects off objects in the scene to form an image consisting of, say, five million three-byte pixels. As with all sensors there will be noise in the image, and in any case there is a lot of data to deal with. In this section we see what can be done to the image data to make it easier to deal with. We will first look at the operations of smoothing the image to reduce noise, and of detecting edges in the image. These are called "early" or "low-level" operations because they are the first in a pipeline of operations. Early vision operations are characterized by their local nature (they can be carried out in one part of the image without regard for anything more than a few pixels away) and by their lack of knowledge: we can smooth images and detect edges without having any idea what objects are in the images. This makes the low-level operations good candidates for implementation in parallel hardware, either *in vivo* or *in silicon*. We will then look at one mid-level operation, segmenting the image into regions. This phase of operations is still operating on the image, not the scene, but it includes non-local processing.

In Section 15.2, **smoothing** meant predicting the value of a state variable at some time t in the past, given evidence from t and from other times up to the present. Now we apply the same idea to the spatial domain rather than the temporal: smoothing means predicting the value of a pixel given the surrounding pixels. Notice that we must keep straight the difference between the observed value measured at a pixel, and the true value that should have been measured at that pixel. These can be different because of random measurement errors or because of a systematic failure—the receptor in the CCD may have gone dead.

One way to smooth an image is to assign to each pixel the average of its neighbors. This will tend to cancel out extreme values. But how many neighbors should we consider—one pixel away, or two, or more? One answer that works well for canceling out Gaussian noise is a weighted average using a **Gaussian filter**. Recall that the Gaussian function with standard deviation σ is

$$G_\sigma(x) = \frac{1}{\sqrt{2\pi\sigma^2}} e^{-x^2/2\sigma^2} \quad \text{in one dimension, or}$$

$$G_\sigma(x, y) = \frac{1}{\sqrt{2\pi\sigma^2}} e^{-(x^2+y^2)/2\sigma^2} \quad \text{in two dimensions.}$$

Applying a Gaussian filter means replacing the intensity $I(x_0, y_0)$ with the sum, over all (x, y) pixels, of $I(x, y)G_\sigma(d)$, where d is the distance from (x_0, y_0) to (x, y) . This kind of weighted sum is so common that there is a special name and notation for it. We say that the function h is the **convolution** of two functions f and g (denoted as $h = f * g$) if we have

GAUSSIAN FILTER

CONVOLUTION

$$h(x) = \sum_{u=-\infty}^{+\infty} f(u)g(x-u) \quad \text{in one dimension, or}$$

$$h(x,y) = \sum_{u=-\infty}^{+\infty} \sum_{v=-\infty}^{+\infty} f(u,v)g(x-u,y-v) \quad \text{in two dimensions.}$$

So the smoothing function is achieved by convolving the image with the Gaussian, $\mathbf{I} * \mathbf{G}_\sigma$. A σ of 1 pixel is enough to smooth over a small amount of noise, whereas 2 pixels will smooth a larger amount, but at the loss of some detail. Because the Gaussian's influence fades at a distance, in practice we can replace the $\pm\infty$ in the sums with something like $\pm 3\sigma$.

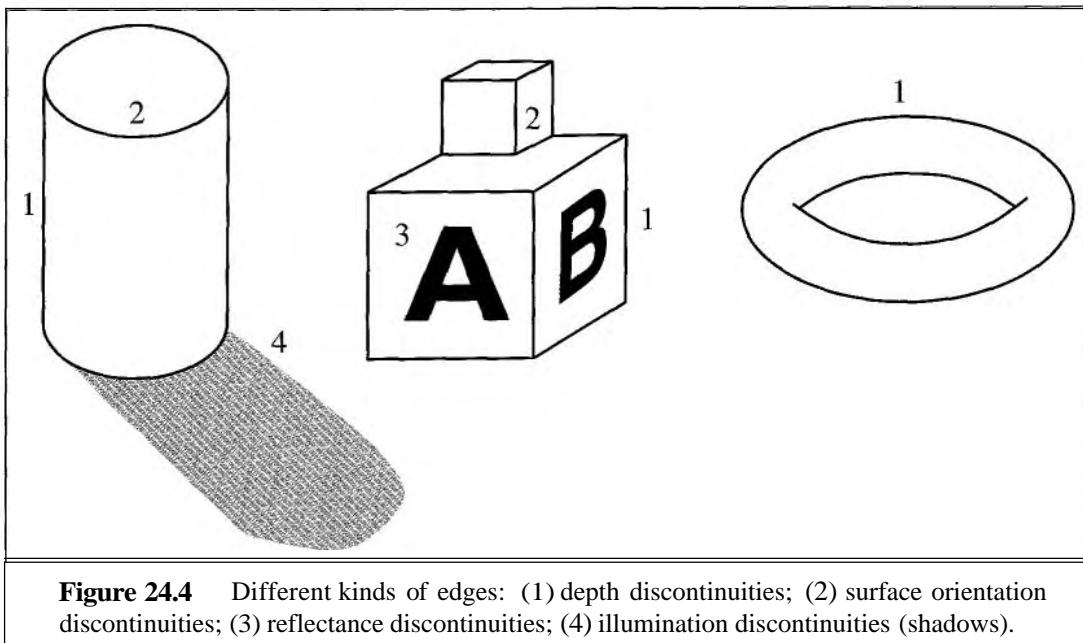
Edge detection

EDGES

The next step in early vision is to detect edges in the image plane. **Edges** are straight lines or curves in the image plane across which there is a "significant" change in image brightness. The goal of edge detection is to abstract away from the messy, multi-megabyte image and towards a more compact, abstract representation, as in Figure 24.4. The motivation is that edge contours in the image correspond to important scene contours. In the figure we have three examples of depth discontinuity, labelled 1; two surface-orientation discontinuities, labelled 2; a reflectance discontinuity, labelled 3; and an illumination discontinuity (shadow), labelled 4.

Edge detection is concerned only with the image, and thus does not distinguish between these different types of discontinuities in the scene, but later processing will.

Figure 24.5(a) shows an image of a scene containing a stapler resting on a desk, and (b) shows the output of an edge detection algorithm on this image. As you can see, there is a difference between the output and an ideal line drawing. The small components edges do not all align with each other, there are gaps where no edge appears, and there are "noise" edges



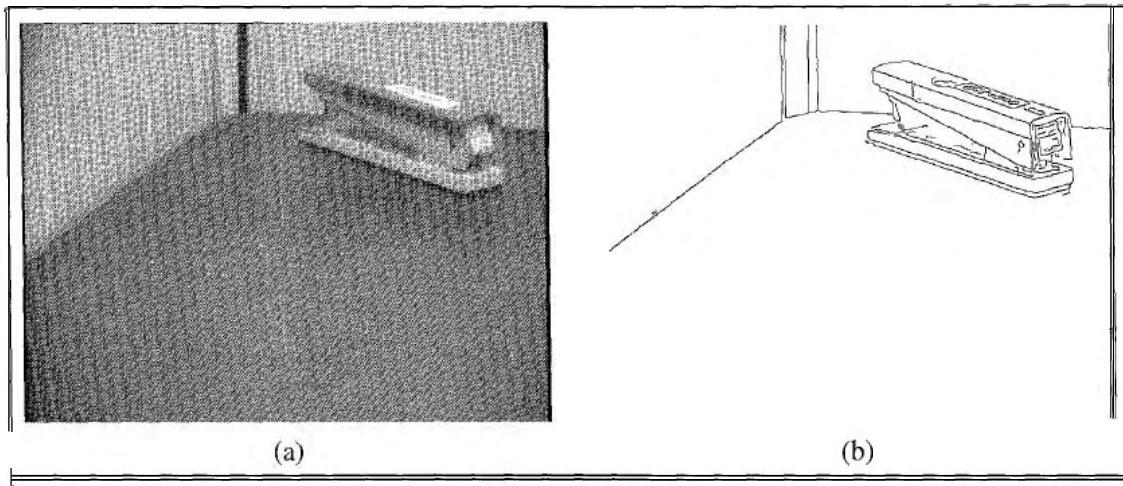


Figure 24.5 (a) Photograph of a stapler (b) Edges computed from (a).

that do not correspond to anything of significance in the scene. Later stages of processing will have to correct for these errors. How do we detect edges in an image? Consider the profile of image brightness along a one-dimensional cross-section perpendicular to an edge—for example, the one between the left edge of the desk and the wall. It looks something like what is shown in Figure 24.6(a). The location of the edge corresponds to $x = 50$.

Because edges correspond to locations in images where the brightness undergoes a sharp change, a naive idea would be to differentiate the image and look for places where the magnitude of the derivative $I'(x)$ is large. Well, that almost works. In Figure 24.6(b), we see that, although there is a peak at $x = 50$, there are also subsidiary peaks at other locations (e.g., $x = 75$) that could be mistaken for true edges. These arise because of the presence of noise in the image. If we smooth the image first, the spurious peaks are diminished, as we see in (c).

We have a chance to make an optimization here: we can combine the smoothing and the edge finding into a single operation. It is a theorem that for any functions f and g , the derivative of the convolution, $(f * g)'$, is equal to the convolution with the derivative, $f * (g)'$. So rather than smoothing the image and then differentiating, we can just convolve the image with the derivative of the Gaussian smoothing function, G'_σ . So in one dimension the algorithm for edge finding is:

1. Convolve the image I with G'_σ to obtain R .
2. Mark as edges those peaks in $\|R(x)\|$ that are above some prespecified threshold T .
The threshold is chosen to eliminate spurious peaks due to noise.

In two dimensions edges may be at any angle θ . To detect vertical edges, we have an obvious strategy: convolve with $G'_\sigma(x)G_\sigma(y)$. In the y direction, the effect is just to smooth (because of the Gaussian convolution), and in the x direction, the effect is that of differentiation accompanied with smoothing. The algorithm for detecting vertical edges then is as follows:

1. Convolve the image $I(x, y)$ with $f_V(x, y) = G'_\sigma(x)G_\sigma(y)$ to obtain $R_V(x, y)$.
2. Mark those peaks in $\|R_V(x, y)\|$ that are above some prespecified threshold T .

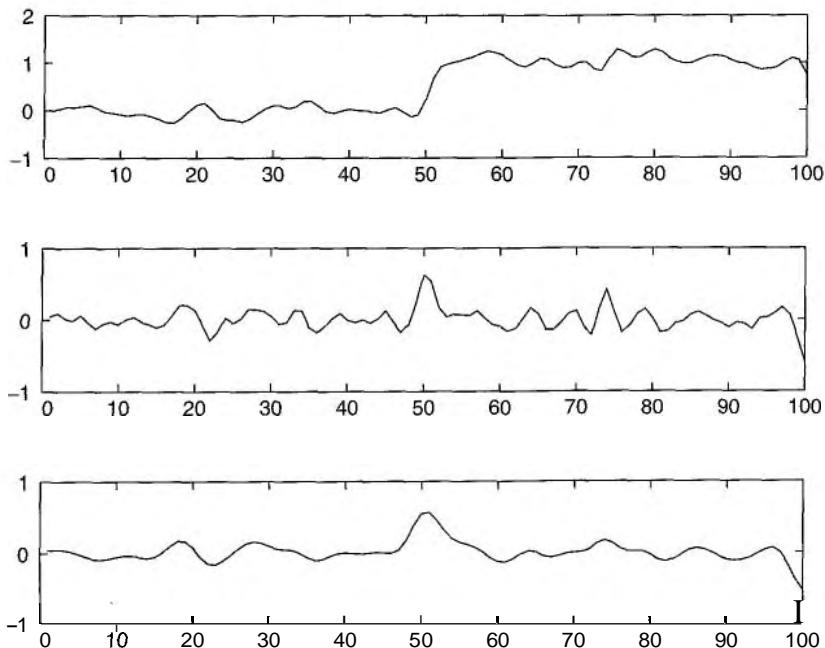


Figure 24.6 Top: Intensity profile $I(x)$ along a one-dimensional section across a step edge. Middle: The derivative of intensity, $I'(x)$. Large values of this function correspond to edges, but the function is noisy. Bottom: The derivative of a smoothed version of the intensity, $(I * G_\sigma)''$, which can be computed in one step as the convolution $\mathbf{I} * G_\sigma''$. The noisy candidate edge at $x = 75$ has disappeared.

In order to detect an edge at an arbitrary orientation, we need to convolve the image with two filters: $f_V = G_\sigma'(x)G_\sigma(y)$ and $f_H = G_\sigma'(y)G_\sigma(x)$, which is just f_V rotated by 90° . The algorithm for detecting edges at arbitrary orientations is then as follows:

1. Convolve the image $\mathbf{I}(x,y)$ with $f_V(x,y)$ and $f_H(x,y)$ to get $R_V(x,y)$ and $R_H(x,y)$, respectively. Define $R(x,y) = R_V^2(x,y) + R_H^2(x,y)$
2. Mark those peaks in $\|R(x,y)\|$ that are above some prespecified threshold T .

Once we have marked edge pixels by this algorithm, the next stage is to link those pixels that belong to the same edge curves. This can be done by assuming that any two neighboring pixels that are both edge pixels with consistent orientations must belong to the same edge curve. This process is called **Canny edge detection** after the inventor, John Canny.

Once detected, edges form the basis for much subsequent processing: we can use them to do stereoptic processing, detect motion, or recognize objects.

Image segmentation

Humans organize their perceptual input; instead of a collection of brightness values associated with individual photoreceptors, we perceive a number of visual groups, usually associated with objects or parts of objects. This ability is equally important for computer vision.

SEGMENTATION

Segmentation is the process of breaking an image into groups, based on similarities of the pixels. The basic idea is the following: Each image pixel can be associated with certain visual properties, such as brightness, color, and texture.² Within an object, or a single part of an object, these attributes vary relatively little, whereas across an inter-object boundary there is typically a large change in one or the other of these attributes. We need to find a partition of the image into sets of pixels such that these constraints are satisfied as well as possible.

There are a number of different ways in which this intuition can be formalized mathematically. For instance, Shi and Malik (2000) set this up as a graph partitioning problem. The nodes of the graph correspond to pixels, and edges to connections between pixels. The weight W_{ij} on the edge connecting a pair of pixels i and j is based on how similar the two pixels are in brightness, color, texture etc. They then find partitions that minimize a normalized cut criterion. Roughly speaking, the criterion for partitioning the graph is to minimize the sum of weights of connections across the groups and maximize the sum of weights of connections within the groups.

Segmentation based purely on low-level, local attributes, such as brightness and color is an error-prone process. To reliably find boundaries associated with objects, one should also incorporate high-level knowledge of the kinds of objects one may expect to encounter in a scene. The Hidden Markov model formalism makes this possible for speech recognition; in the context of images such a unified framework remains a topic of active research. In any case, high-level knowledge of objects is the subject of the next section.

24.4 EXTRACTING THREE-DIMENSIONAL INFORMATION

In this section we show how to go from the two-dimensional image to a three-dimensional representation of the scene. It is important to reason about the scene, because, after all, the agent lives in the world, not in the image plane, and the goal of vision is to be able to interact with objects in the world. However, most agents need only a limited abstract representation of certain aspects of the scene, not of every detail. The algorithms we have seen in the rest of the book for dealing with the world depend on having concise descriptions of objects, not exhaustive enumerations of every three-dimensional surface patch.

OBJECT
RECOGNITION

First we will cover **object recognition**, the process of converting features of the image (such as edges) into model of known objects (such as staplers). Object recognition consists of three steps: Segmenting the scene into distinct objects, determining the position and orientation of each object relative to the observer, and determining the shape of each object.

POSE

Discovering the position and orientation of an object relative to the observer (the so-called **pose** of the object) is most important for manipulation and navigation tasks. To move around in a crowded factory floor, one needs to know the locations of the obstacles, so that one can plan a path that avoids them. If one wants to pick up and grasp an object, one needs to know its position relative to the hand, so that an appropriate trajectory of moves can be generated. Manipulation and navigation actions typically are done in a control loop setting:

² Texture properties are based on statistics measured in a small patch centered at the pixel.

the sensory information provides feedback to modify the motion of the robot, or the motion of the robot's arm.

Let us specify position and orientation in mathematical terms. The position of a point \mathbf{P} in the scene is characterized by three numbers, the (X, Y, Z) coordinates of \mathbf{P} in a coordinate frame with its origin at the pinhole and the Z-axis along the optical axis (Figure 24.1). What we have available is the perspective projection of the point in the image (x, y) . This specifies the ray from the pinhole along which \mathbf{P} lies; what we do not know is the distance. The term "orientation" could be used in two senses:

1. **The orientation of the object as a whole.** This can be specified in terms of a three-dimensional rotation relating its coordinate frame to that of the camera.
2. **The orientation of the surface of the object at \mathbf{P} .** This can be specified by a normal vector, \mathbf{n} —that is a vector specifying the direction that is perpendicular to the surface. Often we express the surface orientation using the variables **slant** and **tilt**. Slant is the angle between the Z-axis and \mathbf{n} . Tilt is the angle between the X-axis and the projection of \mathbf{n} on the image plane.

SLANT
TILT

SHAPE

When the camera moves relative to an object, both the object's distance and its orientation change. What is preserved is the **shape** of the object. If the object is a cube, that fact is not changed when the object moves. Geometers have been attempting to formalize shape for centuries, the basic concept being that shape is what remains unchanged under some group of transformations, for example, combinations of rotations and translations. The difficulty lies in finding a representation of global shape that is general enough to deal with the wide variety of objects in the real world—not just simple forms like cylinders, cones, and spheres—and yet can be recovered easily from the visual input. The problem of characterizing the local shape of a surface is much better understood. Essentially, this can be done in terms of curvature: how does the surface normal change as one moves in different directions on the surface. For a plane, there is no change at all. For a cylinder, if one moves parallel to the axis, there is no change, but in the perpendicular direction, the surface normal rotates at a rate inversely proportional to the radius of the cylinder, and so on. All this is studied in the subject called differential geometry.

The shape of an object is relevant for some manipulation tasks (e.g., deciding where to grasp an object), but its most significant role is in object recognition, where geometric shape along with color and texture provide the most significant cues to enable us to identify objects, classify what is in the image as an example of some class one has seen before, and so on.

The fundamental question is the following: Given the fact that, during perspective projection, all points in the three-dimensional world along a ray from the pinhole have been projected to the same point in the image, how do we recover three-dimensional information? There are a number of cues available in the visual stimulus for this, including **motion**, **binocular stereopsis**, **texture**, **shading**, and **contour**. Each of these cues relies on background assumptions about physical scenes in order to provide (nearly) unambiguous interpretations. We discuss each of these cues in the five subsections that follow.

Motion

OPTICAL FLOW

So far we have considered only a single image at a time.. But video cameras capture 30 frames per second, and the differences between frames can be an important source of information. If the camera moves relative to the three-dimensional scene, the resulting apparent motion in the image is called **optical flow**. This describes the direction and speed of motion of features *in the* image as a result of relative motion between the viewer and the scene. In Figure 24.7(a) and (b), we show two frames from a video of a rotating Rubik's cube. In (c) we display the optical flow vectors computed from these images. The optical flow encodes useful information about scene structure. For example, when viewed from a moving car, distant objects have much slower apparent motion than close objects; thus, the rate of apparent motion can tell us something about distance.

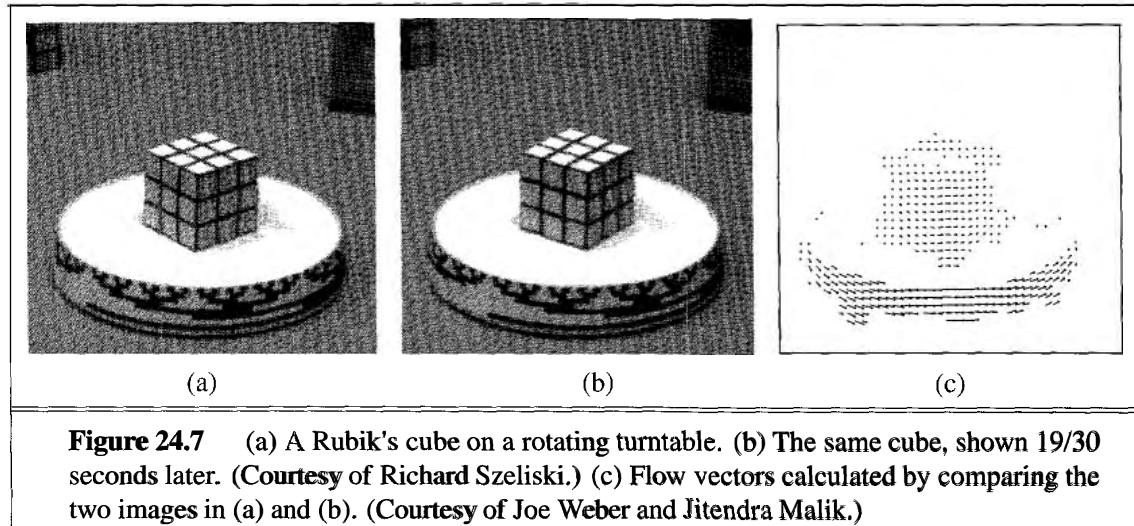


Figure 24.7 (a) A Rubik's cube on a rotating turntable. (b) The same cube, shown 19/30 seconds later. (Courtesy of Richard Szeliski.) (c) Flow vectors calculated by comparing the two images in (a) and (b). (Courtesy of Joe Weber and Jitendra Malik.)

SUM OF SQUARED DIFFERENCES

The optical flow vector field can be represented by its components $v_x(x, y)$ in the x direction and $v_y(x, y)$ in the y direction. To measure optical flow, we need to find corresponding points between one time frame and the next. We exploit the fact that image patches around corresponding points have similar intensity patterns. Consider a block of pixels centered at pixel p , (x_0, y_0) at time t_0 . This block of pixels is to be compared with pixel blocks centered at various candidate pixels q_i at $(x_0 + D_x, y_0 + D_y)$ at time $t_0 + D_t$. One possible measure of similarity is the **sum of squared differences** (SSD):

$$\text{SSD}(D_x, D_y) = \sum_{(x,y)} (I(x, y, t) - I(x + D_x, y + D_y, t + D_t))^2 .$$

CROSS-CORRELATION

Here, (x, y) ranges over pixels in the block centered at (x_0, y_0) . We find the (D_x, D_y) that minimizes the SSD. The optical flow at (x_0, y_0) is then $(v_x, v_y) = (D_x/D_t, D_y/D_t)$. Alternatively, one can maximize the **cross-correlation**:

$$\text{Correlation}(D_x, D_y) = \sum_{(x,y)} I(x, y, t)I(x + D_x, y + D_y, t + D_t) .$$

Cross-correlation works best when there is texture in the scene, resulting in windows containing a significant variation in brightness among the pixels. If one is looking at a uniform white wall, then the cross-correlation is going to be nearly the same for the different candidate matches q , and the algorithm is reduced to making a blind guess.

Suppose that the viewer has translational velocity T and angular velocity ω (which thus describe the **egomotion**). One can derive an equation relating the viewer's velocities, the optical flow, and the positions of objects in the scene. Assuming that $f = 1$, it follows that

$$\begin{aligned} v_x(x, y) &= \left[-\frac{T_x}{Z(x, y)} - \omega_y + \omega_z y \right] - x \left[-\frac{T_z}{Z(x, y)} - \omega_x y + \omega_y x \right] \\ v_y(x, y) &= \left[-\frac{T_y}{Z(x, y)} - \omega_z x + \omega_x \right] - y \left[-\frac{T_z}{Z(x, y)} - \omega_x y + \omega_y x \right], \end{aligned}$$

where $Z(x, y)$ gives the z -coordinate of the point in the scene corresponding to the point in the image at (x, y) .

One can get a good intuition by considering the case of pure translation. In that case, the flow field becomes

$$v_x(x, y) = \frac{-T_x + xT_z}{Z(x, y)}, \quad v_y(x, y) = \frac{-T_y + yT_z}{Z(x, y)}.$$

Now some interesting properties come to light. Both components of the optical flow, $v_x(x, y)$ and $v_y(x, y)$, are zero at the point $x = T_x/T_z, y = T_y/T_z$. This point is called the **focus of expansion** of the flow field. Suppose we change the origin in the x - y plane to lie at the focus of expansion; then the expressions for optical flow take on a particularly simple form. Let (x', y') be the new coordinates defined by $x' = x - T_x/T_z, y' = y - T_y/T_z$. Then

$$v_x(x', y') = \frac{x'T_z}{Z(x', y')}, \quad v_y(x', y') = \frac{y'T_z}{Z(x', y')}.$$

This equation has some interesting applications. Suppose you are a fly trying to land on a wall and you want to know the time to contact at the current velocity. This time is given by Z/T_z . Note that although the instantaneous optical flow field cannot provide either the distance Z or the velocity component T_z , it can provide the ratio of the two and can therefore be used to control the landing approach. Experiments with real flies show that this is exactly what they use. Flies are the most dexterous fliers of any animal or machine, and it is interesting that they do it with a vision system that has terrible spatial resolution (having only about 600 receptors compared to a human's 100 million) but spectacular temporal resolution.

To recover depth, one should make use of multiple frames. If the camera is looking at a rigid body, the shape does not change from frame to frame, and thus we are able to better deal with the inherently noisy optical flow measurements. Results from one such approach due to Tomasi and Kanade (1992) are shown in Figures 24.8 and 24.9.

Binocular stereopsis

Most vertebrates have *two* eyes. This is useful for redundancy in case of a lost eye, but it helps in other ways too. Most prey have eyes on the side of the head to enable a wider field of vision. Predators have the eyes in the front, enabling them to use **binocular stereopsis**.

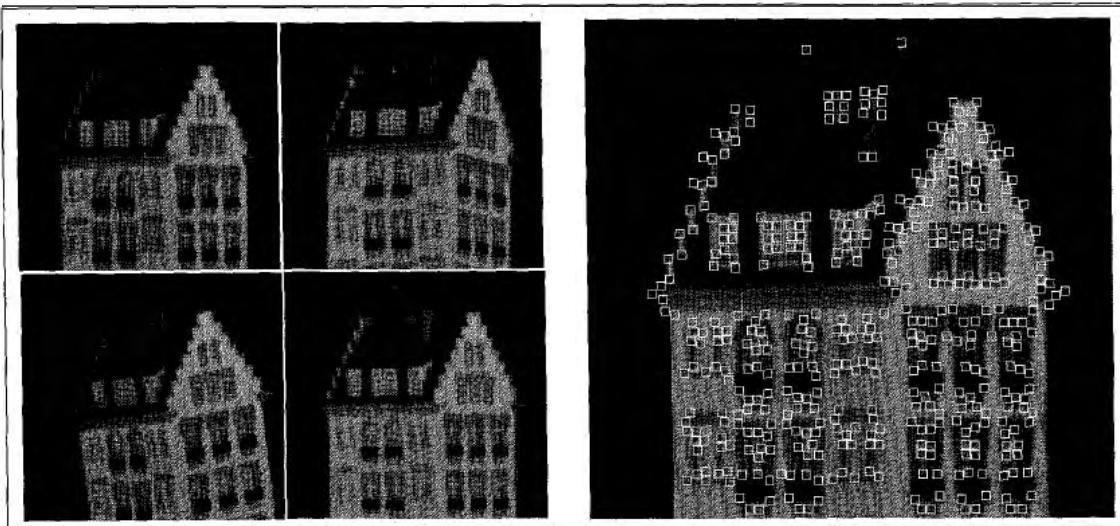


Figure 24.8 (a) Four frames from a video sequence in which the camera is moved and rotated relative to the object. (b) The first frame of the sequence, annotated with small boxes highlighting the features found by the feature detector. (Courtesy of Carlo Tomasi.)

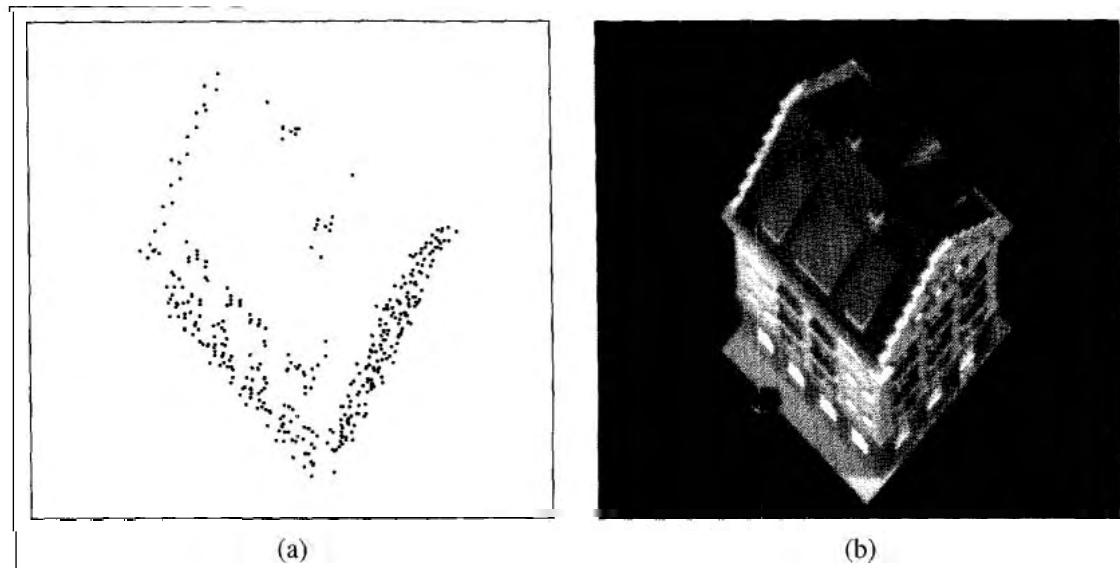


Figure 24.9 (a) three-dimensional reconstruction of the locations of the image features in Figure 24.8, shown from above. (b) The real house, taken from the same position.

DISPARITY

The idea is similar to motion parallax, except that instead of using images over time, we use two (or more) images separated in space, such as are provided by the forward-facing eyes of humans. Because a given feature in the scene will be in a different place relative to the z-axis of each image plane, if we superpose the two images, there will be a **disparity** in the location of the image feature in the two images. You can see this in Figure 24.10, where the nearest point of the pyramid is shifted to the left in the right image and to the right in the left image.

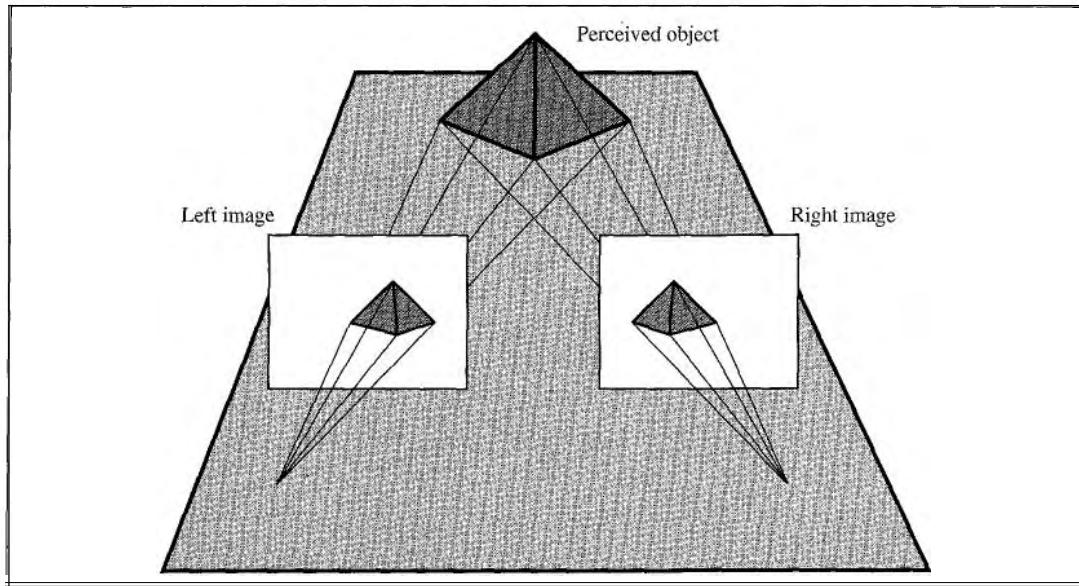


Figure 24.10 The idea of stereopsis: different camera positions result in slightly different two-dimensional views of the same three-dimensional scene.

Let us work out the geometrical relationship between disparity and depth. First, we will consider the case when both the eyes (or cameras) are looking forward with their optical axes parallel. The relationship of the right camera to the left camera is then just translation along the x-axis by an amount b , the baseline. We can use the optical flow equations from the previous section to compute the horizontal and vertical disparity as $H = v$, $A\dot{t}, V = v_y A\dot{t}$, given that $T_x = b/\Delta t$ and $T_y = T_z = 0$. The rotational parameters w_x , w_y , and ω_z are zero. One obtains $H = b/Z$, $V = 0$. In words, the horizontal disparity is equal to the ratio of the baseline to the depth, and the vertical disparity is zero.

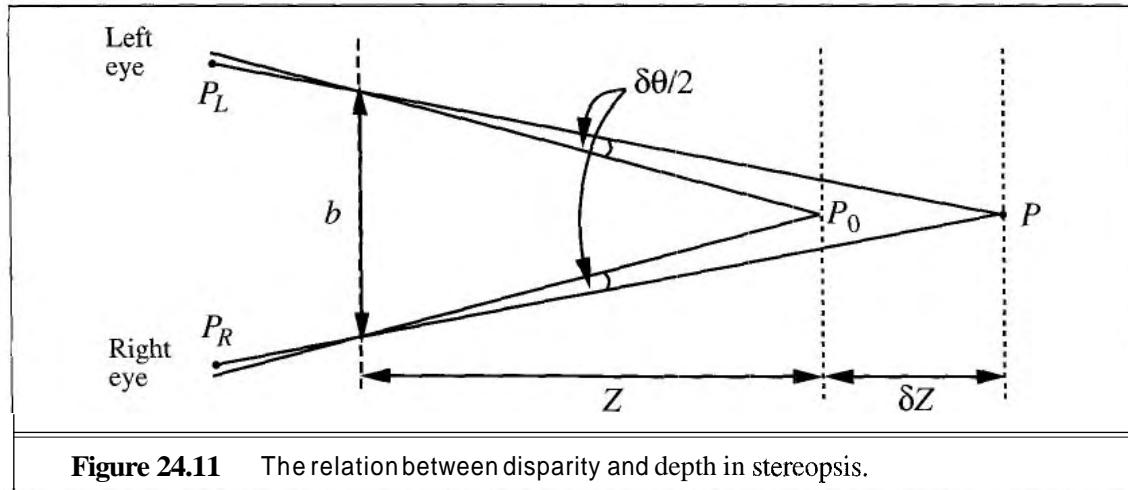
FIXATE

Under normal viewing conditions, humans fixate; that is, there is some point in the scene at which the optical axes of the two eyes intersect. Figure 24.11 shows two eyes fixated at a point P_0 , which is at a distance Z from the midpoint of the eyes. For convenience, we will compute the angular disparity, measured in radians. The disparity at the point of fixation P_0 is zero. For some other point P in the scene that is δZ further away, we can compute the angular displacements of the left and right images of P , which we will call P_L and P_R , respectively. If each of these is displaced by an angle $\delta\theta/2$ relative to P_0 , then the displacement between P_L and P_R , which is the disparity of P , is just $\delta\theta$. From simple geometry, we have

$$\frac{68}{\delta Z} = \frac{-b}{Z^2}.$$

BASELINE

In humans, b (the baseline) is about 6 cm. Suppose that Z is about 100 cm. Then the smallest detectable 68 (corresponding to the pixel size) is about 5 seconds of arc, giving a δZ of 0.4 mm. For $Z = 30$ cm, we get the impressively small value $\delta Z = 0.036$ mm. That is, at a distance of 30 cm, humans can discriminate depths that differ by as little as 0.036 mm, enabling us to thread needles and the like.



Texture gradients

TEXTURE

Texture, in everyday language, is a property of surfaces associated with the tactile quality they suggest ("texture" has the same root as "textile"). In computational vision, it refers to a closely related concept, that of a spatially repeating pattern on a surface that can be sensed visually. Examples include the pattern of windows on a building, the stitches on a sweater, the spots on a leopard's skin, blades of grass on a lawn, pebbles on a beach and a crowd of people in a stadium. Sometimes the arrangement is quite periodic, as in the stitches on a sweater; in other instances, such as pebbles on a beach, the regularity is only in a statistical sense: the density of pebbles is roughly the same on different parts of the beach.

TEXELS

What we just said is true *in the scene*. In the image, the apparent size, shape, spacing, and so on of the texture elements (the **texels**) do indeed vary, as illustrated in Figure 24.12. The tiles are identical in the scene. There are two main causes for the variation in the projected size and shape of the tiles in the image:

1. *Differences in the distances of the texels from the camera.* Recall that under perspective projection, distant objects appear smaller. The scaling factor is $1/Z$.
2. *Differences in the foreshortening of the texels.* This is related to the orientation of each texel relative to the line of sight from the camera. If the texel is perpendicular to the line of sight, there is no foreshortening. The magnitude of the foreshortening effect is proportional to $\cos \alpha$, where α is the slant of the plane of the texel.

TEXTURE GRADIENTS

Through some mathematical analysis, one can compute expressions for the rate of change of various image texel features, such as area, foreshortening, and density. These **texture gradients** are functions of the surface shape, as well as its slant and tilt with respect to the viewer's location.

To recover shape from texture, one can use a two-step process: (a) measure the texture gradients; (b) estimate the surface shape, slant, and tilt that would give rise to the measured texture gradients. We show the results of this process in Figure 24.12.

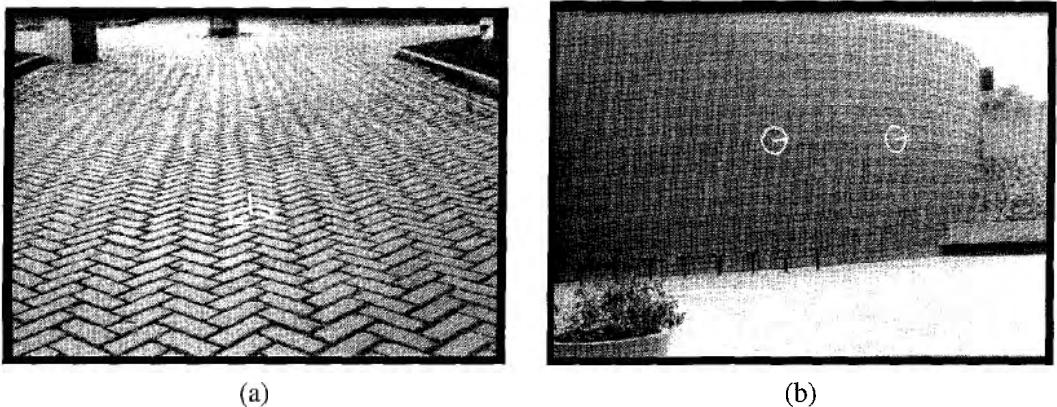


Figure 24.12 (a) A scene illustrating texture gradient. Assuming that the real texture is uniform allows recovery of the surface orientation. The computed surface orientation is indicated by overlaying a white circle and pointer, transformed as if the circle were painted on the surface at that point. (b) Recovery of shape from texture for a curved surface. (Images courtesy of Jitendra Malik and Ruth Rosenholtz (1994).)

Shading

Shading—variation in the intensity of light received from different portions of a surface in a scene—is determined by the geometry of the scene and by the reflectance properties of the surfaces. In computer graphics, the objective is to compute the image brightness $I(x, y)$, given the scene geometry and reflectance properties of the objects in the scene. Computer vision aims to invert the process—that is, to recover the geometry and reflectance properties, given the image brightness $I(x, y)$. This has proved to be difficult to do in anything but the simplest cases.

Let us start with a situation in which we can, in fact, solve for shape from shading. Consider a Lambertian surface illuminated by a distant point light source. We will assume that the surface is distant enough from the camera so that we can use orthographic projection as an approximation to perspective projection. The image brightness is

$$I(x, y) = k\mathbf{n}(x, y) \cdot \mathbf{s},$$

where k is a scaling constant, \mathbf{n} is the unit vector normal to the surface, and \mathbf{s} is the unit vector in the direction of the light source. Because \mathbf{n} and \mathbf{s} are unit vectors, their dot product is just the cosine of the angle between them. The shape of the surface is captured in the variation of the normal vector \mathbf{n} along the surface. Let us assume that k and \mathbf{s} are known. Our problem then is to recover the surface normal vector $\mathbf{n}(x, y)$ given the image intensity $I(x, y)$.

The first observation to make is that the problem of determining \mathbf{n} , given the brightness \mathbf{I} at a given pixel (x, y) , is underdetermined locally. We can compute the angle that \mathbf{n} makes with the light source vector, but that only constrains it to lie on a certain cone of directions with axis \mathbf{s} and apex angle $\theta = \cos^{-1}(I/k)$. To proceed further, note that \mathbf{n} cannot vary arbitrarily from pixel to pixel. It corresponds to the normal vector of a smooth surface patch and consequently must also vary in a smooth fashion—the technical term for the constraint is

INTEGRABILITY

integrability. Several different techniques have been developed to exploit this insight. One is simply to rewrite \mathbf{n} in terms of the partial derivatives; Z_x and Z_y of the depth $Z(x, y)$. This results in a partial differential equation for Z that can be solved to yield the depth $Z(x, y)$, given appropriate boundary conditions.

REFLECTANCEMAP

One can generalize the approach somewhat. It is not necessary for the surface to be Lambertian nor for the light source to be a point source. As long as one is able to compute the **reflectance map** $R(\mathbf{n})$, which specifies the brightness of a surface patch as a function of its surface normal \mathbf{n} , essentially the same kind of techniques can be used.

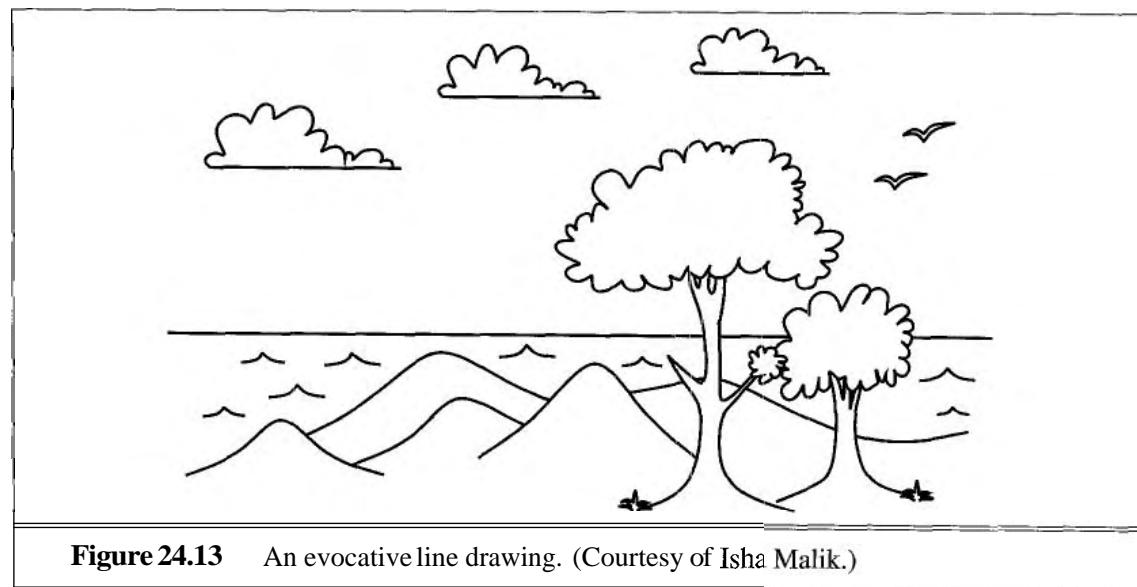
The real difficulty comes in dealing with interreflections. If we consider a typical indoor scene, such as the objects inside an office, surfaces are illuminated not only by the light sources, but also by the light reflected from other surfaces in the scene that effectively serve as secondary light sources. These mutual illumination effects are quite significant. The reflectance map formalism completely fails in this situation: image brightness depends not just on the surface normal, but also on the complex spatial relationships among the different surfaces in the scene.

Humans clearly do get some perception of shape from shading, so this remains an interesting problem in spite of all these difficulties.

Contour

When we look at a line drawing, such as Figure 24.13, we get a vivid perception of three-dimensional shape and layout. How? After all, we saw earlier that there is an infinity of scene configurations that can give rise to the same line drawing. Note that we get even a perception of surface slant and tilt. It could be due to a combination of high-level knowledge (about typical shapes) with low-level constraints.

We will consider the qualitative knowledge available from a line drawing. As discussed earlier, lines in a drawing can have multiple significance. (See Figure 24.4 and the accompa-



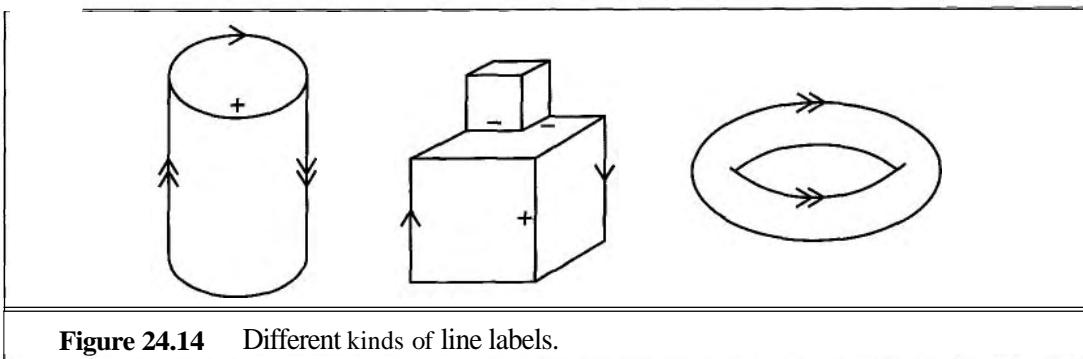
nying text.) The task of evaluating the actual significance of each line in an image is called **line labeling** and was one of the first tasks studied in computer vision. For now, let us deal with a simplified model of the world wherein the objects have no surface marks and the lines due to illumination discontinuities, such as shadow edges and specularities, have been removed in some preprocessing step, enabling us to limit our attention to line drawings where each line corresponds to either a depth or an orientation discontinuity.

Each line then can be classified either as the projection of a **limb** (the locus of points on the surface where the line of sight is tangent to the surface) or as an **edge** (a surface normal discontinuity). In addition, each edge can be classified as convex, concave, or occluding. For occluding edges and limbs, we would like to figure out which of the two surfaces bordering the curve in the line drawing is nearer in the scene. These inferences can be represented by giving each line one of six possible **line labels** as illustrated in Figure 24.14:

1. “+” and “−” labels represent convex and concave edges, respectively. These are associated with surface normal discontinuities wherein both surfaces that meet along the edge are visible.
2. A “←” or a “→” represents an occluding convex edge. When viewed from the camera, both surface patches that meet along the edge lie on the same side, one occluding the other. As one moves in the direction of the arrow, the surfaces are to the right.
3. A “←←” or a “→→” represents a limb. Here, the surface curves smoothly around to occlude itself. As one moves in the direction of the twin arrows, the surface lies to the right. The line of sight is tangential to the surface for all points on the limb. Limbs move on the surface of the object as the viewpoint changes.

Of the 6^n combinatorially possible label assignments to the n lines in a drawing, only a small number are physically possible. The determination of these label assignments is the line labeling problem. Note that the problem makes sense only if the label is the same all the way along a line. This is not always true, because the label can change along a line for images of curved objects. We will deal solely with polyhedral objects, to avoid this concern.

Huffman (1971) and Clowes (1971) independently attempted the first systematic approach to polyhedral scene analysis. Huffman and Clowes limited their analysis to scenes with opaque **trihedral** solids—objects in which exactly three plane surfaces come together at each vertex. For scenes with multiple objects, they also ruled out object alignments that



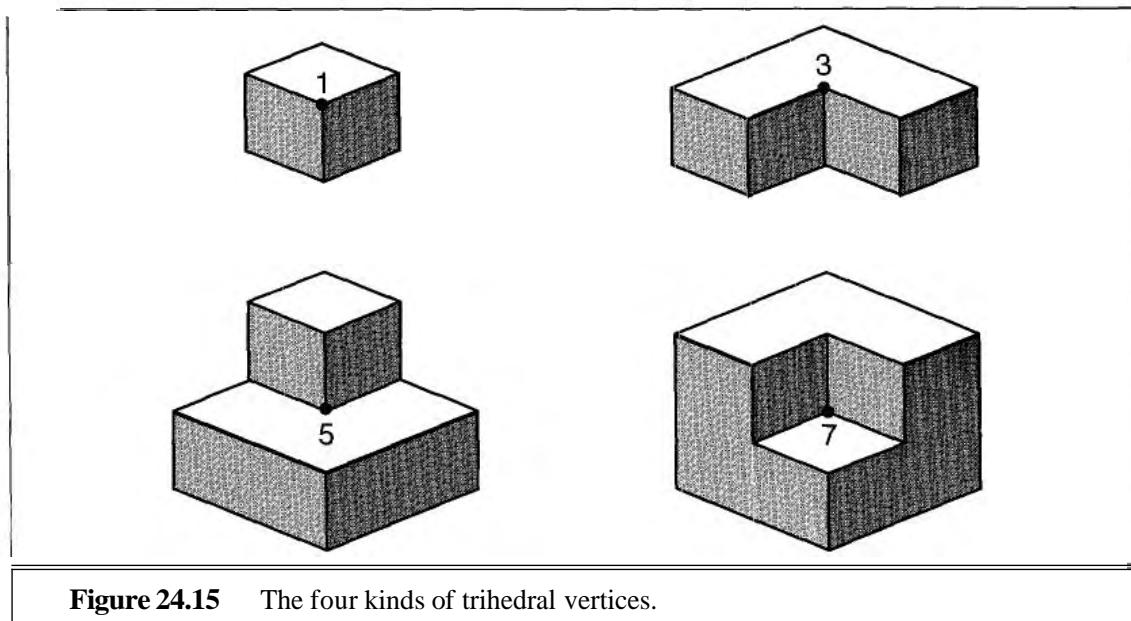


Figure 24.15 The four kinds of trihedral vertices.

CRACKS

would result in a violation of the trihedral assumption, such as two cubes sharing a common edge. **Cracks** (i.e., "edges" across which the tangent planes are continuous) were also not permitted. For the trihedral world, Huffman and Clowes made an exhaustive listing of all the different types of vertices and the different ways in which they could be viewed under general viewpoint. The general viewpoint condition essentially ensures that if there is a small movement of the eye, none of the junctions changes character. For example, this condition implies that if three lines intersect in the image, the corresponding edges in the scene must also intersect.

OCTANTS

The four ways in which three plane surfaces can come together at a vertex are shown in Figure 24.15. These cases have been constructed by taking a cube and dividing it into eight **octants**. We want to generate the different possible trihedral vertices at the center of the cube by filling in various octants. The vertex labeled 1 corresponds to one filled octant, 3 to three filled octants, and so on. Readers should convince themselves that these are indeed all the possibilities. For example, if one fills two octants in a cube, one cannot construct a valid trihedral vertex at the center. Note also that these four cases correspond to different combinations of convex and concave edges that meet at the vertex.

The three edges meeting at the vertex partition the surrounding space into eight octants. A vertex can be viewed from any of the octants not occupied by solid material. Moving the viewpoint within a single octant does not result in a picture with different types of junctions. The vertex labeled 1 in Figure 24.15 can be viewed from any of the remaining seven octants to give the junction labels in Figure 24.16.

An exhaustive listing of the different ways each vertex can be viewed results in the possibilities shown in Figure 24.17. We get four different junction types that can be distinguished in the image: L-, Y-, arrow, and T-junctions. L-junctions correspond to two visible edges. Y- and arrow junctions correspond to a triple of edges—in a Y-junction, none of the three an-

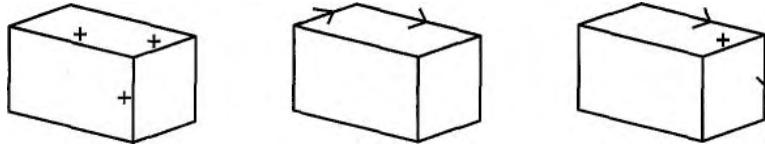


Figure 24.16 The different appearances of the vertex labeled 1 in Figure 24.15.

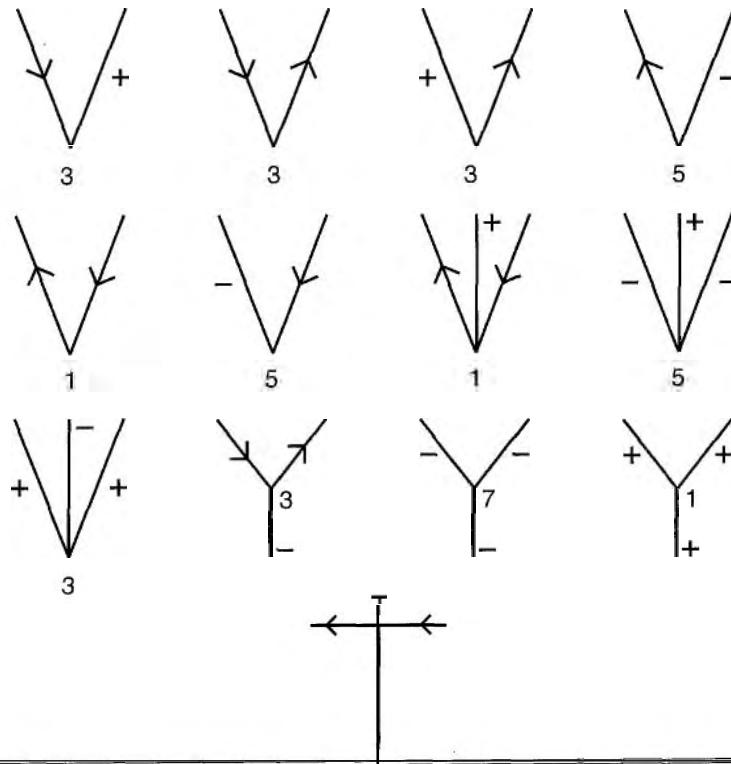


Figure 24.17 The Huffman–Clowes label set.

gles is greater than 180° . T-junctions are associated with occlusion. When a nearer, opaque surface blocks the view of a more distant edge, one obtains a continuous edge meeting a half edge. The four T-junction labels correspond to the occlusion of four different types of edges.

In using this junction dictionary to find a labeling for the line drawing, the problem is to discover which junction interpretations are globally consistent. Consistency is forced by the rule that each line in the picture must be assigned one and only one label along its entire length. Waltz (1975) proposed an algorithm for this problem (actually for an augmented version with shadows, cracks, and separably concave edges) that was one of the first applications of constraint satisfaction in AI (see Chapter 5). In the terminology of CSPs, the variables are the junctions, the values are labelings for the junctions, and the constraints are that each line has a single label. Although the line-labeling problem for trihedral scenes is NP-complete, standard CSP algorithms perform well in practice.

24.5 OBJECT RECOGNITION

Vision enables us to recognize people, animals, and inanimate objects reliably. In AI or computer vision, it is customary to use the term *object recognition* to refer to all of these abilities. This includes determining the class of particular objects that have been imaged—e.g., a face—as well as recognizing specific objects—e.g., Bill Clinton's face. Motivating applications include the following:

BIOMETRIC IDENTIFICATION

◊ **Biometric** identification: Criminal investigations and access control for restricted facilities require the ability to identify unique individuals. Fingerprints, iris scans, and facial photographs result in images that must be matched to specific individuals.

CONTENT-BASED IMAGE RETRIEVAL

◊ Content-based image retrieval: It is easy to find a location in a document, if one exists, for the string "cat"—any text editor provides this capability. Now consider the problem of finding the subset of pixels in an image which correspond to the image of a cat. If one had this capability, one could answer image queries such as "Bill Clinton and Nelson Mandela together," "a skater in mid-air," "the Eiffel Tower at night," and so on, without having had to type in caption keywords for each photograph in a collection. As image and video collections grow, manual annotation cannot scale.

HANDWRITING RECOGNITION

◊ Handwriting recognition: Examples include signatures, address blocks on envelopes, amounts on checks, and pen-based input on PDAs.

Vision is used to recognize not only objects, but also activities. We can identify gaits (a friend's walk), expressions (a smile, a grimace), gestures (a person waving), actions (jumping, dancing) and so on. Research on activity recognition is still in its infancy, so in this section we will concentrate on object recognition.

The problem of visual object recognition is generally easy for people, but has proved to be very difficult for computers. One wants to be able to identify a person's face in spite of variations in illumination, pose with respect to the camera, and facial expression. Any of these changes causes widespread differences in pixel brightness values, so a straightforward comparison of pixels is unlikely to work. When one wants to recognize examples of a category such as "car", one must cope also with the within-category variation. Even the very restricted problem of recognition of handwritten digits in postal zip codes proved to be quite a challenge.

Supervised learning or pattern classification provides a natural framework for studying object recognition. Given images of positive examples ("faces") and negative examples ("nonfaces"), the objective is to learn a function that can map novel images to one of the labels *face*, *nonface*. All of the techniques from Chapters 18 and 20 are plausible candidates: multilayer perceptrons, decision trees, nearest-neighbor classifiers, and kernel machines have all been applied to object recognition problems. We should note, however, that the application of these techniques to object recognition is far from straightforward.

The first challenge is image segmentation. Any image will typically contain multiple objects, so we need first to partition it into subsets of pixels that correspond to single objects. Once the image has been partitioned into regions, one can then input these regions or

assemblies of regions into a classifier to determine object labels. Unfortunately, bottom-up segmentation is an error-prone process, so alternatively one might seek to find object groups top-down. That is, search for a subset of pixels that you can classify as a face, and if you succeed, you have found a group! Purely top-down approaches have high computational complexity, because one needs to examine image windows of different sizes, and at different locations, as well as compare them to all the different object hypotheses. At present, most practical object recognition systems use such a top-down strategy, though this might change as bottom-up techniques improve.

The second challenge is to ensure that the recognition process is robust against variations in illumination and pose. Humans can recognize objects in spite of considerable variation in precise appearance as measured by pixel brightness values. For example, we can recognize a friend's face under different illumination conditions, or at different angles of view. As an even simpler example, consider recognizing the handwritten digit 6. One should be able to do this at different sizes and at different positions in the image, and in spite of small rotations of the figure.³

The key point to note here is that geometrical transformations such as translation, scaling and rotation, or transformations of image brightness caused by moving light sources physically, have a different character than the intra-category variation such as exists between different human faces. Obviously, learning is the only way to learn about the different kinds of human faces, or the different ways of writing the digit 4. On the other hand, the effects of geometric and physical transformations are systematic and one should be able to factor them out by a proper design of the features used to represent the training instances.

To provide invariance under geometrical transformations, one technique that has proved quite effective is to preprocess the image region into a standard position, scale, and orientation. Alternatively, we can merrily ignore the causal nature of the geometrical and physical transformations and think of them as just other sources of variability for the classifier. In the training set, examples need to be provided corresponding to all these variations, and the hope is that the classifier will induce an appropriate set of transformations of the input so that the variations are factored out.

Let us now turn to specific algorithms for object recognition. For simplicity, we focus on the problem in a two-dimensional setting, with both training and test examples given in the form of two-dimensional brightness images. In domains such as handwriting recognition, this is clearly sufficient. Even in the case of three-dimensional objects, an effective strategy is to represent them by multiple two-dimensional views (see Figure 24.18) and classify new objects by comparing them to (some representation of) the stored views.

The previous section showed there are multiple cues for extracting three-dimensional information about a scene. Object recognition is also based on multiple cues—we identify a tiger by its mixture of orange and black colors, by its striped texture, and by its body shape.

Color and texture can be represented using histograms or empirical frequency distributions. Given an example image of a tiger, we can measure the percentage of pixels in the different color bins. Then, when an unknown example is presented, we can compare its color

³ Complete rotation invariance is neither necessary nor desirable—one might then confuse a 6 with a 9!

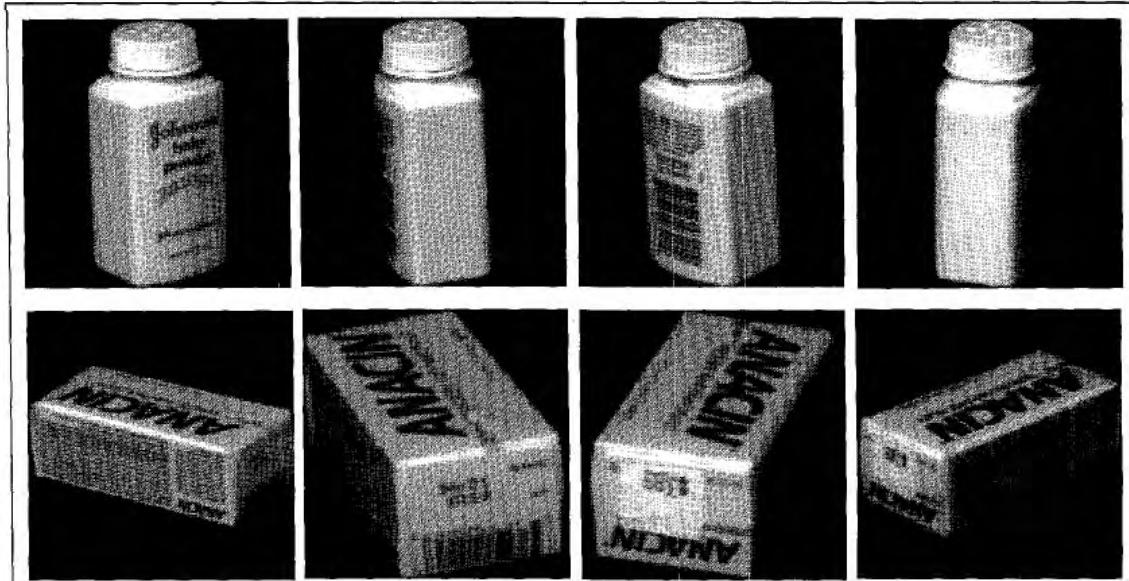


Figure 24.18 Multiple views of two three-dimensional objects.

histogram with that of previously seen tiger examples. To analyze textures, we consider histograms of the responses of an image to convolution with filters of various orientations and scales, searching for a match.

The use of shape for object recognition has proved to be much more difficult. Broadly speaking, there are two main approaches: **brightness-based recognition**, in which pixel brightness values are used directly, and **feature-based recognition**, which involves the use of spatial arrangements of extracted features such as edges or key points. After discussing each of these two approaches in more detail, we will also address the problem of **pose estimation**, i.e., determining the location and orientation of objects in the scene.

Brightness-based recognition

Given the subset of image pixels that corresponds to a candidate object, define the features to be the raw pixel brightness values themselves. Or, in a variant, one might first convolve the image with various linear filters and treat the pixel values in the resulting images as the features. This approach has been very successful at tasks such as handwritten digit recognition, as we saw in Section 20.7.

A variety of statistical methods have been used to develop face detectors from image databases, including neural networks with raw pixel inputs, decision trees with features defined by various bar and edge filters, and naive Bayes models with wavelet features. Some results from the latter approach are shown in Figure 24.19.

One negative aspect of using raw pixels as feature vectors is the great redundancy inherent in this representation. Consider two nearby pixels on the cheek of a face; they are likely to be very highly correlated because of similar geometry, illumination, etc. Data reduction techniques, such as principal component analysis, can be used successfully to reduce the di-

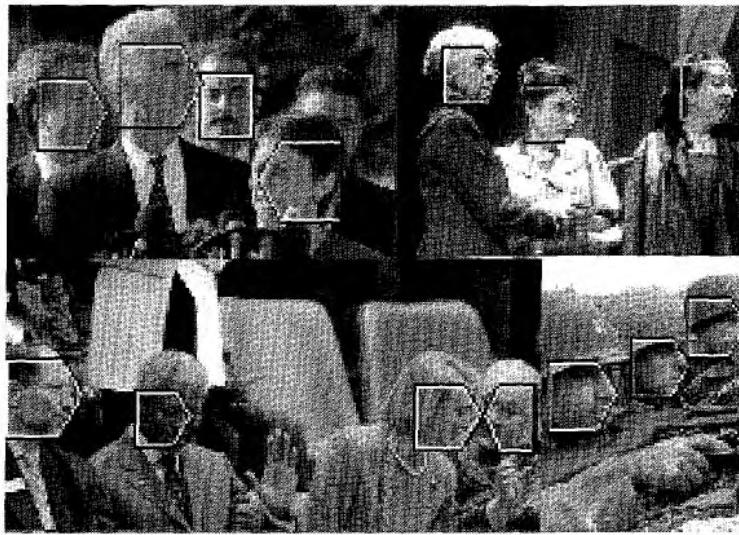


Figure 24.19 Output of a face-finding algorithm. (Courtesy of Henry Schneiderman and Takeo Kanade.)

mensionality of the feature vector, enabling recognition of such things as faces with greater speed than one would get in a higher-dimensional space.

Feature-based recognition

Instead of using raw pixel brightnesses as features, we can detect and mark spatially localized features such as regions and edges (Section 24.3). There are two motivations for using edges. One is data reduction—there are far fewer edges than image pixels. The other is illumination invariance—within a suitable range of contrasts, the edges will be detected at roughly the same locations, independent of precise lighting configuration. Edges are one-dimensional features; two-dimensional features (regions) and zero-dimensional features (points) have also been used. Note the difference in the treatment of spatial location in brightness-based and feature-based approaches. In brightness based approaches, this is coded implicitly as the index to a component of a feature vector. In feature-based approaches, the (x,y) location is the feature.

The arrangement of edges is characteristic of an object—this is one reason why we can interpret line drawings (Figure 24.13) easily, even though such images do not occur in nature! The easiest way to use this knowledge is with a nearest-neighbor classifier. We pre-compute and store the configurations of edges corresponding to views of known objects. Given the configuration of edges corresponding to the unknown object in the query image, we can determine the "distance" to each member of a library of stored views. A nearest-neighbor classifier chooses the closest match.

Many different definitions have been proposed for distances between images. One of the more interesting approaches is based on the idea of **deformable matching**. In his classic work *On Growth and Form*, D'Arcy Thompson (1917) observed that related but not identical

shapes can often be deformed into alignment using simple coordinate transformations.⁴ In this paradigm, we operationalize a notion of shape similarity as a three stage process: (1) solve the correspondence problem between the two shapes, (2) use the correspondences to estimate an aligning transform, and (3) compute the 'distance' between the two shapes as a sum of matching errors between corresponding points, together with a term measuring the magnitude of the aligning transformation.

We represent a shape by a discrete set of points sampled from the internal or external contours on the shape. These can be obtained as locations of edge pixels as found by an edge detector, giving us a set $\{p_1, \dots, p_N\}$ of N points. Figure 24.20(a) and (b) show sample points for two shapes.

Now consider a particular sample point p_i , together with the set of vectors originating from that point to all other sample points on a shape. These vectors express the configuration of the entire shape relative to the reference point. This leads to the following idea: associate with each sample point a descriptor, the **shape context**, which describes the coarse arrangement of the rest of the shape with respect to the point. More precisely, the shape context of p_i is a coarse spatial histogram h_i of the relative coordinates $p_k - p_i$ of the remaining $N-1$ points p_k . A log-polar coordinate system is used for defining the bins ensuring that the descriptor is more sensitive to differences in nearby pixels. An example is shown in Figure 24.20(c).

Note that invariance to translation is intrinsic to the shape context definition since all measurements are taken with respect to points on the object. To achieve scale invariance, all radial distances are normalized by the mean distance between pairs of points.

Shape contexts enable one to solve the correspondence problem between two similar but not identical shapes, such as seen in Figure 24.20(a) and (b). Shape contexts will be different for different points on a single shape S , whereas corresponding (homologous) points on similar shapes S and S' will tend to have similar shape contexts. We can then set up the problem of finding corresponding points between the two shapes as that of finding partners which have similar shape contexts.

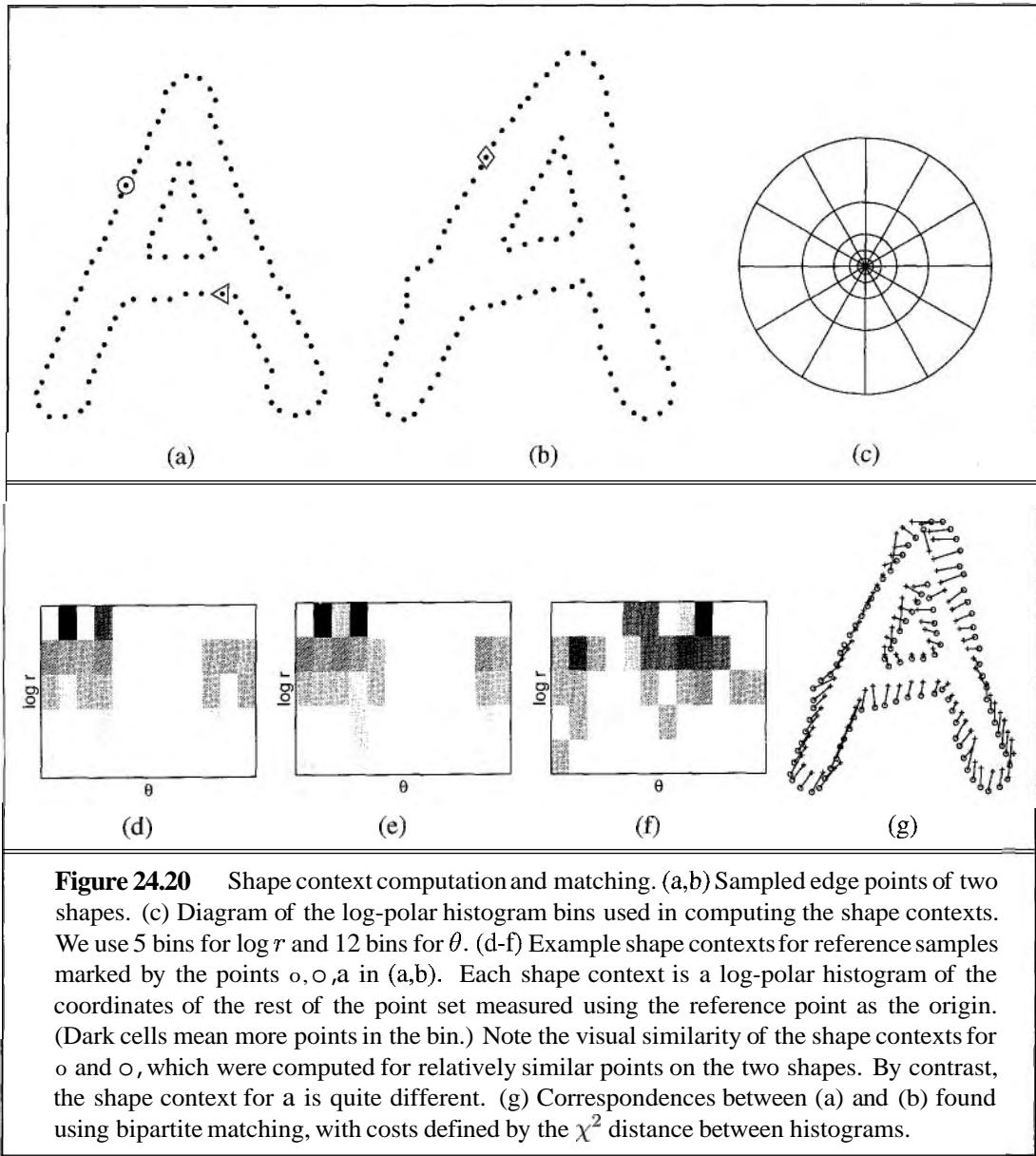
More precisely, consider a point p_i on the first shape: and a point q_j on the second shape. Let $C_{ij} = C(p_i, q_j)$ denote the cost of matching these two points. As shape contexts are distributions represented as histograms, it is natural to use the χ^2 distance:

$$C_{ij} = \frac{1}{2} \sum_{k=1}^K \frac{[h_i(k) - h_j(k)]^2}{h_i(k) + h_j(k)},$$

where $h_i(k)$ and $h_j(k)$ denote the k th bin of the normalized histograms at p_i and q_j . Given the set of costs C_{ij} between all pairs of points i on the first shape and j on the second shape we want to minimize the total cost of matching subject to the constraint that the matching be one-to-one. This is an instance of the **weighted bipartite matching** problem, which can be solved in $O(N^3)$ time using the Hungarian algorithm.

Given the correspondences at sample points, the correspondence can be extended to the complete shape by estimating an aligning transformation that maps one shape onto the other. Regularized thin plate splines are particularly effective. Once the shapes are aligned,

⁴ In modern computer graphics, this idea is referred to as **morphing**.



computing similarity scores is relatively straightforward. The distance between two shapes can be defined as a weighted sum of the shape context distances between corresponding points and the bending energy associated with the thin plate spline. Given this distance measure, one can use a simple nearest-neighbor classifier to solve the recognition problem. The excellent performance of this approach on handwritten digit classification was described in Chapter 20.

Pose Estimation

In addition to determining what an object is, we are also interested in determining its pose, i.e., its position and orientation with respect to the viewer. For instance, in an industrial

manipulation task, the robot arm cannot pick up an object until the pose is known. In the case of rigid objects, whether three-dimensional or two-dimensional, this problem has a simple and well defined solution based on the **alignment method**, which we will now develop.

The object is represented by M features or distinguished points m_1, m_2, \dots, m_M in three-dimensional space—perhaps the vertices of a polyhedral object. These are measured in some coordinate system that is natural for the object. The points are then subjected to an unknown three-dimensional rotation R , followed by translation by an unknown amount t and then projection to give rise to image feature points p_1, p_2, \dots, p_N on the image plane. In general, $N \neq M$, because some model points may be occluded, and the feature detector could miss some features (or invent false ones due to noise). We can express this as

$$p_i = \Pi(Rm_i + t) == Q(m_i)$$

for a three-dimensional model point m_i and the corresponding image point p_i . Here, R is a rotation matrix, t is a translation, and Π denotes perspective projection or one of its approximations, such as scaled orthographic projection. The net result is a transformation Q that will bring the the model point m_i into alignment with the image point p_i . Although we do not know Q initially, we do know (for rigid objects) that Q must be the *same* for all the model points.

One can solve for Q , given the three-dimensional coordinates of three model points and their two-dimensional projections. The intuition is as follows: one can write down equations relating the coordinates of p_i to those of m_i . In these equations, the unknown quantities correspond to the parameters of the rotation matrix R and the translation vector t . If we have enough equations, we ought to be able to solve for Q . We will not give a proof here; we merely state the following result:

Given three noncollinear points m_1, m_2 , and m_3 in the model, and their scaled orthographic projections p_1, p_2 , and p_3 on the image plane, there exist exactly two transformations from the three-dimensional model coordinate frame to a two-dimensional image coordinate frame.

These transformations are related by a reflection around the image plane and can be computed by a simple closed-form solution. If we could identify the corresponding model features for three features in the image, we could compute Q , the pose of the object. In the previous subsection, we discussed a technique for determining correspondences using shape context matching. If the object has well defined corners or other interest points, then an even simpler technique becomes available. The idea is to generate and test. We have to guess an initial correspondence of an image triplet with a model triplet and use the function FIND-TRANSFORM to hypothesize Q . If the guessed correspondence was correct, then Q will be correct and, when applied to the remaining model points, will result in the prediction of the image points. If the guessed correspondence was incorrect, then Q will be incorrect and, when applied to the remaining model points, would not predict the image points.

This is the basis of the ALIGN algorithm shown in Figure 24.21. The algorithm finds the pose for a given model, or returns with failure. The worst-case time complexity of the algorithm is proportional to the number of combinations of model triplets and image triplets, or $\binom{N}{3} \binom{M}{3}$, times the cost of verifying each combination. The cost of verification

```

function ALIGN(image, model) returns a solution or failure
  inputs: image, a list of image feature points
          model, a list of model feature points

  for each  $p_1, p_2, p_3$  in TRIPLETS(image) do
    for each  $m_1, m_2, m_3$  in TRIPLETS(model) do
       $Q \leftarrow \text{FIND-TRANSFORM}(p_1, p_2, p_3, m_1, m_2, m_3)$ 
      if projection according to  $Q$  explains image then
        return  $Q$ 
  return failure

```

Figure 24.21 An informal description of the alignment algorithm.

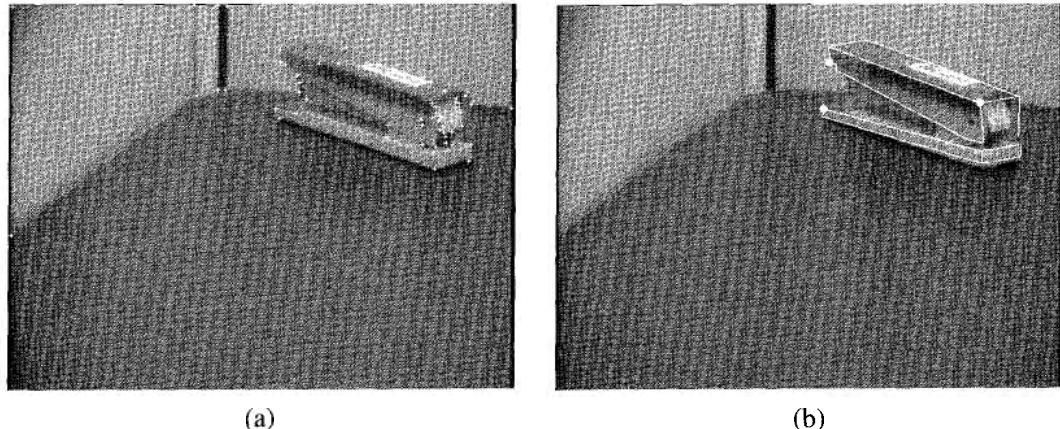


Figure 24.22 (a) Corners found in the stapler image. (b) Hypothesized reconstruction overlaid on the original image. (Courtesy of Clark Olson.)

is $M \log N$, as we must predict the image position of each of M model points, and find the distance to the nearest image point, a $\log N$ operation if the image points are arranged in an appropriate data structure. Thus, the worst-case complexity of the alignment algorithm is $O(M^4N^3 \log N)$, where M and N are the number of model and image points, respectively. Techniques based on pose clustering in combination with randomization bring the complexity down to $O(MN^3)$. Results from the application of this algorithm to the stapler image are shown in Figure 24.22.

24.6 USING VISION FOR MANIPULATION AND NAVIGATION

One of the principal uses of vision is to provide information both for manipulating objects—picking them up, grasping them, twirling them, and so on—and for navigating while avoiding obstacles. The ability to use vision for these purposes is present in the most primitive of

animal visual systems. In many cases, the visual system is minimal, in the sense that it extracts from the available light field just the information the animal needs to inform its behavior. Quite probably, modern vision systems evolved from early, primitive organisms that used a photosensitive spot at one end in order to orient themselves toward (or away from) the light. We saw in Section 24.4 that flies use a very simple optical flow detection system to land on walls. A classic study, *What the Frog's Eye Tells the Frog's Brain* (Lettvin *et al.*, 1959), observes of a frog that, "He will starve to death surrounded by food if it is not moving. His choice of food is determined only by size and movement."

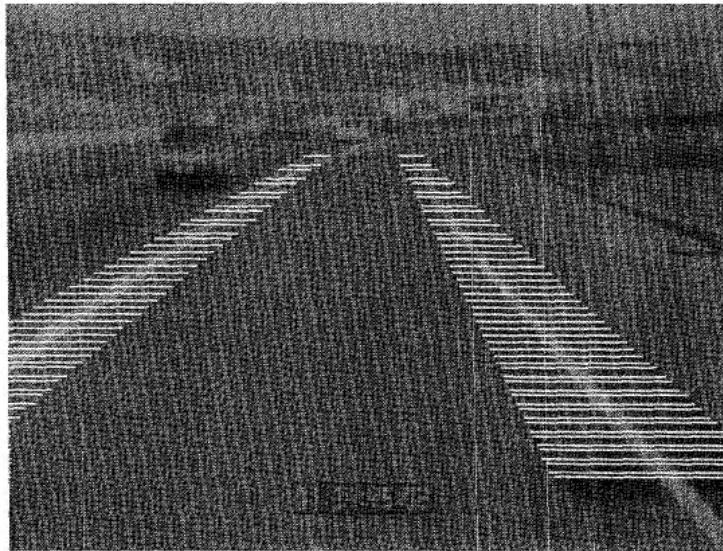


Figure 24.23 Image of a road taken from a camera inside the car. The horizontal white bars indicate the search windows within which the controller searches for the lane markers. The poor quality of the image is not untypical of low-resolution grayscale video.

Computer vision systems are used in “organisms” called robots. Let us consider a particular type of robot: an automated vehicle driving on a freeway. (See Figure 24.23.) First, we analyze the task; then, we identify the vision algorithms that will supply the information needed to perform those tasks well. The tasks faced by the driver include the following:

1. Lateral control—ensure that the vehicle remains securely within its lane or changes lane smoothly when required.
2. Longitudinal control—ensure that there is a safe distance to the vehicle in front.
3. Obstacle avoidance—monitor vehicles in neighboring lanes and be prepared for evasive maneuvers if one of them decides to change lanes,

The problem for the driver is to generate appropriate steering, acceleration, and braking actions to best accomplish these tasks.

For lateral control, one needs to maintain a representation of the position and orientation of the car relative to the lane. In the image shown in Figure 24.23, we can use edge detection algorithms to find edges corresponding to the lane marker segments. We can then fit smooth

curves to these edge elements. The parameters of these curves carry information about the lateral position of the car, the direction it is pointing relative to the lane, and the curvature of the lane. This information, along with information about the dynamics of the car, is all that is needed by the steering control system. Note also that because, from every frame to the next frame, there is only a small change in the position of the projection of the lane in the image, one knows *where* to look for the lane markers in the image—in the figure, we need to look only in the areas marked by parallel white bars.

For longitudinal control, one needs to know distances to the vehicles in front. This can be accomplished with binocular stereopsis or optical flow. Both approaches can be simplified by exploiting the domain constraints derived from the fact that one is driving on a planar surface. Using these techniques, vision-controlled cars can now drive at highway speeds for long periods.



The driving example makes one point very clear: *for a specific task, one does not need to recover all the information that, in principle, can be recovered from an image*. One does not need to recover the exact shape of every vehicle, solve for shape-from-texture on the grass surface adjacent to the freeway, and so on. The needs of the task require only certain kinds of information and one can gain considerable computational speed and robustness by recovering only that information and fully exploiting the domain constraints. Our purpose in discussing the general approaches in the previous section was that they form the basic theory, which one can specialize for the needs of particular tasks.

24.7 SUMMARY

Although perception appears to be an effortless activity for humans, it requires a significant amount of sophisticated computation. The goal of vision is to extract information needed for tasks such as manipulation, navigation, and object recognition.

- The process of **image formation** is well-understood in its geometric and physical aspects. Given a description of a three-dimensional scene, we can easily produce a picture of it from some arbitrary camera position (the graphics problem). Inverting the process by going from an image to a description of the scene is more difficult.
- To extract the visual information necessary for the tasks of manipulation, navigation, and recognition, intermediate representations have to be constructed. Early vision **image-processing** algorithms extract primitive features from the image, such as edges and regions.
- There are several cues in the image that enable one to obtain three-dimensional information about the scene: motion, stereopsis, texture, shading, and contour analysis. Each of these cues relies on background assumptions about physical scenes in order to provide nearly unambiguous interpretations.
- Object recognition in its full generality is a very hard problem. We discussed brightness-based and feature-based approaches. We also presented a simple algorithm for pose estimation. Other possibilities exist.

BIBLIOGRAPHICAL AND HISTORICAL NOTES

Systematic attempts to understand human vision can be traced back to ancient times. Euclid (ca. 300 B.C.) wrote about natural perspective—the mapping that associates, with each point P in the three-dimensional world, the direction of the ray OP joining the center of projection O to the point P . He was well aware of the notion of motion parallax. The mathematical understanding of perspective projection, this time in the context of projection onto planar surfaces, had its next significant advance in the 15th century in Renaissance Italy. Brunelleschi (1413) is usually credited with creating the first paintings based on geometrically correct projection of a three-dimensional scene. In 1435, Alberti codified the rules and inspired generations of artists whose artistic achievements amaze us to this day. Particularly notable in their development of the science of perspective, as it was called in those days, were Leonardo da Vinci and Albrecht Dürer. Leonardo's late 15th century descriptions of the interplay of light and shade (chiaroscuro), umbra and penumbra regions of shadows, and aerial perspective are still worth reading in translation (Kemp, 1989).

Although perspective was known to the Greeks, they were curiously confused by the role of the eyes in vision. Aristotle thought of the eyes as devices emitting rays, rather in the manner of modern laser range finders. This mistaken view was laid to rest by the work of Arab scientists, such as Alhazen, in the 10th century. The development of various kinds of cameras followed. These consisted of rooms (*camera* is Latin for "chamber") where light would be let in through a small hole in one wall to cast an image of the scene outside on the opposite wall. Of course, in all these cameras, the image was inverted, which caused no end of confusion. If the eye was to be thought of as such an imaging device, how do we see right side up? This enigma exercised the greatest minds of the era (including Leonardo). It took the work of Kepler and Descartes to settle the question. Descartes placed an eye from which the opaque cuticle had been removed in a hole in a window shutter. The result was an inverted image formed on a piece of paper laid out on the retina. While the retinal image is indeed inverted, this does cause a problem because the brain interprets the image the right way. In modern jargon, one just has to access the data structure appropriately.

The next major advances in the understanding of vision took place in the 19th century. The work of Helmholtz and Wundt, described in Chapter 1, established psychophysical experimentation as a rigorous scientific discipline. Through the work of Young, Maxwell, and Helmholtz, a trichromatic theory of color vision was established. That humans can see depth if the images presented to the left and right eyes are slightly different was demonstrated by Wheatstone's (1838) invention of the stereoscope. The device immediately became popular in parlors and salons throughout Europe. The essential concept of binocular stereopsis—that two images of a scene taken from slightly different viewpoints carry information sufficient to obtain a three-dimensional reconstruction of the scene, was exploited in the field of photogrammetry. Key mathematical results were obtained; for example, Kruppa (1913) proved that, given two views of five distinct points, one could reconstruct the rotation and translation between the two camera positions as well as the depth of the scene (up to a scale factor). Although the geometry of stereopsis had been understood for a long time, the correspondence

problem in photogrammetry used to be solved by humans trying to match up corresponding points. The amazing ability of humans in solving the correspondence problem was illustrated by Julesz's (1971) invention of the random dot stereogram. Both in computer vision and in photogrammetry, much effort was devoted to solving the correspondence problem in the 1970s and 1980s.

The second half of the 19th century was a major foundational period for the psychophysical study of human vision. In the first half of the 20th century, the most significant research results in vision were obtained by the Gestalt school of psychology, led by Max Wertheimer. With the slogan "The whole is different from the sum of the parts," they promoted the view that complete forms, rather than components such as edges, should be the primary units of perception.

The period after World War II was marked by renewed activity. Most significant was the work of J. J. Gibson (1950, 1979), who pointed out the importance of optical flow, as well as texture gradients in the estimation of environmental variables such as surface slant and tilt. He reemphasized the importance of the stimulus and how rich it was. Gibson, Olum, and Rosenblatt (1955) pointed out that the optical flow field contained enough information to determine the egomotion of the observer relative to the environment. In the computational vision community, work in that area and in the (mathematically equivalent) area of gleaned structure from motion developed mainly in the 1980s and 1990s. The seminal work of Koenderink and van Doorn (1975), Ullman (1979), and Longuet-Higgins (1981) sparked this activity. Early concerns about the stability of structure from motion were allayed by the work of Tomasi and Kanade (1992) who showed that with the use of multiple frames, and the resulting wide base line, shape could be recovered quite accurately.

Chan *et al.* (1998) describe the astounding visual apparatus of the fly, which has temporal visual acuity ten times greater than humans. That is, a fly could watch a movie projected at up to 300 frames per second and recognize individual frames.

A conceptual innovation introduced in the 1990s was the study of projective structure from motion. In this setting camera calibration is not necessary, as was shown by Faugeras (1992). This discovery is related to the introduction of the use of geometrical invariants in object recognition, as surveyed by Mundy and Zisserman (1992), and the development of affine structure from motion by Koenderink and Van Doorn (1991). In the 1990s, with great increase in computer speed and storage, and the widespread availability of digital video, motion analysis found many new applications. Building geometrical models of real world scenes for rendering by computer graphics techniques proved particularly popular, led by reconstruction algorithms such as the one developed by Debevec, Taylor and Malik (1996). The books by Hartley and Zisserman (2000) and Faugeras *et al.* (2001) provide a comprehensive treatment of the geometry of multiple views.

In computational vision, major early works in inferring shape from texture are due to Bajscy and Liebermann (1976) and Stevens (1981). Whereas this work was for planar surfaces, a comprehensive analysis for curved surfaces is due to Garding (1992) and Malik and Rosenholtz (1997).

In the computational vision community, inferring shape from shading was first studied by Berthold Horn (1970). Horn and Brooks (1989) present an extensive survey of the main

papers in the area. This framework made a number of simplifying assumptions, the most critical of which was ignoring the effect of mutual illumination. The importance of mutual illumination has been well appreciated in the computer graphics community, where ray tracing and radiosity have been developed precisely to take mutual illumination into account. A theoretical and empirical critique may be found in Forsyth and Zisserman (1991).

In the area of inferring shape from contour, after the key initial contributions of Huffman (1971) and Clowes (1971), Mackworth (1973) and Sugihara (1984) completed the analysis for polyhedral objects. Malik (1987) developed a labeling scheme for piecewise smooth curved objects. Kirousis and Papadimitriou (1988) showed that line-labeling for trihedral scenes is NP-complete.

Understanding the visual events in the projection of smooth curved objects requires an interplay between differential geometry and singularity theory. The best study is Koenderink's (1990) *Solid Shape*.

The seminal work in three-dimensional object recognition was Roberts's (1963) thesis at MIT. It is often considered to be the first PhD thesis in computer vision and it introduced several key ideas, including edge detection and model-based matching. Canny edge detection was introduced in Canny (1986). The idea of alignment, also first introduced by Roberts, resurfaced in the 1980s in the work of Lowe (1987) and Huttenlocher and Ullman (1990). Significant improvements in the efficiency of pose estimation by alignment were obtained by Olson (1994). Another major strand in research on 3D object recognition has been the approach based on the idea of describing shapes in terms of volumetric primitives, with **generalized cylinders**, introduced by Tom Binford (1971), proving particularly popular.

While computer vision research on object recognition largely focused on issues arising from the projection of three-dimensional objects onto two-dimensional images, there was a parallel tradition in the pattern recognition community that viewed the problem as one of pattern classification. The motivating examples were in domains such as optical character recognition and handwritten zip code recognition where the primary concern is that of learning the typical variations characteristic of a class of objects and separating them from other classes. See LeCun *et al.* (1995) for a comparison of approaches. Other work on object recognition includes that of Sirovitch and Kirby (1987) and of Viola and Jones (2002) for face recognition. Belongie *et al.* (2002) describe the shape context approach. Dickmanns and Zapp (1987) first demonstrated visually controlled car driving on freeways at high speeds; Pomerleau (1993) achieved similar performance using a neural network approach.

Vision Science: Photons to Phenomenology by Stephen Palmer (1999) provides the best comprehensive treatment of human vision; the books *Eye, Brain and Vision* by David Hubel (1988) and *Perception* by Irvin Rock (1984) are short introductions centered on neurophysiology and perception respectively.

For the field of computer vision, the most comprehensive textbook available today is *Computer Vision: A Modern Approach* by David Forsyth and Jean Ponce. Considerably shorter accounts can be found in the books by Nalwa (1993) and by Trucco and Verri (1998). *Robot Vision* (Horn, 1986) and *Three-Dimensional Computer Vision* (Faugeras, 1993) are two older and still useful textbooks, each with its specialized set of topics. David Marr's book *Vision* (Marr, 1982) played a major role in connecting computer vision to the traditional

areas of biological vision—psychophysics and neurobiology. Two of the main journals for computer vision are the *IEEE Transactions on Pattern Analysis and Machine Intelligence* and the *International Journal of Computer Vision*. Computer vision conferences include ICCV (International Conference on Computer Vision), CVPR (Computer Vision and Pattern Recognition), and ECCV (European Conference on Computer Vision).

EXERCISES

24.1 In the shadow of a tree with a dense, leafy canopy, one sees a number of light spots. Surprisingly, they all appear to be circular. Why? After all, the gaps between the leaves through which the sun shines are not likely to be circular.

24.2 Label the line drawing in Figure 24.24, assuming that the outside edges have been labeled as occluding and that all vertices are trihedral. Do this by a backtracking algorithm that examines the vertices in the order A, B, C, and D, picking at each stage a choice consistent with previously labeled junctions and edges. Now try the order B, D, A, and C.

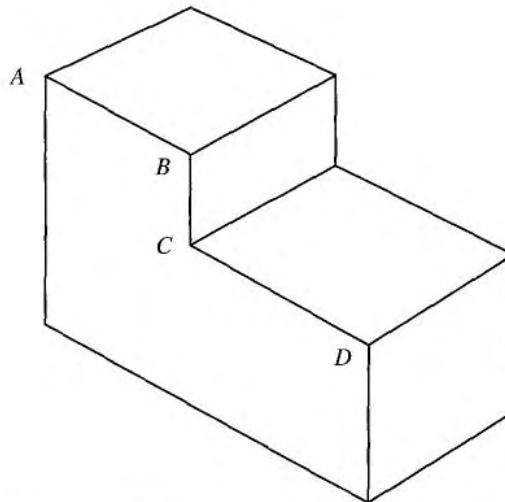


Figure 24.24 A drawing to be labeled, in which all vertices are trihedral.

24.3 Consider an infinitely long cylinder of radius r oriented with its axis along the y-axis. The cylinder has a Lambertian surface and is viewed by a camera along the positive x-axis. What will you expect to see in the image if the cylinder is illuminated by a point source at infinity located on the positive x-axis? Explain your answer by drawing the isobrightness contours in the projected image. Are the contours of equal brightness uniformly spaced?

24.4 Edges in an image can correspond to a variety of events in a scene. Consider the cover of this book, and assume that it is a picture of a real three-dimensional scene. Identify

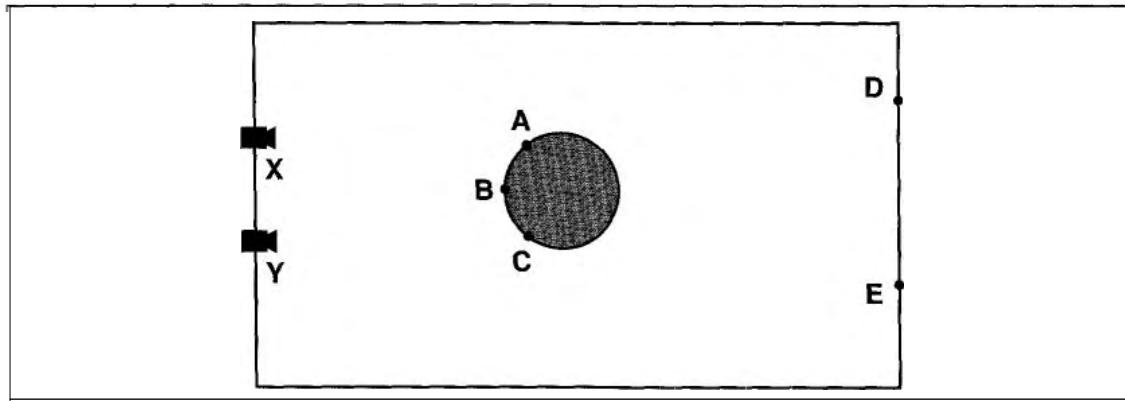


Figure 24.25 Top view of a two-camera vision system observing a bottle with a wall behind.

ten different brightness edges in the image, and for each, state whether it corresponds to a discontinuity in (a) depth, (b) surface normal, (c) reflectance; or (d) illumination.

24.5 Show that convolution with a given function f commutes with differentiation; that is, show that $(f * g)' = f * g'$.

24.6 A stereoscopic system is being contemplated for terrain mapping. It will consist of two CCD cameras, each having 512×512 pixels on a $10 \text{ cm} \times 10 \text{ cm}$ square sensor. The lenses to be used have a focal length of 16 cm, with the focus fixed at infinity. For corresponding points (u_1, v_1) in the left image and (u_2, v_2) in the right image, $v_1 = v_2$ because the x-axes in the two image planes are parallel to the epipolar lines. The optical axes of the two cameras are parallel. The baseline between the cameras is 1 meter.

a. If the nearest range to be measured is 16 meters, what is the largest disparity that will occur (in pixels)?

b. What is the range resolution at 16 meters, due to the pixel spacing?

c. What range corresponds to a disparity of one pixel?

24.7 Suppose we wish to use the alignment algorithm in an industrial situation in which flat parts move along a conveyor belt and are photographed by a camera vertically above the conveyor belt. The pose of the part is specified by three variables--one for the rotation and two for the two-dimensional position. This simplifies the problem and the function FIND-TRANSFORM needs only two pairs of corresponding image and model features to determine the pose. Determine the worst-case complexity of the alignment procedure.

24.8 (Courtesy of Pietro Perona.) Figure 24.25 shows two cameras at X and Y observing a scene. Draw the image seen at each camera, assuming that all named points are in the same horizontal plane. What can be concluded from these two images about the relative distances of points A, B, C, D, and E from the camera baseline, and on what basis?

24.9 Which of the following are true, and which are false?

a. Finding corresponding points in stereo images is the easiest phase of the stereo depth-finding process.

- b. Shape-from-texture can be done by projecting a grid of light-stripes onto the scene.
- c. The Huffman–Clowes labelling scheme can deal with all polyhedral objects.
- d. In line drawings of curved objects, the line label can change from one end of the line to the other.
- e. In stereo views of the same scene, greater accuracy is obtained in the depth calculations if the two camera positions are further apart.
- f. Lines with equal lengths in the scene always project to equal lengths in the image.
- g. Straight lines in the image necessarily correspond to straight lines in the scene.

24.10 Figure 24.23 is taken from the point of view of a car in the exit lane of a freeway. Two cars are visible in the lane immediately to the left. What reasons does the viewer have to conclude that one is closer than the other?

25 ROBOTICS

In which agents are endowed with physical effectors with which to do mischief.

25.1 INTRODUCTION

ROBOTS

Robots are physical agents that perform tasks by manipulating the physical world. To do so, they are equipped with **effectors** such as legs, wheels, joints, and grippers. Effectors have a single purpose: to assert physical forces on the environment.¹ Robots are also equipped with **sensors**, which allow them to perceive their environment. Present day robotics employs a diverse set of sensors, including cameras and ultrasound to measure the environment, and gyroscopes and accelerometers to measure the robot's own motion.

EFFECTOR

SENSOR

MANIPULATOR

MOBILE ROBOT

ULV

UAV

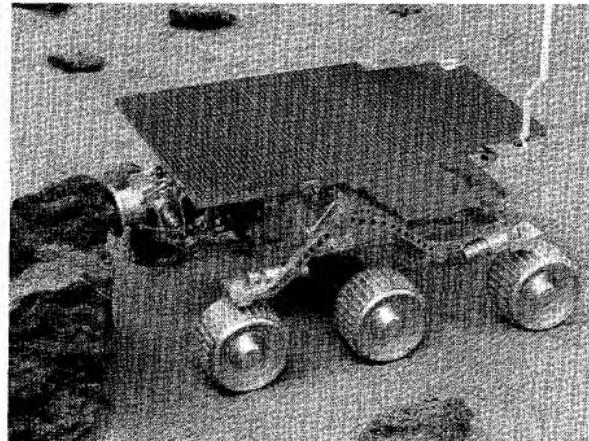
AUV

PLANETARY ROVER

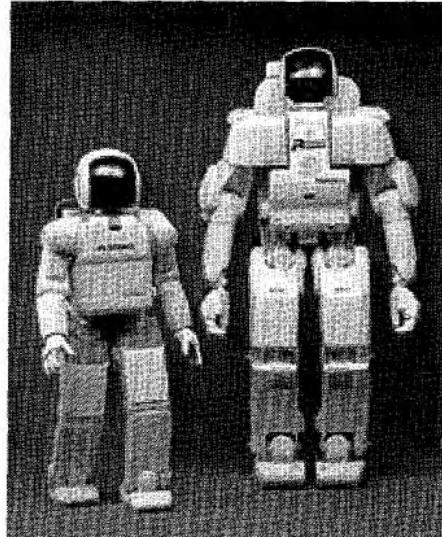
Most of today's robots fall into one of three primary categories. **Manipulators**, or robot arms, are physically anchored to their workplace, for example in a factory assembly line or on the International Space Station. Manipulator motion usually involves an entire chain of controllable joints, enabling such robots to place their effectors in any position within the workplace. Manipulators are by far the most common type of industrial robots, with over a million units installed worldwide. Some mobile manipulators are used in hospitals to assist surgeons. Few car manufacturers could survive without robotic manipulators, and some manipulators have even been used to generate original artwork.

The second category is the **mobile robot**. Mobile robots move about their environment using wheels, legs, or similar mechanisms. They have been put to use delivering food in hospitals, moving containers at loading docks, and similar tasks. Earlier we encountered an example of a mobile robot: the NAVLAB **unmanned land vehicle** (ULV) capable of driverless autonomous highway navigation. Other types of mobile robots include **unmanned air vehicles** (UAV), commonly used for surveillance, crop-spraying, and military operations, **autonomous underwater vehicles** (AUV), used in deep sea exploration, and **planetary rovers**, such as the Sojourner robot shown in Figure 25.1(a).

¹ In Chapter 2 we talked about **actuators**, not effectors. An actuator is a control line that communicates a command to an effector; the effector is the physical device itself.



(a)



(b)

Figure 25.1 (a) NASA's Sojourner, a mobile robot that explored the surface of Mars in July 1997. (b) Honda's P3 and Asimo humanoid robots.

HUMANOID ROBOT

The third type is a hybrid: a mobile robot equipped with manipulators. These include the **humanoid robot**, whose physical design mimics the human torso. Figure 25.1(b) shows two such humanoid robots, both manufactured by Honda Corp. in Japan. Hybrids can apply their effectors further afield than anchored manipulators can, but their task is made harder because they don't have the rigidity that the anchor provides.

The field of robotics also includes prosthetic devices (artificial limbs, ears, and eyes for humans), intelligent environments (such as an entire house that is equipped with sensors and effectors), and multibody systems, wherein robotic action is achieved through swarms of small cooperating robots.

Real robots usually must cope with environments that are partially observable, stochastic, dynamic, and continuous. Some, but not all, robot environments are sequential and multiagent as well. Partial observability and stochasticity are the result of dealing with a large, complex world. The robot cannot see around corners, and motion commands are subject to uncertainty due to gears slipping, friction, etc. Also, the real world stubbornly refuses to operate faster than real time. In a simulated environment, it is possible to use simple algorithms (such as the **Q-learning** algorithm described in Chapter 21) to learn in a few CPU hours from millions of trials. In a real environment, it might take years to run these trials. Furthermore, real crashes really hurt, unlike simulated ones. Practical robotic systems need to embody prior knowledge about the robot, its physical environment, and the tasks that the robot will perform so that the robot can learn quickly and perform safely.

25.2 ROBOT HARDWARE

So far in this book, we have taken the agent architecture—sensors, effectors, and processors—as given, and we have concentrated on the agent program. The success of real robots depends at least as much on the design of sensors and effectors that are appropriate for the task.

Sensors

PASSIVE SENSOR

ACTIVE SENSOR

RANGE FINDER

SONAR SENSOR

Sensors are the perceptual interface between robots and their environments. **Passive sensors**, such as cameras, are true observers of the environment: they capture signals that are generated by other sources in the environment. **Active sensors**, such as sonar, send energy into the environment. They rely on the fact that this energy is reflected back to the sensor. Active sensors tend to provide more information than passive sensors, but at the expense of increased power consumption and with a danger of interference when multiple active sensors are used at the same time. Whether active or passive, sensors can be divided into three types, depending on whether they record distances to objects, entire images of the environment, or properties of the robot itself.

Many mobile robots make use of **range finders**, which are sensors that measure the distance to nearby objects. One common type is the **sonar sensor**, also known as an ultrasonic transducer. Sonar sensors emit directional sound waves, which are reflected by objects, with some of the sound making it back into the sensor. The time and intensity of this returning signal thus carry information about the distance to nearby objects. Underwater sonar sensors are the technology of choice for AUVs. On land, sonar sensors are mainly used for near-range

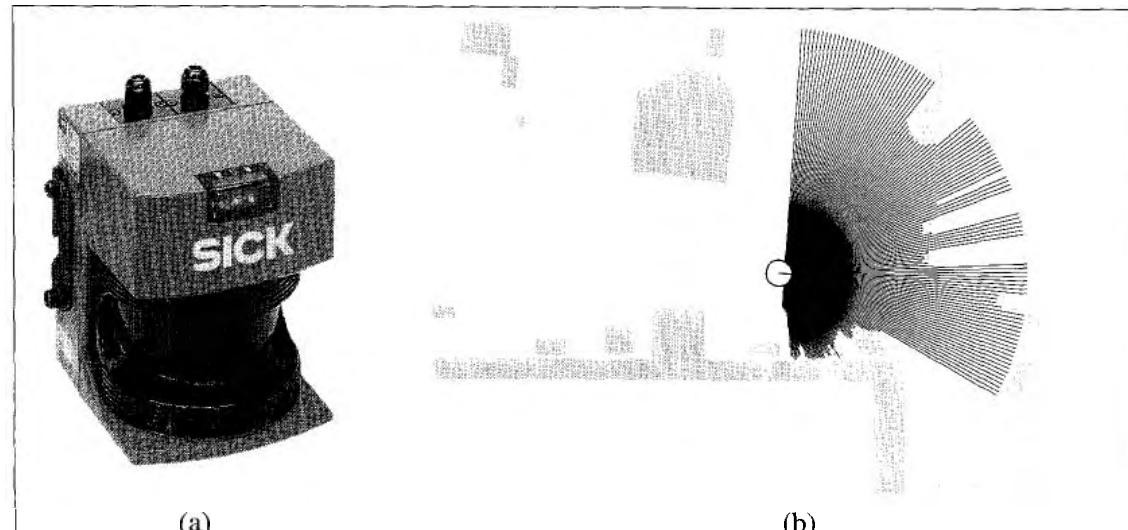


Figure 25.2 (a) The SICK LMS laser range scanner; a popular range sensor for mobile robots. (b) Range scan obtained with a horizontally mounted sensor, projected onto a two-dimensional environment map.

collision avoidance, due to their limited angular resolution. Alternatives to sonar include radar (used primarily by aircraft) and laser. A laser range finder is shown in Figure 25.2.

TACTILE SENSORS
GLOBAL POSITIONING SYSTEM

DIFFERENTIAL GPS

IMAGING SENSOR

PROPRIOCEPTIVE SENSOR

SHAFT DECODER

ODOMETRY

INERTIAL SENSOR

FORCE SENSOR

TORQUE SENSOR

DEGREE OF FREEDOM

KINEMATIC STATE

POSE

DYNAMIC STATE

Some range sensors measure very short or very long distances. Close-range sensors include **tactile sensors** such as whiskers, bump panels, and touch-sensitive skin. At the other end of the spectrum is the **Global Positioning System** (GPS), which measures the distance to satellites that emit pulsed signals. At present, there are two dozen satellites in orbit, each transmitting signals on two different frequencies. GPS receivers can recover the distance to these satellites by analyzing phase shifts. By triangulating signals from multiple satellites, GPS receivers can determine their absolute location on Earth to within a few meters. **Differential GPS** involves a second ground receiver with known location, providing millimeter accuracy under ideal conditions. Unfortunately, GPS does not work indoors or underwater.

The second important class of sensors is **imaging sensors**—the cameras that provide us with images of the environment and, using the computer vision techniques of Chapter 24, models and features of the environment. Stereo vision is particularly important in robotics, because it can capture depth information; although its future is somewhat uncertain as new active technologies for range imaging are being developed successfully.

The third important class is **proprioceptive sensors**, which inform the robot of its own state. To measure the exact configuration of a robotic joint, motors are often equipped with **shaft decoders** that count the revolution of motors in small increments. On robot arms, shaft decoders can provide accurate information over any period of time. On mobile robots, shaft decoders that report wheel revolutions can be used for **odometry**—the measurement of distance travelled. Unfortunately, wheels tend to drift and slip, so odometry is accurate only over short distances. External forces, such as the current for AUVs and the wind for UAVs, increase positional uncertainty. **Inertial sensors**, such as gyroscopes, can help but cannot by themselves prevent the inevitable accumulation of position uncertainty.

Other important aspects of robot state are measured by **force** and **torque sensors**. These are indispensable when robots handle fragile objects or objects whose exact shape and location is unknown. Imagine a one ton robotic manipulator screwing in a light bulb. It would be all too easy to apply too much force and break the bulb. Force sensors allow the robot to sense how hard it is gripping the bulb, and torque sensors allow it to sense how hard it is turning. Good sensors can measure forces in three translational and three rotational directions.

Effectors

Effectors are the means by which robots move and change the shape of their bodies. To understand the design of effectors, it will help first to talk about motion and shape in the abstract, using the concept of a **degree of freedom** (DOF). We count one degree of freedom for each independent direction in which a robot, or one of its effectors, can move. For example, a rigid free-moving robot such as an AUV has six degrees of freedom, three for its (x, y, z) location in space and three for its angular orientation, known as **yaw**, roll, and pitch. These six degrees define the **kinematic state**² or **pose** of the robot. The **dynamic state** of a robot includes one additional dimension for the rate of change of each kinematic dimension.

² "Kinematic" is from the Greek word for motion, as is "cinema."

REVOLUTE JOINT
PRISMATIC JOINT

For nonrigid bodies, there are additional degrees of freedom within the robot itself. For example, in a human arm, the elbow has one degree of freedom—it can flex in one direction—and the wrist has three degrees of freedom—it can move up and down, side to side, and can also rotate. Robot joints also have 1, 2, or 3 degrees of freedom each. Six degrees of freedom are required to place an object, such as a hand, at a particular point in a particular orientation. The arm in Figure 25.3(a) has exactly six degrees of freedom, created by five **revolute joints** that generate rotational motion and one **prismatic joint** that generates sliding motion. You can verify that the human arm as a whole has more than six degrees of freedom by a simple experiment: put your hand on the table, and notice that you still have the freedom to rotate your elbow without changing the configuration of your hand. Manipulators that have more degrees of freedom than required to place an end effector at a target location are easier to control than robots with only the minimum number of DOFs.

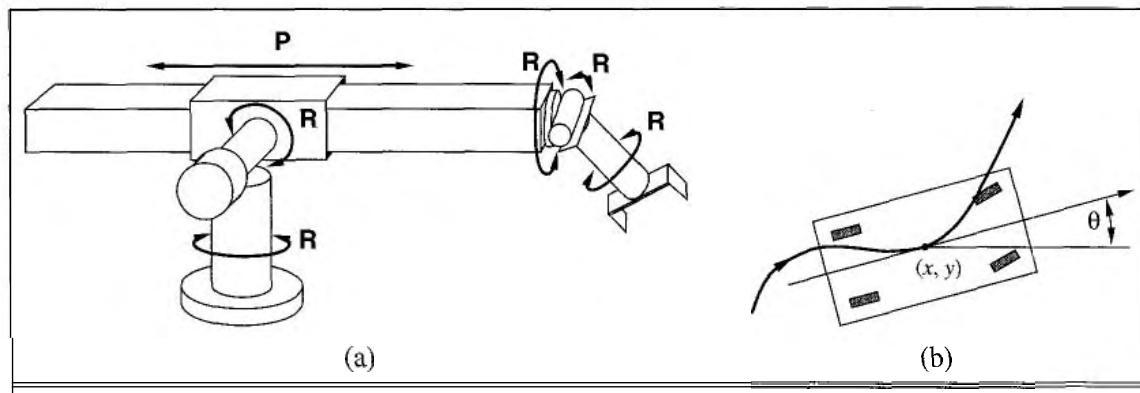


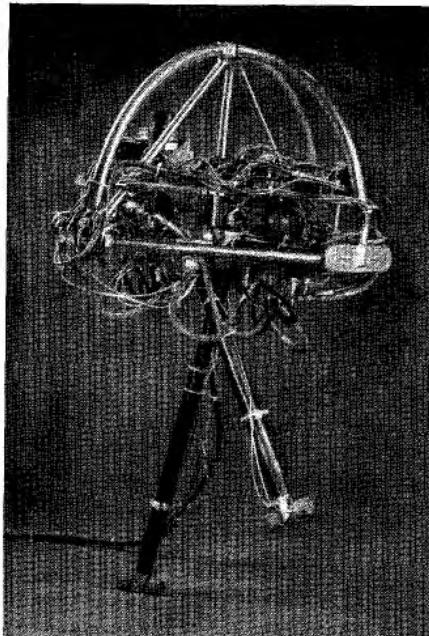
Figure 25.3 (a) The Stanford Manipulator, an early robot arm with five revolute joints (R) and one prismatic joint (P), for a total of six degrees of freedom. (b) Motion of a nonholonomic four-wheeled vehicle with front-wheel steering.

EFFECTIVE DOF
CONTROLLABLE DOF
NONHOLONOMIC

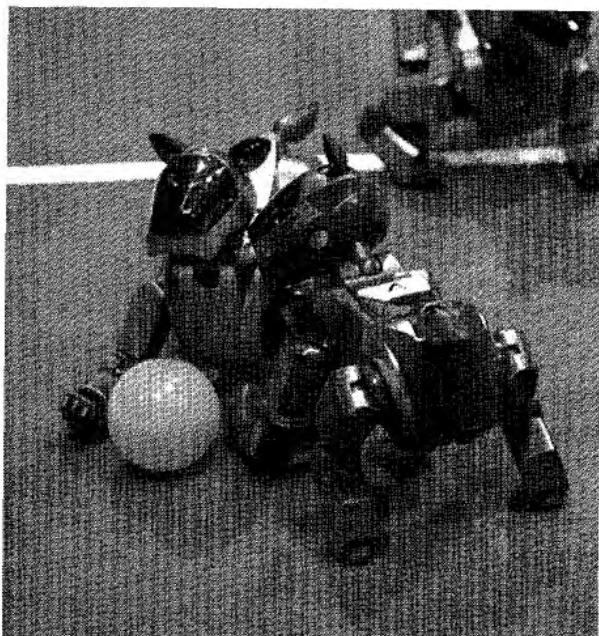
For mobile robots, the DOFs are not necessarily the same as the number of actuated elements. Consider, for example, your average car: it can move forward or backward, and it can turn, giving it two DOFs. In contrast, a car's kinematic configuration is three-dimensional: on an open flat surface, one can easily maneuver a car to any (x, y) point, in any orientation. (See Figure 25.3(b).) Thus, the car has **3 effective degrees of freedom** but **2 controllable degrees of freedom**. We say a robot is **nonholonomic** if it has more effective DOFs than controllable DOFs and **holonomic** if the two numbers are the same. Holonomic robots are easier to control—it would be much easier to park a car that could move sideways as well as forward and backward—but holonomic robots are also mechanically more complex. Most robot arms are holonomic, and most mobile robots are nonholonomic.

DIFFERENTIAL DRIVE
SYNCHRO DRIVE

For mobile robots, there exists a range of mechanisms for locomotion, including wheels, tracks, and legs. **Differential drive** robots possess two independently actuated wheels (or tracks), one on each side, as on a military tank. If both wheels move at the same velocity, the robot moves on a straight line. If they move in opposite directions, the robot turns on the spot. An alternative is the **synchro drive**, in which each wheel can move and turn around its own axis. This could easily lead to chaos, if not for the constraint that all wheels always



(a)



(b)

Figure 25.4 (a) One of Marc Raibert's legged robots in motion. (b) Sony AIBO robots playing soccer. (© 2001, The RoboCup Federation.)

point in the same direction and move at the same speed. Both differential and synchro drives are nonholonomic. Some more expensive robots use holonomic drives, which usually involve three or more wheels that can be oriented and moved independently.

Legs, unlike wheels, can handle very rough terrain. However, legs are notoriously slow on flat surfaces, and they are mechanically difficult to build. Robotics researchers have tried designs ranging from one leg up to dozens of legs. Legged robots have been made to walk, run, and even hop—as we see with the legged robot in Figure 25.4(a). This robot is **dynamically stable**, meaning that it can remain upright while hopping around. A robot that can remain upright without moving its legs is called **statically stable**. A robot is statically stable if its center of gravity is above the polygon spanned by its legs.

Other types of mobile robot use vastly different mechanisms for moving about. Airborne vehicles usually use propellers or turbines. Robotic blimps rely on thermal effects to keep themselves aloft. Autonomous underwater vehicles often use thrusters, similar to those used on submarines.

Sensors and effectors alone do not make a robot. A complete robot also needs a source of power to drive its effectors. The **electric motor** is the most popular mechanism for both manipulator actuation and locomotion, but **pneumatic actuation** using compressed gas and **hydraulic actuation** using pressurized fluids also have their application niches. Most robots also have some means of digital communication such as a wireless network. Finally, there has to be a body frame to hang all the bits and pieces on and a soldering iron for emergencies.

DYNAMICALLY
STABLE
STATICALLY STABLE

ELECTRIC MOTOR
PNEUMATIC
ACTUATION
HYDRAULIC
ACTUATION

25.3 ROBOTIC PERCEPTION

Perception is the process by which robots map sensor measurements into internal representations of the environment. Perception is difficult because in general the sensors are noisy, and the environment is partially observable, unpredictable, and often dynamic. As a rule of thumb, good internal representations have three properties: they contain enough information for the robot to make the right decisions, they are structured so that they can be updated efficiently, and they are natural in the sense that internal variables correspond to natural state variables in the physical world.

In Chapter 15, we saw that Kalman filters, HMMs, and dynamic Bayes nets can represent the transition and sensor models of a partially observable environment, and we described both exact and approximate algorithms for updating the belief state—the posterior probability distribution over the environment state variables. Several dynamic Bayes net models for this process were shown in Chapter 15. For robotics problems, we usually include the robot's own past actions as observed variables in the model, as in the network shown in Figure 17.9. Figure 25.5 shows the notation used in this chapter: \mathbf{X}_t is the state of the environment (including the robot) at time t , \mathbf{Z}_t is the observation received at time t , and A_t is the action taken after the observation is received.

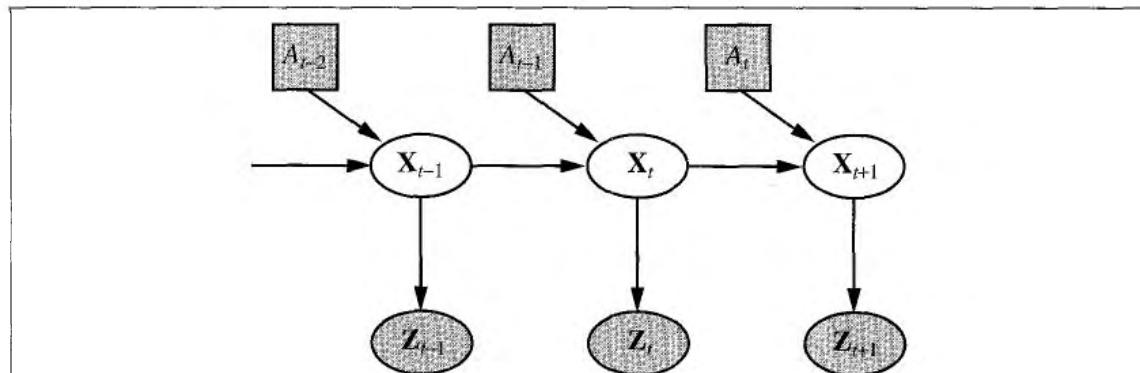


Figure 25.5 Robot perception can be viewed as temporal inference from sequences of actions and measurements, as illustrated by this dynamic Bayes network.

The task of filtering, or updating the belief state, is essentially the same as in Chapter 15. The task is to compute the new belief state, $\mathbf{P}(\mathbf{X}_{t+1} \mid \mathbf{z}_{1:t+1}, a_{1:t})$, from the current belief state $\mathbf{P}(\mathbf{X}_t \mid \mathbf{z}_{1:t}, a_{1:t-1})$ and the new observation \mathbf{z}_{t+1} . The principal differences are that (1) we condition explicitly on the actions as well as the observations, and (2) we must now deal with *continuous* rather than *discrete* variables. Thus, we modify the recursive filtering equation (15.3) to use integration rather than summation:

$$\begin{aligned} & \mathbf{P}(\mathbf{X}_{t+1} \mid \mathbf{z}_{1:t+1}, a_{1:t}) \\ &= \alpha \mathbf{P}(\mathbf{z}_{t+1} \mid \mathbf{X}_{t+1}) \int \mathbf{P}(\mathbf{X}_{t+1} \mid \mathbf{x}_t, a_t) P(\mathbf{x}_t \mid \mathbf{z}_{1:t}, a_{1:t-1}) d\mathbf{x}_t . \end{aligned} \quad (25.1)$$

MOTION MODEL

The equation states that the posterior over the state variables \mathbf{X} at time $t + 1$ is calculated recursively from the corresponding estimate one time step earlier. This calculation involves the previous action a_t and the current sensor measurement \mathbf{z}_{t+1} . For example, if our goal is to develop a soccer-playing robot, \mathbf{X}_{t+1} might be the location of the soccer ball relative to the robot. The posterior $\mathbf{P}(\mathbf{X}_t \mid \mathbf{z}_{1:t}, a_{1:t-1})$ is a probability distribution over all states that captures what we know from past sensor measurements and controls. Equation (25.1) tells us how to recursively estimate this location, by incrementally folding in sensor measurements (e.g., camera images) and robot motion commands. The probability $\mathbf{P}(\mathbf{X}_{t+1} \mid \mathbf{x}_t, a_t)$ is called the **transition model** or **motion model**, and $\mathbf{P}(\mathbf{z}_{t+1} \mid \mathbf{X}_{t+1})$ is the **sensor model**.

LOCALIZATION

Localization

Localization is a generic example of robot perception. It is the problem of determining where things are. Localization is one of the most pervasive perception problems in robotics, because knowledge about where things are is at the core of any successful physical interaction. For example, robot manipulators must know the location of objects they manipulate. Navigating robots must know where they are in order to find their way to goal locations.

TRACKING
GLOBAL LOCALIZATION

The localization problem comes in three flavors of increasing difficulty. If the initial pose of the object to be localized is known, localization is a **tracking** problem. Tracking problems are characterized by bounded uncertainty. More difficult is the **global localization** problem, in which the initial location of the object is entirely unknown. Global localization problems turn into tracking problems once the object of interest has been localized, but they also involve phases where the robot has to manage very broad uncertainties. Finally, we can be mean to our robot and "kidnap" the object it is attempting to localize. Localization under such devious conditions is known as the **kidnapping problem**. Kidnapping is often used to test the robustness of a localization technique under extreme conditions.

KIDNAPPING
PROBLEM

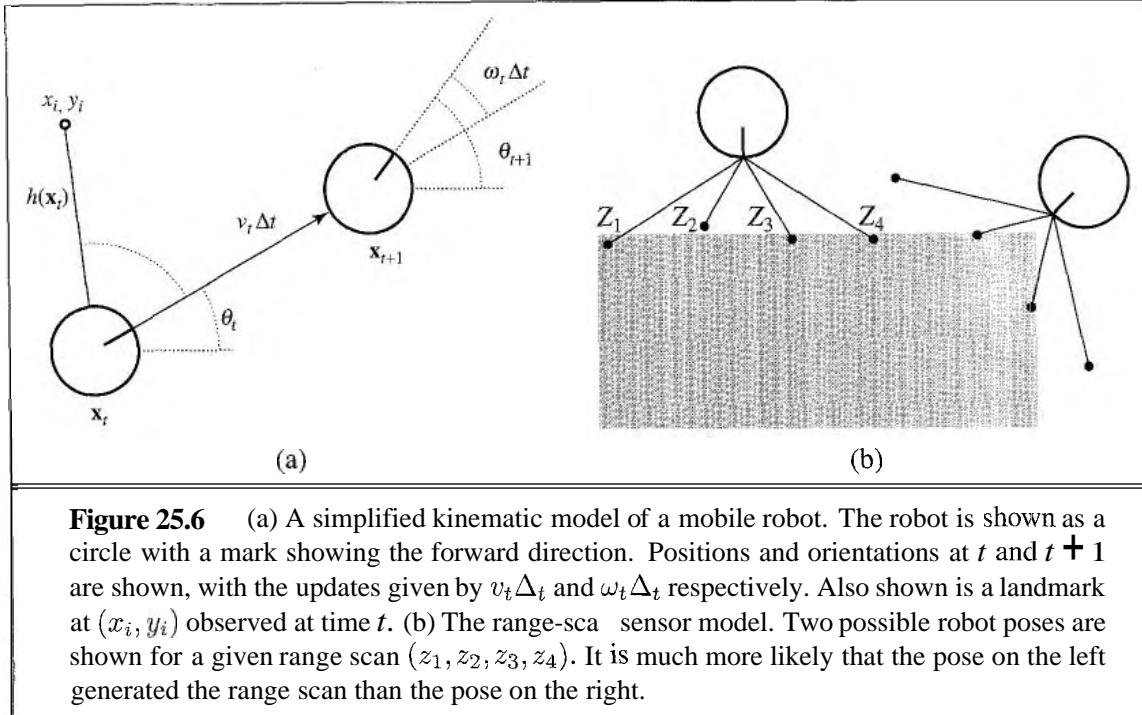
To keep things simple, let us assume that the robot moves slowly in a plane and that it is given an exact map of the environment. (An example of such a map appears in Figure 25.8.) The pose of such a mobile robot is defined by its two Cartesian coordinates with values x and y and its heading with value θ , as illustrated in Figure 25.6(a). (Notice that we exclude the corresponding velocities, so this is a kinematic rather than a dynamic model.) If we arrange those three values in a vector, then any particular state is given by $\mathbf{X}_t = (\mathbf{x}_t, y_t, \theta_t)^\top$.

In the kinematic approximation, each action consists of the "instantaneous" specification of two velocities—a translational velocity v_t and a rotational velocity ω_t . For small time intervals Δt , a crude deterministic model of the motion of such robots is given by

$$\hat{\mathbf{X}}_{t+1} = f(\mathbf{X}_t, \underbrace{v_t, \omega_t}_{a_t}) = \mathbf{X}_t + \begin{pmatrix} v_t \Delta t \cos \theta_t \\ v_t \Delta t \sin \theta_t \\ \omega_t \Delta t \end{pmatrix}.$$

The notation $\hat{\mathbf{X}}$ refers to a deterministic state prediction. Of course, physical robots are somewhat unpredictable. This is commonly modeled by a Gaussian distribution with mean $f(\mathbf{X}_t, v_t, \omega_t)$ and covariance Σ_x . (See Appendix A for a mathematical definition.)

$$\mathbf{P}(\mathbf{X}_{t+1} \mid \mathbf{X}_t, v_t, \omega_t) = N(\hat{\mathbf{X}}_{t+1}, \Sigma_x)$$



Next, we need a sensor model. We will consider two kinds of sensor model. The first assumes that the sensors detect *stable, recognizable* features of the environment called **landmarks**. For each landmark, the range and bearing are reported. Suppose the robot's state is $\mathbf{x}_t = (x_t, y_t, \theta_t)^\top$ and it senses a landmark whose location is known to be $(x_i, y_i)^\top$. Without noise, the range and bearing can be calculated by simple geometry. (See Figure 25.6(a).) The exact prediction of the observed range and bearing would be

$$\hat{\mathbf{z}}_t = h(\mathbf{x}_t) = \left(\begin{array}{c} \sqrt{(x_t - x_i)^2 + (y_t - y_i)^2} \\ \arctan \frac{y_i - y_t}{x_i - x_t} - \theta_t \end{array} \right)$$

Again, noise distorts our measurements. To keep things simple, one might assume Gaussian noise with covariance Σ_z

$$P(\mathbf{z}_t | \mathbf{x}_t) = N(\hat{\mathbf{z}}_t, \Sigma_z)$$

A somewhat different sensor model is often appropriate for range scanners of the kind shown in Figure 25.2. Such sensors produce a vector of range values $\mathbf{z}_t = (z_1, \dots, z_M)^\top$, each of whose bearings is fixed relative to the robot. Given a pose \mathbf{x}_t , let \hat{z}_j be the exact range along the j th beam direction from \mathbf{x}_t to the nearest obstacle. As before, this will be corrupted by Gaussian noise. Typically, we assume that the errors for the different beam directions are independent and identically distributed, so we have

$$P(\mathbf{z}_t | \mathbf{x}_t) = \prod_{j=1}^M e^{-(z_j - \hat{z}_j)^2 / 2\sigma^2}.$$

Figure 25.6(b) shows an example of a four-beam range scan and two possible robot poses, one of which is reasonably likely to have produced the observed scan and one of which is not.

```

function MONTE-CARLO- LOCALIZATION( $\wedge, z, N, model, map$ ) returns a set of samples
  inputs:  $a$ , the previous robot motion command
     $\mathbf{z}$ , a range scan with  $\mathbf{M}$  readings  $z_1, \dots, z_M$ 
     $N$ , the number of samples to be maintained
     $model$ , a probabilistic environment model with pose prior  $\mathbf{P}(\mathbf{X}_0)$ ,
      motion model  $\mathbf{P}(\mathbf{X}_1|\mathbf{X}_0, A_0)$ , and range sensor noise model  $P(Z|\hat{Z})$ 
     $map$ , a 2D map of the environment
  static:  $S$ , a vector of samples of size  $N$ , initially generated from  $\mathbf{P}(\mathbf{X}_0)$ 
  local variables:  $W$ , a vector of weights of size  $N$ 

  for  $i = 1$  to  $N$  do
     $S[i] \leftarrow$  sample from  $\mathbf{P}(\mathbf{X}_1|\mathbf{X}_0 = S[i], A_0 = a)$ 
     $W[i] \leftarrow 1$ 
    for  $j = 1$  to  $M$  do
       $\hat{z} \leftarrow$  EXACT-RANGE( $j, S[i], map$ )
       $W[i] \leftarrow W[i] \cdot P(Z = z_j | \hat{Z} = \hat{z})$ 
     $S \leftarrow$  WEIGHTED-SAMPLE-WITH-REPLACEMENT( $N, S, W$ )
  return  $S$ 

```

Figure 25.7 A Monte Carlo localization algorithm using a range scan sensor model with independent noise.

Comparing the range scan model to the landmark model, we see that the range scan model has the advantage that there is no need to *identify* a landmark before the range scan can be interpreted; indeed, in Figure 25.6(b), the robot faces a featureless wall. On the other hand, if there *is* a visible, identifiable landmark, it can provide immediate localization.

Chapter 15 described the Kalman filter, which represents the belief state as a single multivariate Gaussian, and the particle filter, which represents the belief state by a collection of particles that correspond to states. Most modern localization algorithms use one of two representations of the robot's belief $\mathbf{P}(\mathbf{X}_t | \mathbf{z}_{1:t}, a_{1:t-1})$.

Localization using particle filtering is called **Monte Carlo localization**, or MCL. The MCL algorithm is essentially identical to the particle-filtering algorithm of Figure 15.15. All we need to do is supply the appropriate motion model and sensor model. Figure 25.7 shows one version using the range scan model. The operation of the algorithm is illustrated in Figure 25.8 as the robot finds out where it is inside an office building. In the first image, the particles are uniformly distributed based on the prior, indicating global uncertainty about the robot's position. In the second image, the first set of measurements arrives and the particles form clusters in the areas of high posterior belief. In the third, enough measurements are available to push all the particles to a single location.

The Kalman filter is the other major way to localize. A Kalman filter represents the posterior $\mathbf{P}(\mathbf{X}_t | \mathbf{z}_{1:t}, a_{1:t-1})$ by a Gaussian. The mean of this Gaussian will be denoted μ_t and its covariance Σ_t . The main problem with Gaussian beliefs is that they are only closed under linear motion models f and linear measurement models h . For nonlinear f or h , the result of

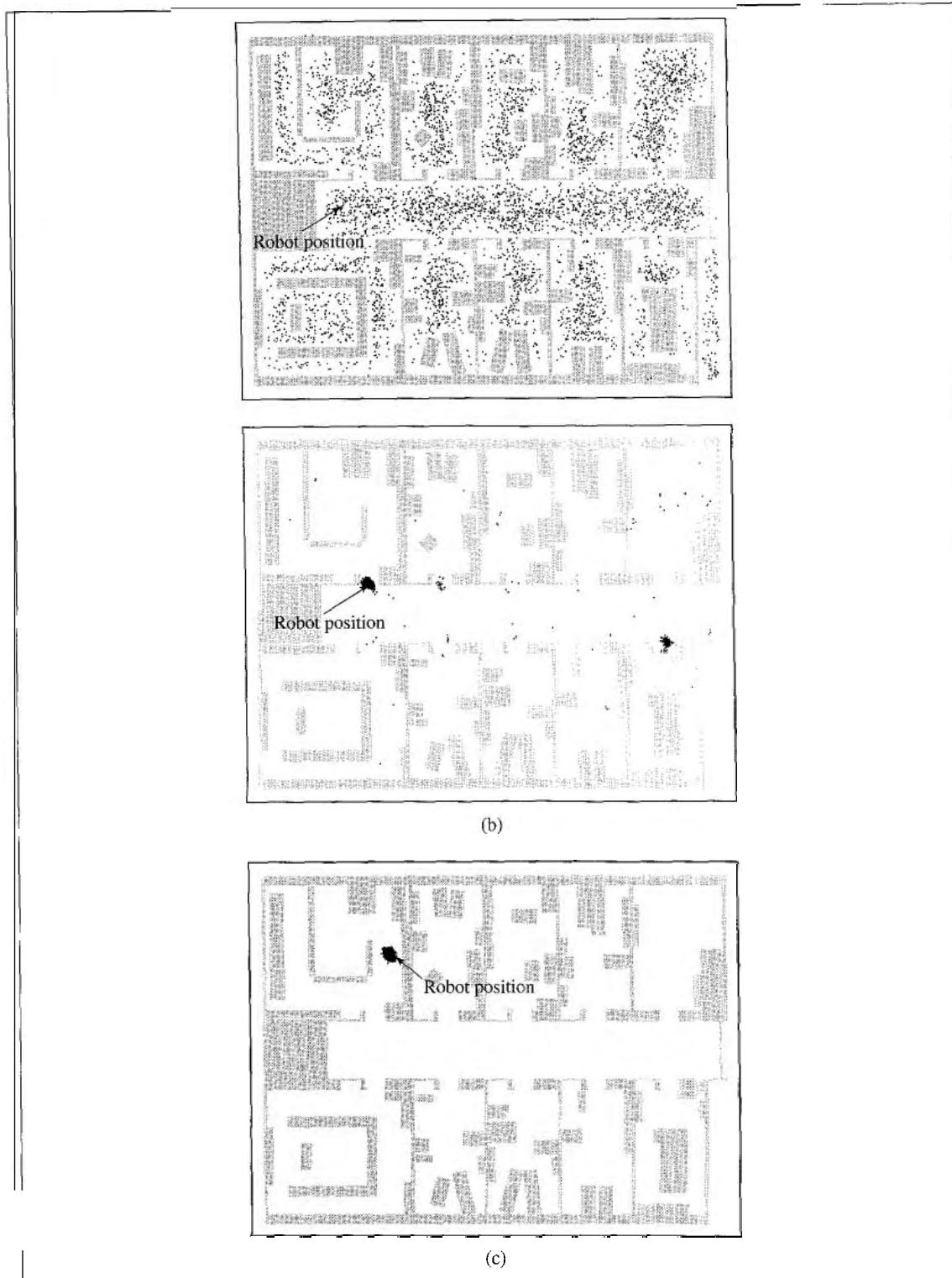


Figure 25.8 Monte Carlo localization, a particle-filter algorithm for mobile robot localization. Top: initial, global uncertainty. Middle: approximately bimodal uncertainty after navigating in the (symmetric) corridor. Bottom: unimodal uncertainty after entering a distinctive office.

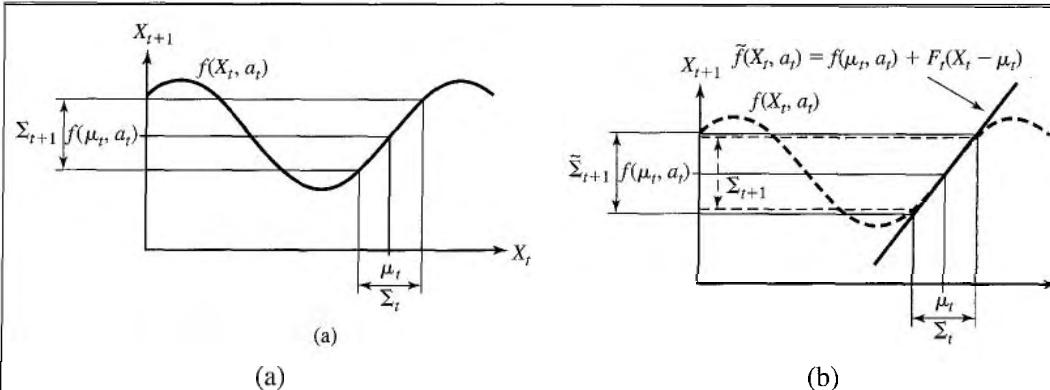


Figure 25.9 One-dimensional illustration of a linearized motion model: (a) The function f , and the projection of a mean μ_t and a covariance interval (based on Σ_t) into time $t+1$. (b) The linearized version is the tangent of f at μ_t . The projection of the mean μ_t is correct. However, the projected covariance $\tilde{\Sigma}_{t+1}$ differs from Σ_{t+1} .

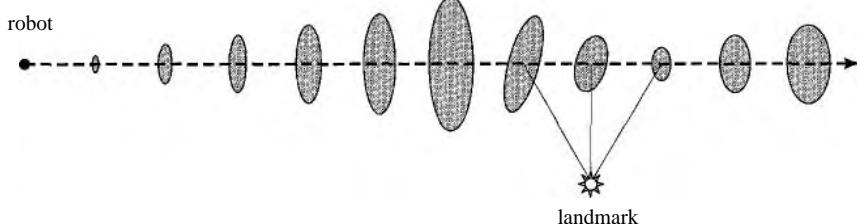


Figure 25.10 Example of localization using the extended Kalman filter. The robot moves on a straight line. As it progresses, its uncertainty increases gradually, as illustrated by the error ellipses. When it observes a landmark with known position, the uncertainty is reduced.

LINEARIZATION

TAYLOR EXPANSION

updating a filter is usually not Gaussian. Thus, localization algorithms using the Kalman filter **linearize** the motion and sensor models. Linearization is a local approximation of a nonlinear function by a linear function. Figure 25.9 illustrates the concept of linearization for a (one-dimensional) robot motion model. On the left, it depicts a nonlinear motion model $f(\mathbf{x}_t, a_t)$ (the control a_t is omitted in this graph since it plays no role in the linearization). On the right, this function is approximated by a linear function $\tilde{f}(\mathbf{x}_t, a_t)$. This linear function is tangent to f at the point μ_t , the mean of our state estimate at time t . Such a linearization is called (first degree) **Taylor expansion**. A Kalman Filter that linearizes f and h via Taylor expansion is called **extended Kalman filter** (or EKF). Figure 25.10 shows a sequence of estimates of a robot running an extended Kalman filter localization algorithm. As the robot moves, the uncertainty in its location estimate increases, as shown by the error ellipses. Its error decreases as it senses the range and bearing to a landmark with known location. The error finally increases again as the robot loses sight of the landmark. EKF algorithms work well if landmarks are easily identified. Otherwise, the posterior distribution may be multimodal, as in Figure 25.8(b). The problem of needing to know the identity of landmarks is an instance of the **data association** problem discussed at the end of Chapter 15.

Mapping

So far, we discussed the problem of localizing a single object. In robotics, one often seeks to localize many objects. The classical example of such a problem is that of robotic mapping. Imagine a robot that is not given a map of its environment. Rather, it has to construct such a map by itself. Clearly, humankind has developed amazing skills in mapping places as big as our entire planet. So a natural problem in robotics is to devise algorithms that enable robots to do the same.

SIMULTANEOUS
LOCALIZATION AND
MAPPING

In the literature, the robot mapping problem is often referred to as ***simultaneous localization and mapping***, abbreviated as **SLAM**. Not only must the robot construct a map, it must do so *without knowing where it is*. SLAM is one of the core problems in robotics. We will consider the version in which the environment is fixed. This is quite difficult enough; it gets much worse when the environment is allowed to change while the robot moves around.

From a statistical perspective, mapping is a Bayesian inference problem, just like localization. If we denote the map by M and the robot pose at time t by \mathbf{X}_t as before, we can rewrite Equation (25.1) to include the entire map in the posterior:

$$\begin{aligned} & \mathbf{P}(\mathbf{X}_{t+1}, M \mid \mathbf{z}_{1:t+1}, a_{1:t}) \\ &= \alpha \mathbf{P}(\mathbf{z}_{t+1} \mid \mathbf{X}_{t+1}, M) \int \mathbf{P}(\mathbf{X}_{t+1} \mid \mathbf{x}_t, a_t) P(\mathbf{x}_t, M \mid \mathbf{z}_{1:t}, a_{1:t-1}) d\mathbf{x}_t. \end{aligned}$$

This equation actually conveys some good news: the conditional distributions needed for incorporating actions and measurements are essentially the same as in the robot localization problem. The main caveat is that the new state space—the space of all robot poses and all maps—has many more dimensions. Just imagine you want to represent an entire building in a photo-realistic way. This will probably require hundreds of millions of numbers. Each number will be a random variable and contributes to the enormously high dimension of the state space. What makes this problem even harder is that fact that the robot may not even know in advance how large its environment is. Thus, the dimensionality of M has to be adjusted dynamically during mapping.

Probably the most widely used method for the SLAM problem is the EKF. It is usually combined with a landmark sensing model and requires that the landmarks are all distinguishable. In the previous section, the posterior estimate was represented by a Gaussian with mean $\boldsymbol{\mu}_t$ and covariance $\boldsymbol{\Sigma}_t$. In the EKF approach to the SLAM problem, the posterior is again Gaussian, but now the mean $\boldsymbol{\mu}_t$ is a much larger vector. It comprises not only the robot pose but also the location of all features (or landmarks) in the map. If there are n such features, this vector will be of dimension $2n + 3$ (it takes two values to specify a landmark location and three to specify the robot pose). Consequently, the matrix $\boldsymbol{\Sigma}_t$ is of dimension $(2n + 3)$ by $(2n + 3)$. It possesses the following structure:

$$\boldsymbol{\Sigma}_t = \begin{pmatrix} \boldsymbol{\Sigma}_{XX} & \boldsymbol{\Sigma}_{XM} \\ \boldsymbol{\Sigma}_{XM}^\top & \boldsymbol{\Sigma}_{MM} \end{pmatrix}. \quad (25.2)$$

Here $\boldsymbol{\Sigma}_{XX}$ is the robot pose covariance, which we already encountered in the context of localization. $\boldsymbol{\Sigma}_{XM}$ is a matrix of size 3 by $2n$ that expresses the correlation between features in the map and robot coordinates. Finally, $\boldsymbol{\Sigma}_{MM}$ is a matrix of size $2n$ by $2n$ that specifies

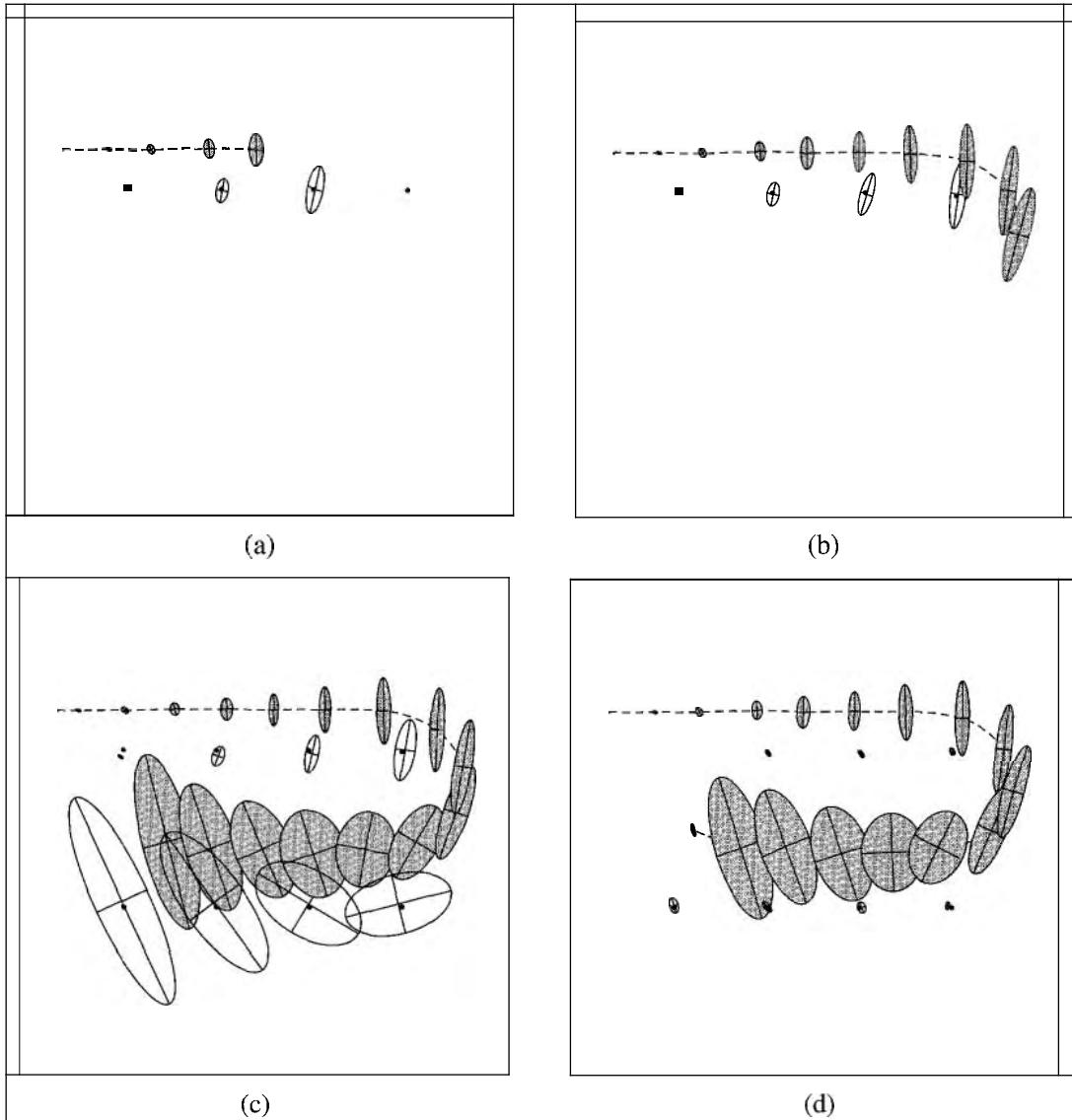


Figure 25.11 EKF applied to the robot mapping problem. The robot's path is a dotted line, and its estimations of its own position are shaded ellipses. Eight distinguishable landmarks of unknown location are shown as small dots, and their location estimations are shown as white ellipses. In (a)–(c) the robot's positional uncertainty is increasing, as is its uncertainty about the landmarks it encounters, stages during which the robot encounters new landmarks, which are mapped with increasing uncertainty. In (d) the robot senses the first landmark again, and the uncertainty of *all* landmarks decreases, thanks to the fact that the estimates are correlated.

the covariance of features in the map, including all pairwise correlations. The memory requirements for EKFs are therefore quadratic in n , the number of features in the map, and the updating time is also quadratic in n .

Before delving into mathematical detail, let us study the EKF solution graphically. Fig-

Figure 25.11 shows a robot in an environment with eight landmarks, arranged in two rows of four landmarks each. Initially, the robot has no idea where the landmarks are located. We will suppose that each landmark is a different color, and the robot can reliably detect which is which. The robot starts at a well-defined location towards the left, but it gradually loses certainty as to where it is. This is indicated by the error ellipses in Figure 25.11(a), whose width increases as the robot moves forward. As the robot moves, it senses the range and bearing to nearby landmarks, and these observations lead to estimates of the location of these landmarks. Naturally, the uncertainty in these landmark estimation is closely tied to the uncertainty in the robot's localization. Figures 25.11(b) and (c) illustrate the robot's belief as it moves further and further through its environment.

An important detail of all these estimates—which is not at all obvious from our graphical depiction—is the fact that we are maintaining a single Gaussian over all estimates. The error ellipses in Figure 25.11 are merely projections of this Gaussian into the subspaces of robot and landmark coordinates. This multivariate Gaussian posterior maintains correlations between all estimates. This observation becomes important for understanding what happens in Figure 25.11(d). Here the robot observes a previously mapped landmark. As a result, its own uncertainty shrinks tremendously. So does the uncertainty of all other landmarks. This is a consequence of the fact that the robot estimate and the landmark estimates are highly correlated in the posterior Gaussian. Finding out knowledge about one variable (here the robot pose) automatically reduces the uncertainty in all others.

The EKF algorithm for mapping resembles the EKF localization algorithm in the previous section. The key difference here arises from the added landmark variables in the posterior. The motion model for landmarks is trivial: they don't move. The function f , therefore, is the identity function for those variables. The measurement function is essentially the same as before. The only difference is that the Jacobian H_t in the EKF update equation is taken not only with respect to the robot pose, but also with respect to the landmark location that was observed at time t . The resulting EKF equations are even more horrifying than the ones stated before, for which reason we will simply omit them here.

However, there is another difficulty that we have silently ignored so far: the fact that the size of the map M is not known in advance. Hence, the number of elements in the final estimate μ_t and Σ_t is unknown as well. It has to be determined dynamically as the robot discovers new landmarks. The solution to this problem is quite straightforward: as the robot discovers a new landmark, it simply adds a new element to the posterior. By initializing the variance of this new element to a very large value, the resulting posterior is the same as if the robot had known of the existence of the landmark in advance.

Other types of perception

Not all of robot perception is about localization and mapping. Robots also perceive the temperature, odors, acoustic signals, and so on. Many of these quantities can be estimated probabilistically, just as in localization and mapping. All that is required for such estimators are conditional probability distributions that characterize the evolution of state variables over time, and other distributions that describe the relation of measurements to state variables.

However, not all working perception systems in robotics rely on probabilistic representations. In fact, while the internal state in all our examples had a clear physical interpretation, this does not necessarily have to be the case. For example, picture a legged robot that attempts to lift a leg over an obstacle. Suppose this robot uses a rule by which it initially moves the leg at a low height, but then lifts it higher and higher if the previous height resulted in a collision with this obstacle. Would we say that the commanded leg height is a representation of some physical quantity in the world? Maybe, in that it relates to the height and steepness of an obstacle. However, we can also think of the leg height as an auxiliary variable of the robot controller, devoid of direct physical meaning. Such representations are not uncommon in robotics, and for certain problems they work well.

The trend in robotics is clearly towards representations with well-defined semantics. Probabilistic techniques outperform other approaches in many hard perceptual problems such as localization and mapping. However, statistical techniques are sometimes too cumbersome, and simpler solutions may be just as effective in practice. To help decide which approach to take, experience working with real physical robots is the best teacher.

25.4 PLANNING TO MOVE

POINT-TO-POINT MOTION

In robotics, decisions ultimately involve motion of effectors. The **point-to-point motion** problem is to deliver the robot or its end-effector to a designated target location. A greater challenge is the **compliant motion** problem, in which a robot moves while being in physical contact with an obstacle. An example of compliant motion is a robot manipulator that screws in a light bulb, or a robot that pushes a box across a table top.

COMPLIANT MOTION

PATH PLANNING

We begin by finding a suitable representation in which motion planning problems can be described and solved. It turns out that the **configuration space**—the space of robot states defined by location, orientation, and joint angles—is a better place to work than the original 3D space. The **path planning** problem is to find a path from one configuration to another in configuration space. We have already encountered various versions of the path planning problem throughout this book; in robotics, the primary characteristic of path planning is that it involves *continuous* spaces. The literature on robot path planning distinguishes a range of different techniques specifically aimed at finding paths in high-dimensional continuous spaces. The major families of approaches are known as **cell decomposition** and **skeletonization**. Each reduces the continuous path-planning problem to a discrete graph search problem by identifying some canonical states and paths within the free space. Throughout this section, we assume that motion is deterministic and that localization of the robot is exact. Subsequent sections will relax these assumptions.

Configuration space

The first step towards a solution to the robot motion problem is to devise an appropriate problem representation. We will start with a simple representation for a simple problem. Consider the robot arm shown in Figure 25.12(a). It has two joints that move independently.

WORKSPACE

LINKAGE
CONSTRAINTSCONFIGURATION
SPACEINVERSE
KINEMATICS

Moving the joints alters the (x, y) coordinates of the elbow and the gripper. (The arm cannot move in the z direction.) This suggests that the robot's configuration can be described by a four-dimensional coordinate: (x_e, y_e) for the location of the elbow relative to the environment and (x_g, y_g) for the location of the gripper. Clearly, these four coordinates characterize the full state of the robot. They constitute what is known as workspace representation, since the coordinates of the robot are specified in the same coordinate system as the objects it seeks to manipulate (or to avoid). Workspace representations are well-suited for collision checking, especially if the robot and all objects are represented by simple polygonal models.

The problem with the workspace representation is that not all workspace coordinates are actually attainable, even in the absence of obstacles. This is because of the **linkage constraints** on the space of attainable workspace coordinates. For example, the elbow position (x_e, y_e) and the gripper position (x_g, y_g) are always a fixed distance apart, because they are joined by a rigid forearm. A robot motion planner defined over workspace coordinates faces the challenge of generating paths that adhere to these constraints. This is particularly tricky because the state space is continuous and the constraints are nonlinear.

It turns out to be easier to plan with a configuration space representation. Instead of representing the state of the robot by the Cartesian coordinates of its elements, we represent the state by a configuration of the robot's joints. Our example robot possesses two joints. Hence, we can represent its state with the two angles φ_s and φ_e for the shoulder joint and elbow joint, respectively. In the absence of any obstacles, a robot could freely take on any value in configuration space. In particular, when planning a path one could simply connect the present and the target configuration by a straight line. In following this path, the robot would then change its joints at a constant velocity, until a target location is reached.

Unfortunately, configuration spaces have their own problems. The task of a robot is usually expressed in workspace coordinates, not in configuration space coordinates. For example, we might want a robot to move its end-effector to a certain coordinate in workspace, possibly with a specification of its orientation as well. This raises the question as to how to map such workspace coordinates into configuration space. In general the inverse of this problem, transforming configuration space coordinates into workspace coordinates, is simple: it involves a series of quite obvious coordinate transformations. These transformations are linear for prismatic joints and trigonometric for revolute joints. This chain of coordinate transformation is known as kinematics, a term we already encountered when discussing mobile robots.

The inverse problem of calculating the configuration of a robot whose effector location is specified in workspace coordinates is known as inverse kinematics. Calculating the inverse kinematics is generally hard, especially for robots with many DOFs. In particular, the solution is seldom unique. For our example robot arm, there are two distinct configurations for which the gripper takes on the same workspace coordinates as in the figure.

In general, this two-link robot arm has between zero and two inverse kinematic solutions for any set of workspace coordinates. Most industrial robots have infinitely many solutions. To see how this is possible, simply imagine we added a third revolute joint to our example robot, one whose rotational axis is parallel to the ones of the existing joint. In such a case, we can keep the location (but not the orientation!) of the gripper fixed and still freely rotate its

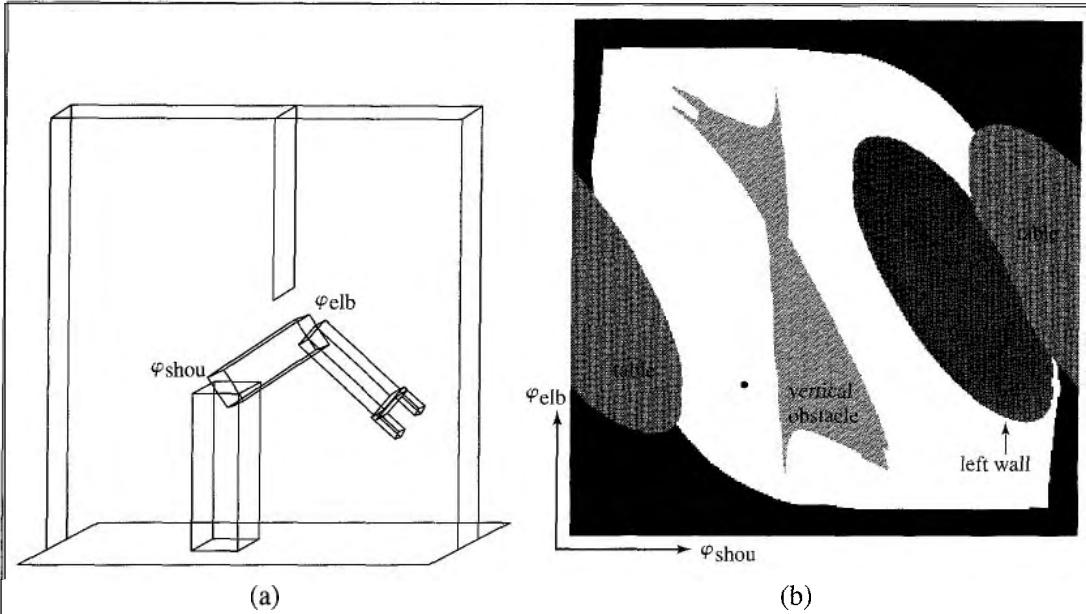


Figure 25.12 (a) Workspace representation of a robot arm with 2 DOFs. The workspace is a box with a flat obstacle hanging from the ceiling. (b) Configuration space of the same robot. Only white regions in the space are configurations that are free of collisions. The dot in this diagram corresponds to the configuration of the robot shown on the left.

internal joints, for most configurations of the robot. With a few more joints (how many?) we can achieve the same effect while keeping the orientation constant as well. We have already seen an example of this in the "experiment" of placing your hand on the desk and moving your elbow. The kinematic constraint of your hand position is insufficient to determine the configuration of your elbow. In other words, the inverse kinematics of your shoulder-arm assembly possesses an infinite number of solutions.

The second problem with configuration space representations arises from the obstacles that may exist in the robot's workspace. Our example in Figure 25.12(a) shows several such obstacles, including a free hanging obstacle that protrudes into the center of the robot's workspace. In workspace, such obstacles take on simple geometric forms—especially in most robotics textbooks, which tend to focus on polygonal obstacles. But how do they look in configuration space?

Figure 25.12(b) shows the configuration space for our example robot, under the specific obstacle configuration shown in Figure 25.12(a). The configuration space can be decomposed into two subspaces: the space of all configurations that a robot may attain, commonly called **free space**, and the space of unattainable configurations, called **occupied space**. The white area in Figure 25.12(b) corresponds to the free space. All other regions correspond to occupied space. The different shading of the occupied space corresponds to the different objects in the robot's workspace; the black region surrounding the entire free space corresponds to configurations in which the robot collides with itself. It is easy to see that extreme values of the shoulder or elbow angles cause such a violation. The two oval-shaped regions on both

FREE SPACE

OCCUPIED SPACE

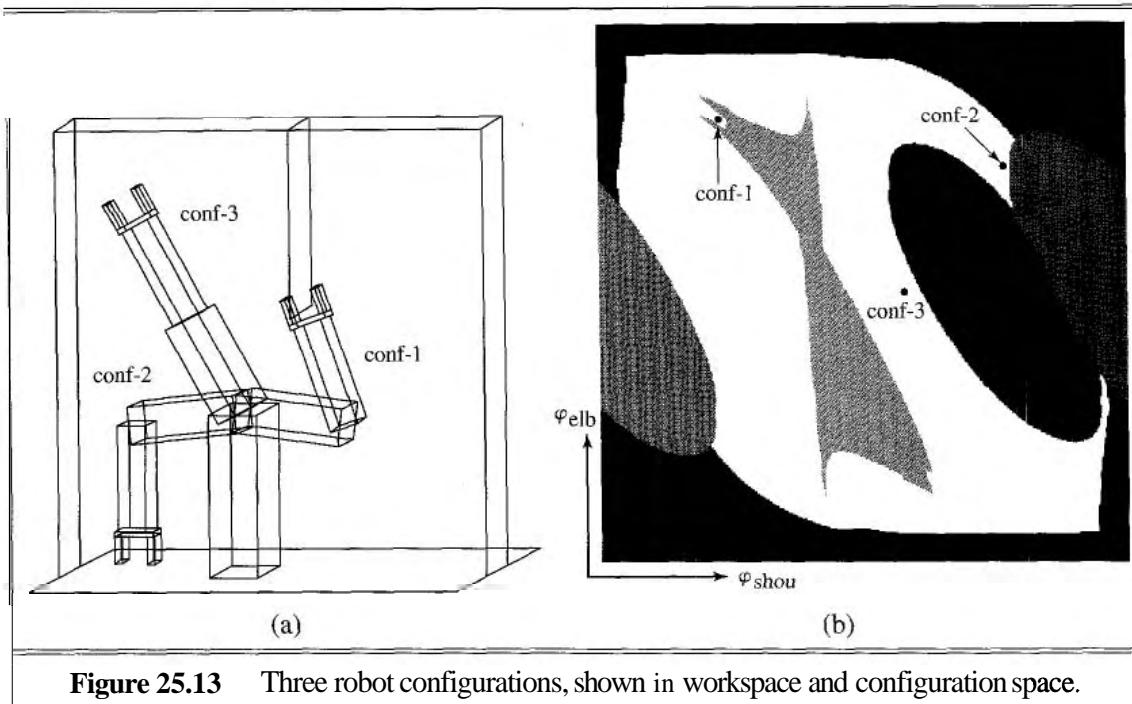


Figure 25.13 Three robot configurations, shown in workspace and configuration space.

sides of the robot correspond to the table on which the robot is mounted. Similarly, the third oval region corresponds to the left wall. Finally, the most interesting object in configuration space is the simple vertical obstacle impeding the robot's workspace. This object has a funny shape: it is highly nonlinear and at places even concave. With a little bit of imagination the reader will recognize the shape of the gripper at the upper left end. We encourage the reader to pause for a moment and study this important diagram. The shape of this obstacle is not at all obvious! The dot inside Figure 25.12(b) marks the configuration of the robot, as shown in Figure 25.12(a). Figure 25.13 depicts three additional configurations, both in workspace and in configuration space. In configuration “conf-1,” the gripper encloses the vertical obstacle.

In general, even if the robot's workspace is represented by flat polygons, the shape of the free space can be very complicated. In practice, therefore, one usually probes a configuration space instead of constructing it explicitly. A planner may generate a configuration and then test to see if it is in free space by applying the robot kinematics and then checking for collisions in workspace coordinates.

Cell decomposition methods

CELL DECOMPOSITION

Our first approach to path planning uses **cell** decomposition—that is, it decomposes the free space into a finite number of contiguous regions, called cells. These regions have the important property that the path planning problem within a single region can be solved by simple means (e.g., moving along a straight line). The path planning problem then becomes a discrete graph search problem, very much like the search problems introduced in Chapter 3.

The simplest cell decomposition consists of a regularly spaced grid. Figure 25.14(a) shows a square grid decomposition of the space and a solution path that is optimal for this

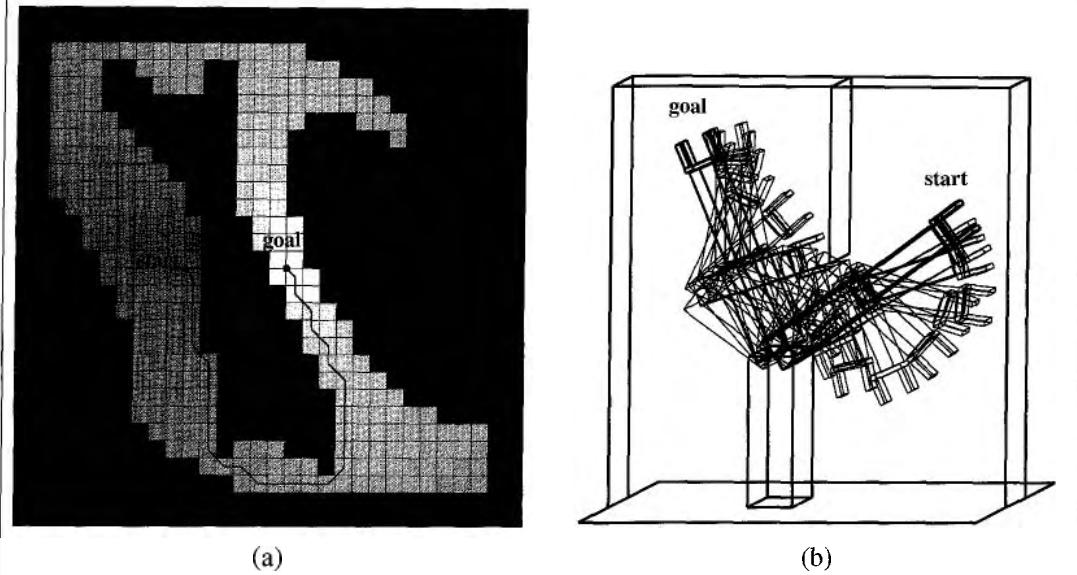


Figure 25.14 (a) Value function and path found for a discrete grid cell approximation of the configuration space. (b) The same path visualized in workspace coordinates. Notice how the robot bends its elbow to avoid a collision with the vertical obstacle.

grid size. We have also used grayscale shading to indicate the *value* of each free-space grid cell—i.e., the cost of the shortest path from that cell to the goal. (These values can be computed by a deterministic form of the VALUE-ITERATION algorithm given in Figure 17.4.) Figure 25.14(b) shows the corresponding work space trajectory for the arm.

Such a decomposition has the advantage that it is extremely simple to implement, but it also suffers from two limitations. First, it is only workable for low-dimensional configuration spaces, as the number of grid cells increases exponentially with d , the number of dimensions. Second, there is the problem of what to do with cells that are "mixed"—that is, neither entirely within free space nor entirely within occupied space. A solution path that includes such a cell may not be a real solution, because there may be no way to cross the cell in the desired direction in a straight line. This would make the path planner *unsound*. On the other hand, if we insist that only completely free cells may be used, the planner will be *incomplete*, because it might be the case that the only paths to the goal may go through mixed cells—especially if the cell size is comparable to that of the passageways and clearances in the space.

There are two ways to fix the cell decomposition method to avoid these problems. The first is to allow *further subdivision* of the mixed cells—perhaps using cells of half the original size. This can be continued recursively until a path is found that lies entirely within free cells. (Of course, the method only works if there is a way to decide if a given cell is a mixed cell, which is easy only if the configuration space boundaries have relatively simple mathematical descriptions.) This method is complete provided there is a bound on the smallest passageway through which a solution must pass. Although it focuses most of the computational effort on the tricky areas within the configuration space, it still fails to scale well to high-dimensional

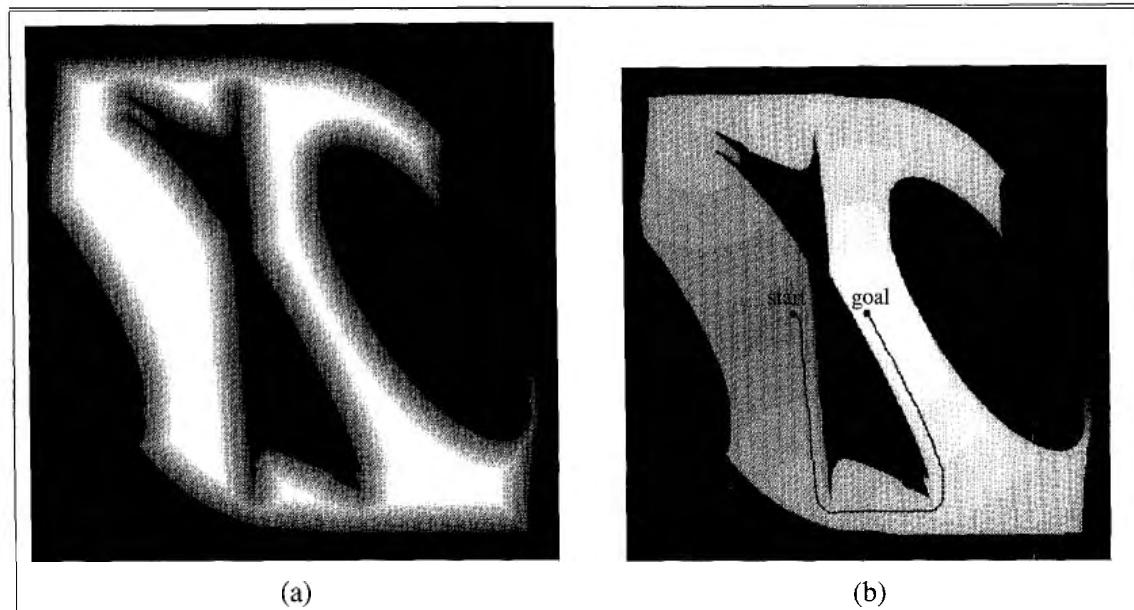


Figure 25.15 (a) A repelling potential field pushes the robot away from obstacles. (b) Path found by simultaneously minimizing path length and the potential.

EXACT CELL DECOMPOSITION

problems because each recursive splitting of a cell creates 2^d smaller cells. A second way to obtain a complete algorithm is to insist on an **exact cell decomposition** of the free space. This method must allow cells to be irregularly shaped where they meet the boundaries of free space, but the shapes must still be "simple" in the sense that it should be easy to compute a traversal of any free cell. This technique requires some quite advanced geometric ideas, so we shall not pursue it further here.

Examining the solution path shown in Figure 25.14(a), we can see additional difficulties that will have to be resolved. First, notice that the path contains arbitrarily sharp corners; a robot moving at any finite speed could not execute such a path. Second, notice that the path goes very close to the obstacle. Anyone who has driven a car knows that a parking lot stall with one millimeter of clearance on either side is not really a parking space at all; for the same reason, we would prefer solution paths that are robust with respect to small motion errors.

POTENTIAL FIELD

We would like to maximize the clearance from obstacles while minimizing the path length. This can be achieved by introducing a **potential field**. A potential field is a function defined over state space, whose value grows with the distance to the closest obstacle. Figure 25.15(a) shows such a potential field—the darker a configuration state, the closer it is to an obstacle. When used in path planning, this potential field becomes an additional cost term in the optimization. This induces an interesting trade-off. On the one hand, the robot seeks to minimize path length to the goal. On the other, it tries to stay away from obstacles by virtue of minimizing the potential function. With the appropriate weight between both objectives, a resulting path may look like the one shown in Figure 25.15(b). This figure also displays the value function derived from the combined cost function, again calculated by value iteration. Clearly, the resulting path is longer, but it is also safer.

Skeletonization methods

SKELETONIZATION

The second major family of path-planning algorithms is based on the idea of **skeletonization**. These algorithms reduce the robot's free space to a one-dimensional representation, for which the planning problem is easier. This lower-dimensional representation is called a **skeleton** of the configuration space.

VORONOI GRAPH

Figure 25.16 shows an example skeletonization: it is a **Voronoi graph** of the free space—the set of all points that are equidistant to two or more obstacles. To do path planning with a Voronoi graph, the robot first changes its present configuration to a point on the Voronoi graph. It is easy to show that this can always be achieved by a straight-line motion in configuration space. Second, the robot follows the Voronoi graph until it reaches the point nearest to the target configuration. Finally, the robot leaves the Voronoi graph and moves to the target. Again, this final step involves straight-line motion in configuration space.

In this way, the original path-planning problem is reduced to finding a path on the Voronoi diagram, which is generally one-dimensional (except in certain non-generic cases) and has finitely many points where three or more one-dimensional curves intersect. Thus, finding the shortest path along the Voronoi graph is a discrete graph search problem of the kind discussed in Chapters 3 and 4. Following the Voronoi graph may not give us the shortest path, but the resulting paths tend to maximize clearance. Disadvantages of Voronoi graph techniques are that they are difficult to apply to higher dimensional configuration spaces, and that they tend to induce unnecessarily large detours when the configuration space is wide open. Furthermore, computing the Voronoi diagram can be difficult, specifically in configuration space, where the shape of obstacles can be complex.

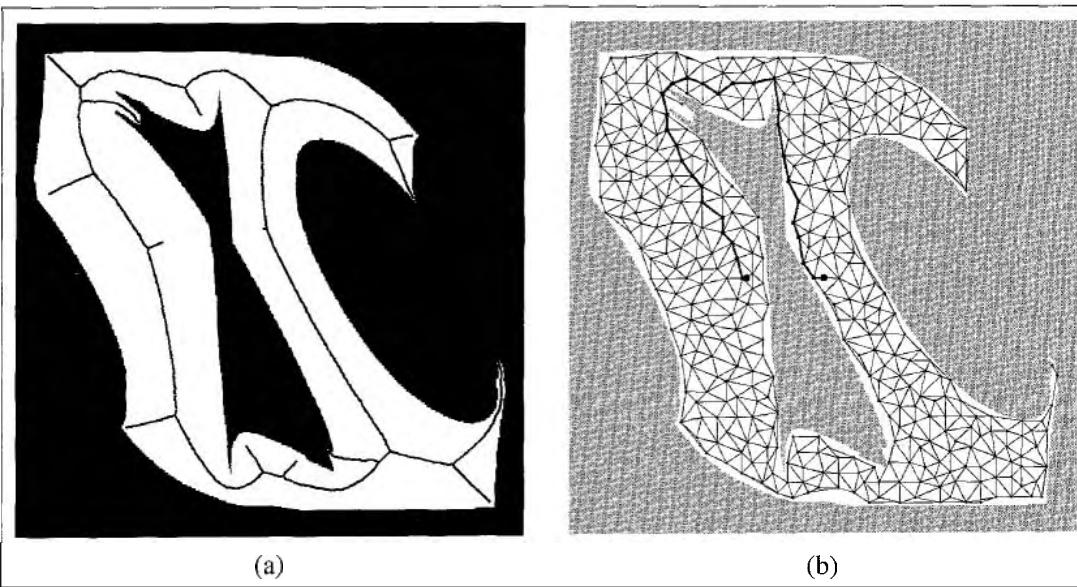


Figure 25.16 (a) The Voronoi diagram is the set of points equidistant to two or more obstacles in configuration space. (b) A probabilistic roadmap, composed of 400 randomly chosen points in free space.

An alternative to the Voronoi diagrams is the **probabilistic roadmap**, a skeletonization approach that offers more possible routes, and thus deals better with wide open spaces. Figure 25.16(b) shows an example of a probabilistic roadmap. The graph is created by randomly generating a large number of configurations, and discarding those that do not fall into free space. Then, we join any two nodes by an arc if it is "easy" to reach one node from the other—for example, by a straight line in free space. The result of all this is a randomized graph in the robot's free space. If we add the robot's start and target configurations to this graph, path planning amounts to a discrete graph search. Theoretically, this approach is incomplete, because a bad choice of random points may leave us without any paths from start to target. It is possible to bound the probability of failure in terms of the number of points generated and certain geometric properties of the configuration space. It is also possible to direct the generation of sample points towards the areas where a partial search suggests that a good path may be found, working bidirectionally from both the start and the goal positions. With these improvements, probabilistic roadmap planning tends to scale better to high-dimensional configuration spaces than most alternative path planning techniques.

25.5 PLANNING UNCERTAIN MOVEMENTS

None of the robot motion planning algorithms discussed thus far addresses a key characteristic of robotics problems: *uncertainty*. In robotics, uncertainty arises from partial observability of the environment and from the stochastic (or unmodeled) effects of the robot's actions. Errors can also arise from the use of approximation algorithms such as particle filtering, which does not provide the robot with an exact belief state even if the stochastic nature of the environment is modeled perfectly.

Most of today's robots use deterministic algorithms for decision making, such as the various path planning algorithms discussed thus far. To do so, it is common practice to extract the **most likely state** from the state distribution produced by the localization algorithm. The advantage of this approach is purely computational. Planning paths through configuration space is already a challenging problem; it would be worse if we had to work with a full probability distribution over states. Ignoring uncertainty in this way works when the uncertainty is small.

Unfortunately, ignoring the uncertainty does not always work. In some problems the robot's uncertainty is simply too large. For example, how can we use a deterministic path planner to control a mobile robot that has no clue where it is? In general, if the robot's true state is not the one identified by the maximum likelihood rule, the resulting control will be suboptimal. Depending on the magnitude of the error this can lead to all sorts of unwanted effects, such as collisions with obstacles.

The field of robotics has adopted a range of techniques for accommodating uncertainty. Some are derived from the algorithms given in Chapter 17 for decision making under uncertainty. If the robot only faces uncertainty in its state transition, but its state is fully observable, the problem is best modeled as a *Markov Decision process*, or *MDP*. The solution of an MDP

is an optimal **policy**, which tells the robot what to do in every possible state. In this way, it can handle all sorts of motion errors, whereas a single-path solution from a deterministic planner would be much less robust. In robotics, policies are usually called **navigation functions**. The value function shown in Figure 25.14(a) can be converted into such a navigation function simply by following the gradient.

Just as in Chapter 17, partial observability makes the problem much harder. The resulting robot control problem is a **partially observable** MDP, or POMDP. In such situations, the robot usually maintains an internal belief state, like the ones discussed in Section 25.3. The solution to a POMDP is a policy defined over the robot's belief state. Put differently, the input to the policy is an entire probability distribution. This enables the robot to base its decision not only on what it knows, but also on what it does not know. For example, if it is uncertain about a critical state variable, it can rationally invoke an **information gathering action**. This is impossible in the MDP framework, since MDPs assume full observability. Unfortunately, techniques that solve POMDPs exactly are inapplicable to robotics—there are no known techniques for continuous spaces. Discretization usually produces POMDPs that are far too large for known techniques to handle. All we can do at present is to try to keep the pose uncertainty to a minimum; for example, the **coastal navigation** heuristic requires the robot to stay near known landmarks to decrease its pose uncertainty. This, in turn, gradually decreases the uncertainty in the mapping of new landmarks that are nearby, which then allows the robot to explore more territory.

Robust methods

Uncertainty can also be handled using so-called **robust** methods rather than probabilistic methods. A robust method is one that assumes a bounded amount of uncertainty in each aspect of a problem, but does not assign probabilities to values within the allowed interval. A robust solution is one that works no matter what actual values occur, provided they are within the assumed interval. An extreme form of robust method is the **conformant planning** approach given in Chapter 12—it produces plans that work with no state information at all.

Here, we look at a robust method that is used for **fine-motion planning** (or FMP) in robotic assembly tasks. Fine motion planning involves moving a robot arm in very close proximity to a static environment object. The main difficulty with fine-motion planning is that the required motions and the relevant features of the environment are very small. At such small scales, the robot is unable to measure or control its position accurately and may also be uncertain of the shape of the environment itself; we will assume that these uncertainties are all bounded. The solutions to FMP problems will typically be conditional plans or policies that make use of sensor feedback during execution and are guaranteed to work in all situations consistent with the assumed uncertainty bounds.

A fine-motion plan consists of a series of **guarded motions**. Each guarded motion consists of (1) a motion command and (2) a termination condition, which is a predicate on the robot's sensor values, and returns true to indicate the end of the guarded move. The motion commands are typically **compliant motions** that allow the robot to slide if the motion command would cause collision with an obstacle. As an example, Figure 25.17 shows a

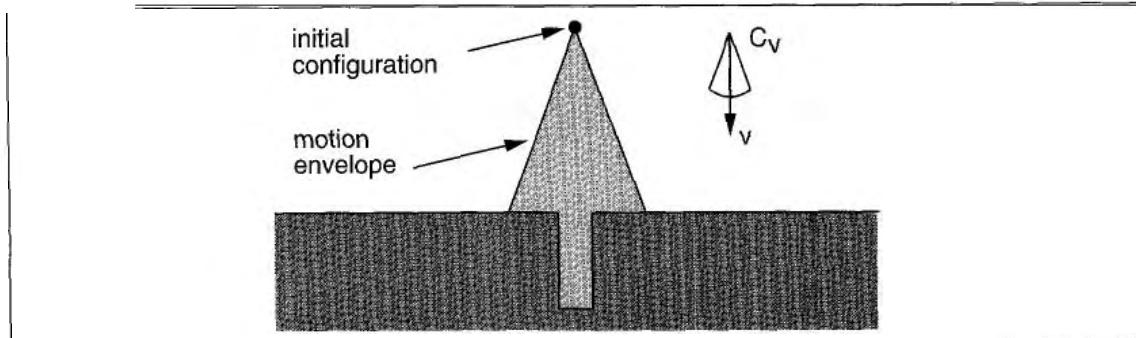


Figure 25.17 A two-dimensional environment, velocity uncertainty cone, and envelope of possible robot motions. The intended velocity is v , but with uncertainty the actual velocity could be anywhere in C_v , resulting in a final configuration somewhere in the motion envelope, which means we wouldn't know if we hit the hole or not.

two-dimensional configuration space with a narrow vertical hole. It could be the configuration space for insertion of a rectangular peg into a hole that is slightly larger. The motion commands are constant velocities. The termination conditions are contact with a surface. To model uncertainty in control, we assume that instead of moving in the commanded direction, the robot's actual motion lies in the cone C_v about it. The figure shows what would happen if we commanded a velocity straight down from the start region s . Because of the uncertainty in velocity, the robot could move anywhere in the conical envelope, possibly going into the hole, but more likely landing to one side of it. Because the robot would not then know which side of the hole it was on, it would not know which way to move.

A more sensible strategy is shown in Figures 25.18 and 25.19. In Figure 25.18, the robot deliberately moves to one side of the hole. The motion command is shown in the figure, and the termination test is contact with any surface. In Figure 25.19, a motion command is given that causes the robot to slide along the surface and into the hole. This assumes we use a compliant motion command. Because all possible velocities in the motion envelope are to the right, the robot will slide to the right whenever it is in contact with a horizontal surface. It will slide down the right-hand vertical edge of the hole when it touches it, because all possible velocities are down relative to a vertical surface. It will keep moving until it reaches the bottom of the hole, because that is its termination condition. In spite of the control uncertainty, all possible trajectories of the robot terminate in contact with the bottom of the hole—that is, unless surface irregularities cause the robot to stick in one place.

As one might imagine, the problem of constructing fine-motion plans is not trivial; in fact, it is a good deal harder than planning with exact motions. One can either choose a fixed number of discrete values for each motion or use the environment geometry to choose directions that give qualitatively different behavior. A fine-motion planner takes as input the configuration-space description, the angle of the velocity uncertainty cone, and a specification of what sensing is possible for termination (surface contact in this case). It should produce a multistep conditional plan or policy that is guaranteed to succeed, if such a plan exists.

Our example assumes that the planner has an exact model of the environment, but it is possible to allow for bounded error in this model as follows. If the error can be described in

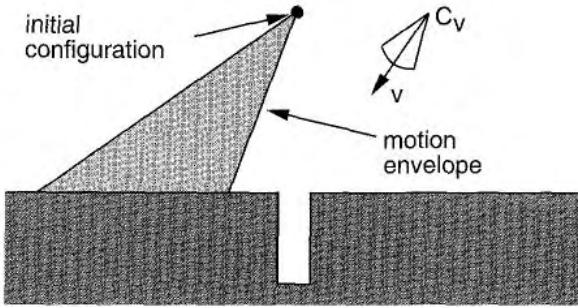


Figure 25.18 The first motion command and the resulting envelope of possible robot motions. No matter what the error, we know the final configuration will be to the left of the hole.

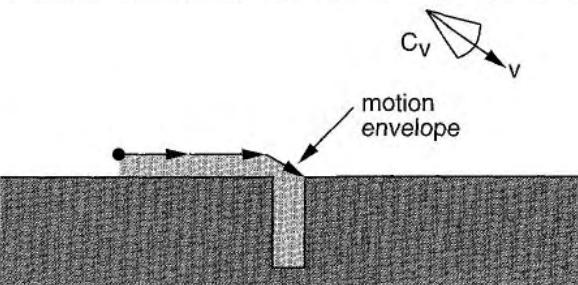


Figure 25.19 The second motion command and the envelope of possible motions. Even with error, we will eventually get into the hole.

terms of parameters, those parameters can be added as degrees of freedom to the configuration space. In the last example, if the depth and width of the hole were uncertain, we could add them as two degrees of freedom to the configuration space. It is impossible to move the robot in these directions in the configuration space or to sense its position directly. But both those restrictions can be incorporated when describing this problem as an FMP problem by appropriately specifying control and sensor uncertainties. This gives a complex, four-dimensional planning problem, but exactly the same planning techniques can be applied. Notice that unlike the decision-theoretic methods in Chapter 17, this kind of robust approach results in plans designed for the worst-case outcome, rather than maximizing the expected quality of the plan. Worst-case plans are only optimal in the decision-theoretic sense if failure during execution is much worse than any of the other costs involved in execution.

25.6 MOVING

So far, we have talked about how to plan motions, but not about how to move. Our plans—particularly those produced by deterministic path planners—assume that the robot can simply follow any path that the algorithm produces. In the real world, of course, this is not the case.

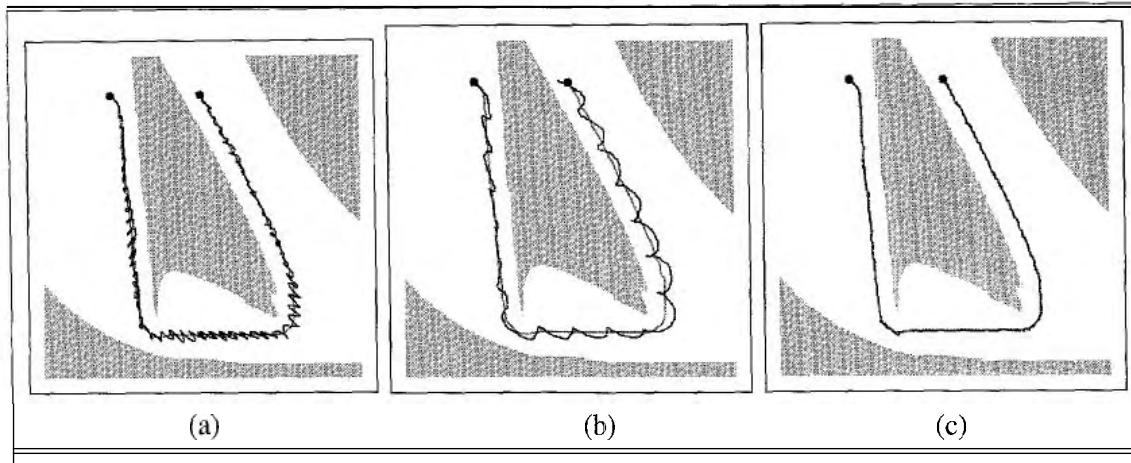


Figure 25.20 Robot arm control using (a) proportional control with gain factor 1.0, (b) proportional control with gain factor 0.1, and (c) PD control with gain factors 0.3 for the proportional and 0.8 for the differential component. In all cases the robot arm tries to follow the path shown in gray.

Robots have inertia and cannot execute arbitrary paths except at arbitrarily slow speeds. In most cases, the robot gets to exert forces rather than specify positions. This section discusses methods for calculating these forces.

Dynamics and control

Section 25.2 introduced the notion of **dynamic state**, which extends the kinematic state of a robot by modeling a robot's velocities. For example, in addition to the angle of a robot joint, the dynamic state also captures the rate of change of the angle. The transition model for a dynamic state representation includes the effect of forces on this rate of change. Such models are typically expressed via **differential equations**, which are equations that relate a quantity (e.g., a kinematic state) to the change of the quantity over time (e.g., velocity). In principle, we could have chosen to plan robot motion using dynamic models, instead of our kinematic models. Such a methodology would lead to superior robot performance, if we could generate the plans. However, the dynamic state is more complex than the kinematic space, and the curse of dimensionality would render motion planning problems intractable for all but the most simple robots. For this reason, practical robot system often rely on simpler kinematic path planners.

A common technique to compensate for the limitations of kinematic plans is to use a separate mechanism, a **controller**, for keeping the robot on track. Controllers are techniques for generating robot controls in real time using feedback from the environment, so as to achieve a control objective. If the objective is to keep the robot on a preplanned path, it is often referred to as a **reference controller** and the path is called a **reference path**. Controllers that optimize a global cost function are known as **optimal controllers**. Optimal policies for MDPs are, in effect, optimal controllers.

On the surface, the problem of keeping a robot on a pre-specified path appears to be relatively straightforward. In practice, however, even this seemingly simple problem has its

DIFFERENTIAL EQUATIONS

CONTROLLER

REFERENCE
CONTROLLER
REFERENCE PATH
OPTIMAL
CONTROLLERS

pitfalls. Figure 25.20(a) illustrates what can go wrong. Shown there is the path of a robot that attempts to follow a kinematic path. Whenever a deviation occurs—whether due to noise or to constraints on the forces the robot can apply—the robot provides an opposing force whose magnitude is proportional to this deviation. Intuitively, this might appear plausible, since deviations should be compensated by a counter-force to keep the robot on track. However, as Figure 25.20(a) illustrates, our controller causes the robot to vibrate rather violently. The vibration is the result of a natural inertia of the robot arm: once driven back to its reference position the robot then overshoots, which induces a symmetric error with opposite sign. As Figure 25.20(a) illustrates, such overshooting may continue along an entire trajectory, and the resulting robot motion is far from desirable. Clearly, there is a need for better control.

To arrive at a better controller, let us formally describe the type of controller that produced the overshooting. Controllers that provide force in negative proportion to the observed error are known as **P controllers**. The letter **P** stands for *proportional*, indicating that the actual control is proportional to the error of the robot manipulator. More formally, let $y(t)$ be the reference path, parameterized by time index t . The control a_t generated by a P controller has the following form:

$$a_t = K_P(y(t) - x_t).$$

GAIN PARAMETER Here x_t is the state of the robot at time t . K_P is a so-called **gain parameter** of the controller that regulates how strongly the controller corrects for deviations between the actual state x_t and the desired one $y(t)$. In our example, $K_P = 1$. At first glance, one might think that choosing a smaller value for K_P remedies the problem. Unfortunately, this is not the case. Figure 25.20(b) shows a trajectory for $K_P = .1$, still exhibiting oscillatory behavior. Lower values of the gain parameter may simply slow down the oscillation, but do not solve the problem. In fact, in the absence of friction, the P controller is essentially a spring law; so it will oscillate indefinitely around a fixed target location.

Traditionally, problems of this type fall into the realm of **control theory**, a field of increasing importance to researchers in AI. Decades of research in this field have led to a large number of controllers that are superior to the simple control law given above. In particular, a reference controller is said to be **stable** if small perturbations lead to a bounded error between the robot and the reference signal. It is said to be **strictly stable** if it is able to return to its reference path upon such perturbations. Clearly, our P controller appears to be stable but not strictly stable, since it fails to return to its reference trajectory.

The simplest controller that achieves strict stability in our domain is known as a **PD controller**. The letter 'P' stands again for *proportional*, and 'D' stands for *derivative*. PD controllers are described by the following equation:

$$a_t = K_P(y(t) - x_t) + K_D \frac{\partial(y(t) - x_t)}{\partial t} \quad (25.3)$$

As this equation suggests, PD controllers extend P controllers by a differential component, which adds to the value of a_t a term that is proportional to the first derivative of the error $y(t) - x_t$ over time. What is the effect of such a term? In general, a derivative term dampens the system that is being controlled. To see, consider a situation where the error $(y(t) - x_t)$ is changing rapidly over time, as is the case for our P controller above. The derivative of this

STABLE

STRICTLY STABLE

PDCONTROLLER

error will then counteract the proportional term, which will reduce the overall response to the perturbation. However, if same error persists and does not change, the derivative will vanish and the proportional term dominates the choice of control.

Figure 25.20(c) shows the result of applying this PD controller to our robot arm, using as gain parameters $K_P = .3$ and $K_D = .8$. Clearly, the resulting path is much smoother, and does not exhibit any obvious oscillations. As this example suggests, a differential term can make a controller stable that otherwise is not.

In practice, PD controllers also possess failure modes. In particular, PD controllers may fail to regulate an error down to zero, even in the absence of external perturbations. This is not obvious from our robot example, but sometimes an over-proportional feedback is required to drive an error down to zero. The solution to this problem lies in adding a third term to the control law, based on the integrated error over time:

$$a_t = K_P(y(t) - x_t) + K_I \int (y(t) - x_t) dt + K_D \frac{\partial(y(t) - x_t)}{\partial t} \quad (25.4)$$

Here K_I is yet another gain parameter. The term $\int (y(t) - x_t) dt$ calculates the integral of the error over time. The effect of this term is that long-lasting deviations between the reference signal and the actual state are corrected. If, for example, x_t is smaller than $y(t)$ for a long period of time, this integral will grow until the resulting control a_t forces this error to shrink. Integral terms, then, ensure that a controller does not exhibit systematic error, at the expenses of increased danger of oscillatory behavior. A controller with all three terms is called a **PID controller**. PID controllers are widely used in industry, for a variety of control problems.

PIDCONTROLLER

Potential field control

We introduced potential fields as an additional cost function in robot motion planning, but they can also be used for generating robot motion directly, dispensing with the path planning phase altogether. To achieve this, we have to define an attractive force that pulls the robot towards its goal configuration and a repellent potential field that pushes the robot away from obstacles. Such a potential field is shown in Figure 25.21. Its single global minimum is the target configuration, and the value is the sum of the distance to this target configuration and the proximity to obstacles. No planning was involved in generating the potential field shown in the figure. Because of this, potential fields are well-suited to real-time control. Figure 25.21 shows two trajectories of a robot that performs hill climbing in the potential field, under two different initial configurations. In many applications, the potential field can be calculated efficiently for any given configuration. Moreover, optimizing the potential amounts to calculating the gradient of the potential for the present robot configuration. These calculations are usually extremely efficient, especially when compared to path planning algorithms, all of which are exponential in the dimensionality of the configuration space (the DOFs).

The fact that the potential field approach manages to find a path to the goal in such an efficient manner, even over long distances in configuration space, raises the question as to whether there is a need for planning in robotics at all. Are potential field techniques sufficient, or were we just lucky in our example? The answer is that we were indeed lucky. Potential fields have many local minima that can trap the robot. In this example, the robot approaches

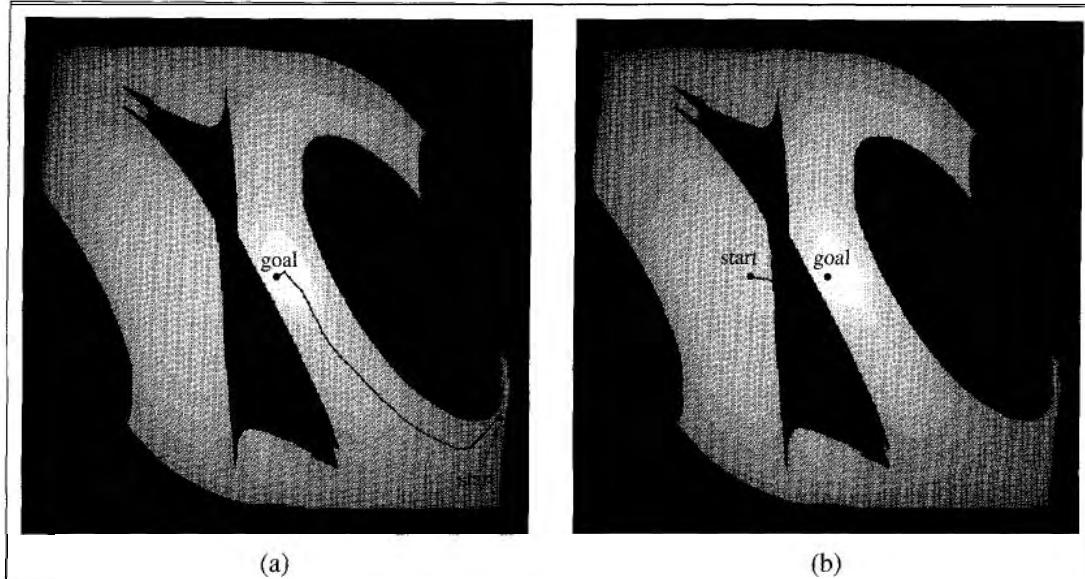


Figure 25.21 Potential field control. The robot ascends a potential field composed of repelling forces asserted from the obstacles, and an attracting force that corresponds to the target configuration. (a) Successful path. (b) Local optimum.

the obstacle by simply rotating its shoulder joint, until it gets stuck on the wrong side of the obstacle. The potential field is not rich enough to make the robot bend its elbow so that the arm fits under the obstacle. In other words, potential field techniques are great for local robot control but they still require global planning. Another important drawback with potential fields is that the forces they generate depend only on the obstacle and robot positions, not on the robot's velocity. Thus, potential field control is really a kinematic method and may fail if the robot is moving quickly.

Reactive control

So far we have consider control decisions that require some model of the environment for constructing either a reference path or a potential field. There are some difficulties with this approach. First, models that are sufficiently accurate are often difficult to obtain, especially in complex or remote environments, such as the surface of Mars. Second, even in cases where we can devise a model with sufficient accuracy, computational difficulties and localization error might render these techniques impractical. In some cases, a reflex agent design—so-called **reactive** control—is more appropriate.

REACTIVE CONTROL
HEXAPOD

One such example is the six-legged robot, or **hexapod** shown in Figure 25.22(a), with the task of walking through rough terrain. The robot's sensors are grossly inadequate to obtain models of the terrain at sufficient accuracy for any of the path planning techniques described in the previous section to work. Moreover, even if we added sufficiently accurate sensors, the twelve degrees of freedom (two for each leg) would render the resulting path planning problem computationally intractable.

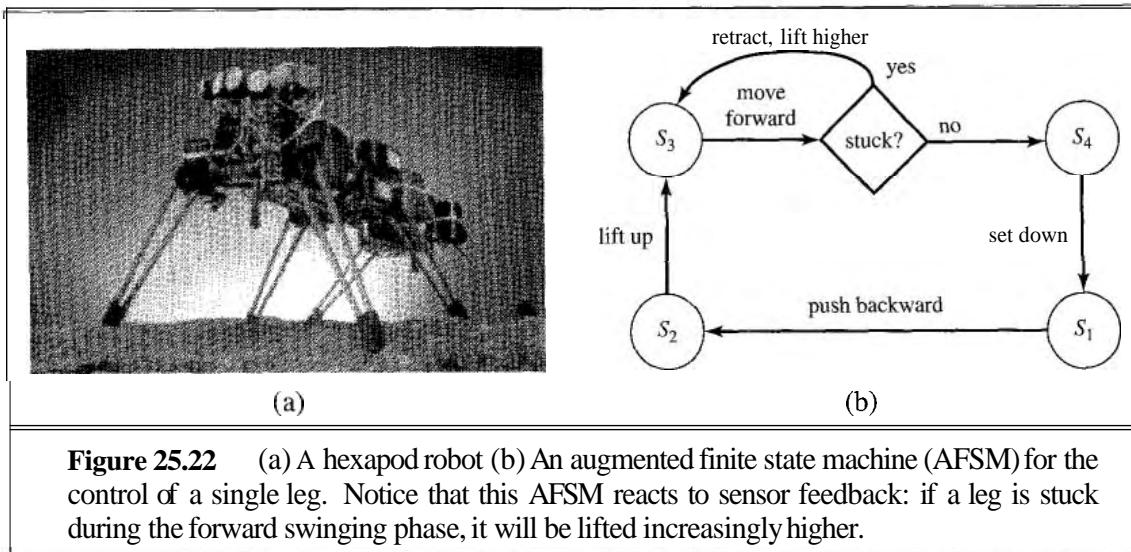


Figure 25.22 (a) A hexapod robot (b) An augmented finite state machine (AFSM) for the control of a single leg. Notice that this AFSM reacts to sensor feedback: if a leg is stuck during the forward swinging phase, it will be lifted increasingly higher.

It is possible, nonetheless, to specify a controller directly without an explicit environmental model. (We have already seen this with the PD controller, which was able to keep a complex robot arm on target *without* an explicit model of the robot dynamics; it did, however, require a reference path generated from a kinematic model.) For the example of our legged robot, specifying a control law turns out to be surprisingly simple at the right level of abstraction. A viable control law might make each leg move in cycles, so that for some of the time it touches the ground, and for the remaining time it moves in the air. All six legs should be coordinated so that three of them (on opposite ends) are always on the ground to provide physical support. Such a control pattern is easily programmed and works great on flat terrain. On rugged terrain, obstacles may prevent legs from swinging forward. This problem can be overcome by a remarkably simple control rule: *when a leg's forward motion is blocked, simply retract it, lift it higher; and try again*. The resulting controller is shown in Figure 25.22(b) as a finite state machine; it constitutes a reflex agent with state, where the internal state is represented by the index of the current machine state (s_1 through s_4).

Variants of this simple feedback-driven controller has been found to generate a remarkably robust walking pattern, capable of maneuvering the robot over rugged terrain. Clearly, such a controller is model-free, and it does not deliberate or use search for generating controls. When executing such a controller, environment feedback plays a crucial role in the behavior generated by the robot. The software alone does not specify what will actually happen when the robot is placed in an environment. Behavior that emerges through the interplay of a (simple) controller and a (complex) environment is often referred to as **emergent behavior**. Strictly speaking, all robots discussed in this chapter exhibit emergent behavior, due to the fact that no model is perfect. Historically, however, the term has been reserved for control techniques that do not utilize explicit environmental models. Emergent behavior is also characteristic of a great number of biological organisms.

Technically, reactive controllers are just one implementation of a policy for an MDP (or, if they have internal state, for a POMDP). In Chapter 17, we encountered several tech-

niques for generating policies from models of the robot and its environment. In robotics, crafting such policies by hand is of great practical importance, due to our inability to formulate accurate models. Chapter 21 described reinforcement learning methods for constructing policies from experience. Some of those methods—such as Q-learning and the policy search methods—require no model of the environment and are capable of generating high-quality controllers for robots, but instead rely on vast amounts of training data.

25.7 ROBOTIC SOFTWARE ARCHITECTURES

SOFTWARE ARCHITECTURE

A methodology for structuring algorithms is called a **software architecture**. An architecture usually includes languages and tools for writing programs, as well as an overall philosophy for how programs can be brought together.

Modern-day software architectures for robotics must decide how to combine reactive control and model-based deliberative control. In many ways, reactive and deliberate control have orthogonal strengths and weaknesses. Reactive control is sensor-driven and appropriate for making low-level decisions in real time. However, reactive control rarely yields a plausible solution at the global level, because global control decisions depend on information that cannot be sensed at the time of decision making. For such problems, deliberate control is more appropriate.

Consequently, most robot architectures use reactive techniques at the lower levels of control with deliberate techniques at the higher levels. We encountered such a combination in our discussion of PD controllers, where we combined a (reactive) PD controller with a (deliberate) path planner. Architectures that combine reactive and deliberate techniques are usually called **hybrid architectures**.

HYBRID ARCHITECTURES

Subsumption architecture

SUBSUMPTION ARCHITECTURE

The **subsumption architecture** (Brooks, 1986) is a framework for assembling reactive controllers out of finite state machines. Nodes in these machines may contain tests for certain sensor variables, in which case the execution trace of a finite state machine is conditioned on the outcome of such a test. Arcs can be tagged with messages that will be generated when traversing them, and that are sent to the robot's motors or to other finite state machines. Additionally, finite state machines possess internal timers (clocks) that control the time it takes to traverse an arc. The resulting machines are usually referred to as **augmented finite state machines**, or AFSMs, where the augmentation refers to the use of clocks.

AUGMENTED FINITE STATE MACHINE

An example of a simple AFSM is the four-state machine shown in Figure 25.22(b), which generates cyclic leg motion for a hexapod walker. This AFSM implements a cyclic controller, whose execution mostly does not rely on environmental feedback. The forward swing phase, however, relies on sensor feedback. If the leg is stuck, meaning that it has failed to execute the forward swing, the robot retracts the leg, lifts up a little higher, and attempts to execute the forward swing once again. Thus, the controller is able to *react* to contingencies arising from the interplay of the robot and its environment.

The subsumption architecture offers additional primitives for synchronizing AFSMs, and for combining output values of multiple, possibly conflicting AFSMs. In this way, it enables the programmer to compose increasingly complex controllers in a bottom-up fashion. In our example, we might begin with AFSMs for individual legs, followed by an AFSM for coordinating multiple legs. On top of this, we might implement higher-level behaviors such as collision avoidance, which might involve backing up and turning.

The idea of composing robot controllers from AFSMs is quite intriguing. Imagine how difficult it would be to generate the same behavior with any of the configuration space path planning algorithms described in the previous section. First, we would need an accurate model of the terrain. The configuration space of a robot with six legs, each of which is driven by two independent motors, totals eighteen dimensions (twelve dimensions for the configuration of the legs, and six for the location and orientation of the robot relative to its environment). Even if our computers were fast enough to find paths in such high-dimensional spaces, we would have to worry about nasty effects such as the robot sliding down a slope. Because of such stochastic effects, a single path through configuration space would almost certainly be too brittle, and even a PID controller might not be able to cope with such contingencies. In other words, generating motion behavior deliberately is simply too complex a problem for present-day robot motion planning algorithms.

Unfortunately, the subsumption architecture has problems of its own. First, the AFSMs are usually driven by raw sensor input, an arrangement that works if the sensor data is reliable and contains all necessary information for decision making, but fails if sensor data has to be integrated in nontrivial ways over time. Subsumption-style controllers have therefore mostly been applied to local tasks, such as wall following or moving towards visible light sources. Second, the lack of deliberation makes it difficult to change the task of the robot. A subsumption-style robot usually does just one task, and it has no notion of how to modify its controls to accommodate different control objectives (just like the dung beetle on page 37). Finally, subsumption-style controllers tend to be difficult to understand. In practice, the intricate interplay between dozens of interacting AFSMs (and the environment) is beyond what most human programmers can comprehend. For all these reasons, the subsumption architecture is rarely used in commercial robotics, despite its great historical importance. However, some its descendants are.

Three-layer architecture

THREE-LAYER ARCHITECTURE

REACTIVE LAYER

EXECUTIVE LAYER

Hybrid architectures combine reaction with deliberation. By far the most popular hybrid architecture is the **three-layer architecture**, which consists of a reactive layer, an executive layer, and a deliberate layer.

The **reactive layer** provides low-level control to the robot. It is characterized by a tight sensor-action loop. Its decision cycle is often on the order of milliseconds.

The **executive layer** (or sequencing layer) serves as the glue between the reactive and the deliberate layer. It accepts directives by the deliberate layer, and sequences them for the reactive layer. For example, the executive layer might handle a set of via-points generated by a deliberate path planner, and make decisions as to which reactive behavior to invoke.

Decision cycles at the executive layer are usually in the order of a second. The executive layer is also responsible for integrating sensor information into an internal state representation. For example, it may host the robot's localization and online mapping routines.

The **deliberate layer** generates global solutions to complex tasks using planning. Because of the computational complexity involved in generating such solutions, its decision cycle is often in the order of minutes. The deliberate layer (or planning layer) uses models for decision making. Those models might be pre-supplied or learned from data, and they usually utilize state information gathered at the executive layer.

Variants of the three-layer architecture can be found in most modern-day robot software systems. The decomposition into three layers is not very strict. Some robot software systems possess additional layers, such as user interface layers that control the interaction with people, or layers responsible for coordinating a robot's actions with that of other robots operating in the same environment.

Robotic programming languages

Many robotic controllers have been implemented with special purpose programming languages. For example, many programs for the subsumption architecture have been implemented in the **behavior language** defined by Brooks (1990). This language is a rule-based real-time control language that compiles into AFSM controllers. Individual rules in a Lisp-like syntax are compiled into AFSMs, and multiple AFSMs are integrated through a collection of local and global message-passing mechanisms.

Just like the subsumption architecture, the behavior language is limited in its focus on simple AFSMs with a relatively narrow definition of the communication flow between modules. Recent research has built on this idea, leading to a range of programming languages similar in spirit to the behavior language, but more powerful and faster when executed. One such language is the **generic robot language**, or GRL (Horswill, 2000). GRL is a functional programming language for programming large modular control systems. Just as in the behavior language, GRL uses finite state machines as its basic building blocks. On top of this, it provides a much broader range of constructs for defining communication flow and synchronization constraints between different modules than the behavior language. Programs in GRL are compiled into efficient imperative languages, such as C.

Another important programming language (and associated architecture) for concurrent robot software is the reactive action plan system, or RAPS (Firby, 1994). RAPS enables programmers to specify goals, plans (or partial policies) associated with these goals, and conditions under which those plans will likely succeed. Crucially, RAPS also provides facilities for handling the inevitable failures that occur with real robotic systems. The programmer can specify detection routines for various kinds of failure and supply an exception-handling routine for each kind. In three-layer architectures, RAPS is often used in the executive layer, to handle contingencies that do not require replanning.

There are several other languages that allow for reasoning and learning to take place in the robot. For example, GOLOG (Levesque *et al.*, 1997b) is a programming language that seamlessly blends deliberate problem solving (planning) and direct specification of reactive

control. Programs in GOLOG are formulated in situation calculus (Section 10.3), with the additional option of nondeterministic action operators. In addition to the specification of a control program with possible nondeterministic actions, the programmer also has to provide a complete model of the robot and its environment. Whenever the control program reaches a nondeterministic choice point, a planner (in the form of a theorem prover) is invoked to determine what to do next. In this way, the programmer can specify partial controllers and rely on built-in planners to make the final control choice. The beauty of GOLOG lies in its seamless integration of reactivity and deliberation. Despite the strong requirements of GOLOG (full observability, discrete states, full model), GOLOG has provided high-level control for a series of indoor mobile robots.

CES

CES, short for C++ for embedded systems, is a language extension of C++ that integrates probabilities and learning (Thrun, 2000). CES's data types are probability distributions, allowing the programmer to calculate with uncertain information without the effort usually required to implement probabilistic techniques. More importantly, CES makes it possible to train robot software with examples, very much like the learning algorithms discussed in Chapter 20. CES enables programmers to leave "gaps" in the code that are filled by learnable functions—typically differentiable parametric representations such as neural networks. These functions are then learned inductively in explicit training phases, where the trainer has to specify the desired output behavior. CES has been demonstrated to work well in partially observable and continuous domains.

ALISP

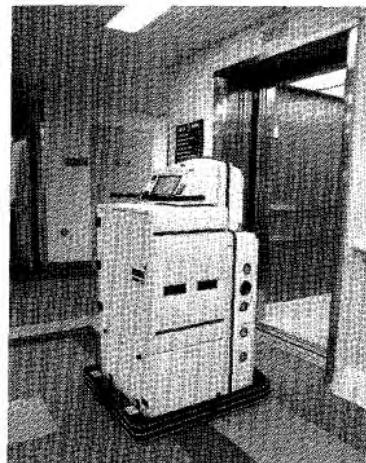
ALisp (Andre and Russell, 2002) is an extension of Lisp. ALisp allows programmers to specify nondeterministic choice points, similar to the choice points in GOLOG. However, instead of relying on a theorem prover to make decisions, ALisp inductively learns the right action via reinforcement learning. Thus ALisp can be seen as a flexible means to incorporate domain knowledge—especially knowledge about the hierarchical "subroutine" structure of desired behaviors—into a reinforcement learner. As yet, ALisp has been applied to robotics problems only in simulation, but it provides a promising methodology for building robots that learn through interaction with the environment.

25.8 APPLICATION DOMAINS

We will now list some of the prime application domains for robotic technology.

Industry and Agriculture. Traditionally, robots have been fielded in areas that require difficult human labor, yet are structured enough to be amenable to robotic automation. The best example is the assembly line, where manipulators routinely perform tasks such as assembly, part placement, material handling, welding, and painting. In many of these tasks, robots have become more cost-effective than human workers.

Outdoors, many of the heavy machines that we use to harvest, mine, or excavate earth have been turned into robots. For example, a recent project at Carnegie Mellon has demonstrated that robots can strip paint off large ships about 50 times faster than people can, and with a much reduced environmental impact. Prototypes of autonomous mining robots have



(a)



(b)

Figure 25.23 (a) The Helpmate robot transports food and other medical items in dozens of hospitals world-wide. (b) Surgical robots in the operating room (by da Vinci Surgical Systems).

been found to be faster and more precise than people in transporting ore in underground mines. Robots have been used to generate high-precision maps of abandoned mines and sewer systems. While many of these systems are still in their prototype stages, it is only a matter of time until robots will take over much of the semi-mechanical work that is presently performed by people.

Transportation. Robotic transportation has many facets: from autonomous helicopters that deliver objects to locations that would be hard to access by other means, to automatic wheelchairs that transport people who are unable to control wheelchairs by themselves, to autonomous straddle carriers that outperform skilled human drivers when transporting containers from ships to trucks on loading docks. A prime example of indoor transportation robots, or gofers, is the Helpmate robot shown in Figure 25.23(a). This robot has been deployed in dozens of hospitals to transport food and other items. Researchers have developed car-like robotic systems that can navigate autonomously on highways or across off-road terrain. In factory settings, autonomous vehicles are now routinely deployed to transport goods in warehouses and between production lines.

Many of these robots require environmental modifications for their operation. The most common modifications are localization aids such as inductive loops in the floor, active beacons, bar-code tags, and GPS satellites. An open challenge in robotics is the design of robots that can use natural cues, instead of artificial devices, to navigate, particularly in environments such as the deep ocean where GPS is unavailable.

Hazardous environments. Robots have assisted people in cleaning up nuclear waste, most notably in Chernobyl and Three Mile Island. Robots were present after the collapse of the World Trade Center, where they entered structures deemed too dangerous for human search and rescue crews.

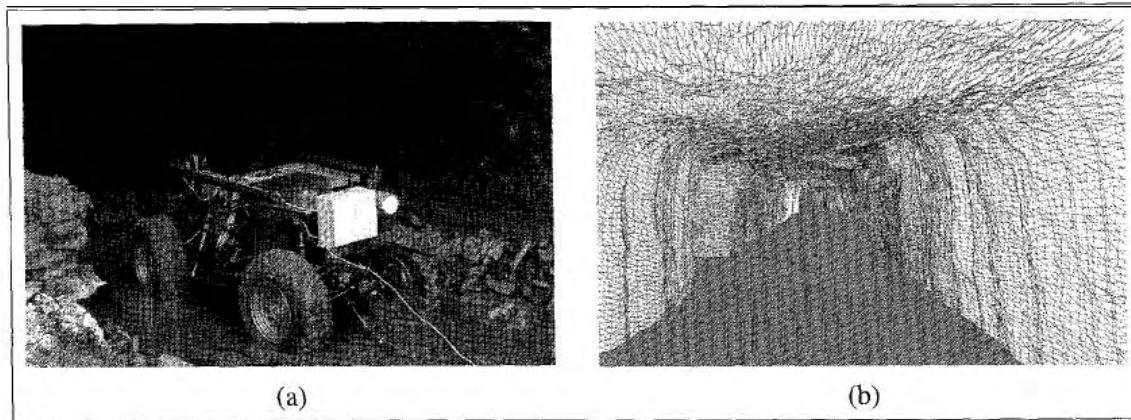


Figure 25.24 (a) A robot mapping an abandoned coal mine. (b) A 3-D map of the mine acquired by the robot.

Some countries have used robots to transport ammunition and to defuse bombs—a notoriously dangerous task. A number of research projects are presently developing prototype robots for clearing minefields, on land and at sea. Most existing robots for these tasks are teleoperated—a human operates them by remote control. Providing such robots with autonomy is an important next step.

Exploration. Robots have gone where no-one has gone before, including the surface of Mars. (See Figure 25.1(a).) Robotic arms assist astronauts in deploying and retrieving satellites and in building the International Space Station. Robots also help explore under the sea. They are routinely used to acquire maps of sunken ships. Figure 25.24 shows a robot mapping an abandoned coal mine, along with a 3-D model of the mine acquired using range sensors. In 1996, a team of researchers released a legged robot into the crater of an active volcano to acquire data important for climate research. Unmanned air vehicles known as **drones** are used in military operations. Robots are becoming very effective tools for gathering information in domains that are difficult (or dangerous) to access for people.

DRONES

Health care. Robots are increasingly used to assist surgeons with instrument placement when operating on organs as intricate as brains, eyes, and hearts. Figure 25.23(b) shows such a system. Robots have become indispensable tools in certain types of hip replacements, thanks to their high precision. In pilot studies, robotic devices have been found to reduce the danger of lesions when performing colonoscopies. Outside the operating room, researchers have begun to develop robotic aides for elderly and handicapped people, such as intelligent robotic walkers and intelligent toys that provide reminders to take medication.

Personal Services. Service is an up-and-coming application domain of robotics. Service robot assist individuals in performing daily tasks. Commercially available domestic service robots include autonomous vacuum cleaners, lawn mowers, and golf caddies. All these robots can navigate autonomously and perform their tasks without human help. Some service robots operate in public places, such as robotic information kiosks that have been deployed in shopping malls and trade fairs, or in museums as tour-guides. Service tasks require human interaction, and the ability to cope robustly with unpredictable and dynamic environments.

Entertainment. Robots have begun to conquer the entertainment and toy industry. We saw the Sony AIBO in Figure 25.4(b); this dog-like robot toy is being used as a research platform in AI labs around the world. One of the challenging AI tasks studied with this platform is robotic soccer, a competitive game very much like human soccer, but played with autonomous mobile robots. Robot soccer provides great opportunities for research in AI, since it raises a range of problems prototypical for many other, more serious robot applications. Annual robotic soccer competitions have attracted large numbers of AI researchers and added a lot of excitement to the field of robotics.

Human augmentation. A final application domain of robotic technology is that of human augmentation. Researchers have developed legged walking machines that can carry people around, very much like a wheelchair. Several research efforts presently focus on the development devices that make it easier for people to walk or move their arms, by providing additional forces through extra-skeletal attachments. If such devices are attached permanently, they can be thought of as artificial robotic limbs. Robotic teleoperation, or telepresence, is another form of human augmentation. Teleoperation involves carrying out tasks over long distances, with the aid of robotic devices. A popular configuration for robotic teleoperation is the master–slave configuration, where a robot manipulator emulates the motion of a remote human operator, measured through a haptic interface. All these systems augment people's ability to interact with their environments. Some projects go as far as replicating humans, at least at a very superficial level. Humanoid robots are now available commercially through several companies in Japan.

25.9 SUMMARY

Robotics concerns itself with intelligent agents that manipulate the physical world. In this chapter, we have learned the following basics of robot hardware and software.

- Robots are equipped with sensors for perceiving their environment and effectors with which they can assert physical forces on their environment. Most robots are either manipulators anchored at fixed locations or mobile robots that can move.
- Robotic perception concerns itself with estimating decision-relevant quantities from sensor data. To do so, we need an internal representation and a method for updating this internal representation over time. Common examples of hard perceptual problems include localization and mapping.
- Probabilistic filtering algorithms such as Kalman filters and particle filters are useful for robot perception. These techniques maintain the belief state, i.e., a posterior distribution over state variables.
- The planning of robot motion is usually done in configuration space, where each point specifies the location and orientation of the robot and its joint angles.
- Configuration spaces search algorithms include cell decomposition techniques, which decompose the space of all configurations into finitely many cells, and skeletonization

techniques, which project configuration spaces onto lower-dimensional manifolds. The motion planning problem is then solved using search in these simpler structures.

- A path found by a search algorithm can be executed by using the path as the reference trajectory for a PID controllers.
- Potential field techniques navigate robots by potential functions, defined over the distance to obstacles and the target location. Potential field techniques may get stuck in local minima, but they can generate motion directly without the need for path planning.
- Sometimes, it is easier to specify a robot controller directly, rather than deriving a path from an explicit model of the environment. Such controllers can often be written as simple finite state machines.
- The subsumption architecture enables programmers to compose robot controllers from interconnected finite state machines, augmented by built-in timers.
- Three-layer architectures are common frameworks for developing robot software that integrate deliberation, sequencing of subgoals, and control.
- Special purpose programming languages exist that facilitate robot software development. These languages offer constructs for developing multithreaded software, for integrating control directives into planning, and for learning from experience.

BIBLIOGRAPHICAL AND HISTORICAL NOTES

The word **robot** was popularized by Czech playwright Karel Čapek in his 1921 play *R.U.R.* (Rossum's Universal Robots). The robots, which were grown chemically rather than constructed mechanically, end up resenting their masters and decide to take over. It appears (Glanc, 1978) that it was actually Čapek's brother, Josef, who first combined the Czech words "robořit" (obligatory work) and "robotník" (serf) to yield "robot" in his 1917 short story *Opilec*.

The term *robotics* was first used by (Asimov, 1950). Robotics (under other names) has a much longer history, however. In ancient Greek mythology, a mechanical man named Talos was supposedly designed and built by Hephaestos, the Greek god of metallurgy. Wonderful automata were built in the 18th century—Jacques Vaucanson's mechanical duck from 1738 being one early example—but the complex behaviors they exhibited were entirely fixed in advance. Possibly the earliest example of a programmable robot-like device was the Jacquard loom (1805), described in Chapter 1.

UNIMATE

The first commercial robot was a robot arm called **Unimate**, short for *universal automation*. Unimate was developed by Joseph Engelberger and George Devol. In 1961, the first Unimate robot was sold to General Motors, where it was used for manufacturing TV picture tubes. 1961 was also the year when Devol obtained the first U.S. patent on a robot. Eleven years later, in 1972, Nissan Corp. was among the first to automate an entire assembly line with robots, developed by Kawasaki with robots supplied by Engelberger and Devol's company Unimation. This development initiated a major revolution that took place mostly in Japan and the U.S., and that is still ongoing. Unimation followed up in 1978 with the devel-

opment of the **PUMA** robot, short for Programmable Universal Machine for Assembly. The PUMA robot, initially developed for General Motors, was the *de facto* standard for robotic manipulation for the two decades that followed. At present, the number of operating robots is estimated at one million world-wide, more than half of which are installed in Japan.

The literature on robotics research can be divided roughly into two parts: mobile robots and stationary manipulators. Grey Walter's "turtle," built in 1948, could be considered the first autonomous mobile robot, although its control system was not programmable. The "Hopkins Beast," built in the early 1960s at Johns Hopkins University, was much more sophisticated; it had pattern-recognition hardware and could recognize the cover plate of a standard AC power outlet. It was capable of searching for outlets, plugging itself in, and then recharging its batteries! Still, the Beast had a limited repertoire of skills. The first general-purpose mobile robot was "Shakey," developed at what was then the Stanford Research Institute (now SRI) in the late 1960s (Fikes and Nilsson, 1971; Nilsson, 1984). Shakey was the first robot to integrate perception, planning, and execution, and much subsequent research in AI was influenced by this remarkable achievement. Other influential projects include the Stanford Cart and the CMU Rover (Moravec, 1983). Cox and Wilfong (1990) describes classic work on autonomous vehicles.

The field of robotic mapping has evolved from two distinct origins. The first thread began with work by Smith and Cheeseman (1986), who applied Kalman filters to the simultaneous localization and mapping problem. This algorithm was first implemented by Moutarlier and Chatila (1989), and later extended to by Leonard and Durrant-Whyte (1992). Dissanayake *et al.* (2001) describes the state of the art. The second thread began with the development of the **occupancy grid** representation for probabilistic mapping, which specifies the probability that each (x, y) location is occupied by an obstacle (Moravec and Elfes, 1985). An overview of the state of the art in robotic mapping can be found in (Thrun, 2002). Kuipers and Levitt (1988) were among the first to propose topological rather than metric mapping, motivated by models of human spatial cognition.

Early mobile robot localization techniques are surveyed by Borenstein *et al.* (1996). Although Kalman filtering was well-known as a localization method in control theory for decades, the general probabilistic formulation of the localization problem did not appear in the AI literature until much later, through the work of Tom Dean and colleagues (1990, 1990) and Simmons and Koenig (1995). The latter work introduced the term **Markov localization**. The first real-world application of this technique was by Burgard *et al.* (1999), through a series of robots that were deployed in museums. Monte Carlo localization based on particle filters was developed by Fox *et al.* (1999) and is now widely used. The **Rao-Blackwellized particle filter** combines particle filtering for robot localization with exact filtering for map building (Murphy and Russell, 2001; Montemerlo *et al.*, 2002).

Research on mobile robotics has been stimulated over the last decade by two important competitions. AAAI's annual mobile robot competition began in 1992. The first competition winner was CARMEL (Congdon *et al.*, 1992). Progress has been steady and impressive: in the most recent competition (2002), the robots had to enter the conference complex, find their way to the registration desk, register for the conference, and give a talk. The **Robocup** competition, launched in 1995 by Kitano and colleagues (1997), aims by 2050 to "develop a

OCCUPANCY GRID

MARKOV LOCALIZATION

RAO-BLACKWELLIZED PARTICLE FILTER

ROBOCUP

team of fully autonomous humanoid robots that can win against the human world champion team in soccer." Play occurs in leagues for simulated robots, wheeled robots of different sizes, and four-legged Sony Aibo robots. In 2002, the competition event drew teams from almost 30 different countries and over 100,000 spectators.

HAND-EYE
MACHINES

The study of manipulator robots, called originally called **hand-eye machines**, has evolved along quite different lines. The first major effort at creating a hand–eye machine was Heinrich Ernst's MH-1, described in his MIT Ph.D. thesis (Ernst, 1961). The Machine Intelligence project at Edinburgh also demonstrated an impressive early system for vision-based assembly called FREDDY (Michie, 1972). After these pioneering efforts, a great deal of work focused on geometric algorithms for deterministic and fully observable motion planning problems. The PSPACE-hardness of robot motion planning was shown in a seminal paper by Reif (1979). The configuration space representation is due to Lozano-Perez (1983). Highly influential was a series of papers by Schwartz and Sharir on what they called **piano movers** problems (Schwartz *et al.*, 1987).

PIANO MOVERS

Recursive cell decomposition for configuration space planning was originated by Brooks and Lozano-Perez (1985) and improved significantly by Zhu and Latombe (1991). The earliest skeletonization algorithms were based on Voronoi diagrams (Rowat, 1979) and **visibility graphs** (Wesley and Lozano-Perez, 1979). Guibas *et al.* (1992) developed efficient techniques for calculating Voronoi diagrams incrementally, and Choset (1996) generalized Voronoi diagrams to much broader motion planning problems. John Canny's Ph.D. thesis (1988) established the first singly exponential algorithm for motion planning using a different skeletonization method called the **silhouette** algorithm. The text by Jean-Claude Latombe (1991) covers a variety of approaches to the motion planning problem. (Kavraki *et al.*, 1996) developed probabilistic roadmaps, which are currently the most effective method. Fine motion planning with limited sensing was investigated by (Lozano-Perez *et al.*, 1984) and Canny and Reif (1987) using the idea of interval uncertainty rather than probabilistic uncertainty. Landmark-based navigation (Lazanas and Latombe, 1992) uses many of the same ideas in the mobile robot arena.

VISIBILITY GRAPH

The control of robots as dynamical systems—whether for manipulation or navigation—has generated a huge literature on which the material in this chapter barely touches. Important works include a trilogy on impedance control by Hogan (1985) and a general study of robot dynamics by Featherstone (1987). Dean and Wellman (1991) were among the first to try to tie together control theory and AI planning systems. Three classical textbook on the mathematics of robot manipulation are due to Paul (1981), Craig (1989), and Yoshikawa (1990). The area of **grasping** is also important in robotics—the problem of determining a stable grasp is quite difficult (Mason and Salisbury, 1985). Competent grasping requires touch sensing, or **haptic feedback**, to determine contact forces and detect slip (Fearing and Hollerbach, 1985).

SILHOUETTE

Potential field control, which attempts to solve the motion planning and control problems simultaneously, was introduced into the robotics literature by Khatib (1986). In mobile robotics, this idea was viewed as a practical solution to the collision avoidance problem, and was later extended into an algorithm called **vector field histograms** by Borenstein (1991). Navigation functions, the robotics version of a control policy for deterministic MDPs, were introduced by Koditschek (1987).

GRASPING

HAPTIC FEEDBACK

VECTOR FIELD
HISTOGRAMS

The topic of software architectures for robots engenders much religious debate. The good old-fashioned AI candidate—the three-layer architecture—dates back to the design of Shakey and is reviewed by Gat (1998). The subsumption architecture is due to Rodney Brooks (1986), although similar ideas were developed independently by Braitenberg (1984), whose book, *Vehicles*, describes a series of simple robots based on the behavioral approach. The success of Brooks's six-legged walking robot was followed by many other projects. Connell, in his Ph.D. thesis (1989), developed a mobile robot capable of retrieving objects that was entirely reactive. Extensions of the behavior-based paradigm to multirobot systems can be found in (Mataric, 1997) and (Parker, 1996). GRL (Horswill, 2000) and COLBERT (Konolige, 1997) abstract the ideas of concurrent behavior-based robotics into general robot control languages. Arkin (1998) surveys the state of the art.

Situated automata (Rosenschein, 1985; Kaelbling and Rosenschein, 1990), described in Chapter 7, have also been used to control mobile robots for exploration and delivery tasks. Situated automata are closely related to behavior-based designs in that they consist of finite-state machines that track aspects of the environment state using simple combinatorial circuitry. Whereas the behavior-based approach stresses the absence of explicit representation, situated automata are constructed algorithmically from declarative environment models so that the representational content of each state register is well-defined.

There exist several good recent textbooks on mobile robotics. In addition to the textbooks referenced above, the collection by Kortenkamp *et al.* (1998) provides a comprehensive overview of contemporary mobile robot architectures and systems. Two recent textbooks by Dudek and Jenkin (2000) and Murphy (2000) cover robotics more generally. A recent book on robot manipulation addresses advanced topics such as compliant motion (Mason, 2001). The major conference for robotics is the IEEE *International Conference on Robotics and Automation*. Robotics journals include IEEE *Robotics and Automation*, the International *Journal of Robotics Research*, and *Robotics and Autonomous Systems*.

EXERCISES

25.1 Monte Carlo localization is *biased* for any finite sample size—i.e., the expected value of the location computed by the algorithm differs from the true expected value—because of the way particle filtering works. In this question, you are asked to quantify this bias.

To simplify, consider a world with four possible robot locations: $\mathbf{X} = \{x_1, x_2, x_3, x_4\}$. Initially, we draw $N \geq 1$ samples uniformly from among those locations. As usual, it is perfectly acceptable if more than one sample is generated for any of the locations \mathbf{X} . Let Z be a Boolean sensor variable characterized by the following conditional probabilities:

$$\begin{array}{ll} P(z | x_1) = 0.8 & P(\neg z | x_1) = 0.2 \\ P(z | x_2) = 0.4 & P(\neg z | x_2) = 0.6 \\ P(z | x_3) = 0.1 & P(\neg z | x_3) = 0.9 \\ P(z | x_4) = 0.1 & P(\neg z | x_4) = 0.9 \end{array}$$

MCL uses these probabilities to generate particle weights, which are subsequently normalized and used in the resampling process. For simplicity, let us assume we only generate one new sample in the resampling process, regardless of N . This sample might correspond to any of the four locations in X . Thus, the sampling process defines a probability distribution over X .

- a. What is the resulting probability distribution over X for this new sample? Answer this question separately for $N = 1, \dots, 10$, and for $N = \infty$.
- b. The difference between two probability distributions P and Q can be measured by the KL divergence, which is defined as

$$KL(P, Q) = \sum_i P(x_i) \log \frac{P(x_i)}{Q(x_i)}.$$

What are the KL divergences between the distributions in (a) and the true posterior?

- c. What modification of the problem formulation (not the algorithm!) would guarantee that the specific estimator above is unbiased even for finite values of N ? Provide at least two such modifications (each of which should be sufficient).



25.2 Implement Monte Carlo localization for a simulated robot with range sensors. A grid map and range data are available from the code repository at aima.cs.berkeley.edu. Your exercise is complete if you can demonstrate successful global localization of the robot.

25.3 Consider the robot arm shown in Figure 25.12. Assume that the robot's base element is 60cm long and that its upper arm and forearm are each 40cm long. As argued on page 917, the inverse kinematics of a robot is often not unique. State an explicit closed-form solution of the inverse kinematics for this arm. Under what exact conditions is the solution unique?



25.4 Implement an algorithm for calculating the Voronoi diagram of an arbitrary 2-D environment, described by an $n \times n$ Boolean array. Illustrate your algorithm by plotting the Voronoi diagram for 10 interesting maps. What is the complexity of your algorithm?

25.5 This exercise explores the relationship between workspace and configuration space using the examples shown in Figure 25.25.

- a. Consider the robot configurations shown in Figure 25.25(a) through (c), ignoring the obstacle shown in each of the diagrams. Draw the corresponding arm configurations in configuration space. (Hint: Each arm configuration maps to a single point in configuration space, as illustrated in Figure 25.12(b).)
- b. Draw the configuration space for each of the workspace diagrams in Figure 25.25(a)–(c). (Hint: The configuration spaces share with the one shown in Figure 25.25(a) the region that corresponds to self-collision, but differences arise from the lack of enclosing obstacles and the different locations of the obstacles in these individual figures.)
- c. For each of the black dots in Figure 25.25(e)–(f), draw the corresponding configurations of the robot arm in workspace. Please ignore the shaded regions in this exercise.
- d. The configuration spaces shown in Figure 25.25(e)–(f) have all been generated by a single workspace obstacle (dark shading), plus the constraints arising from the self-collision constraint (light shading). Draw, for each diagram, the workspace obstacle that corresponds to the darkly shaded area.

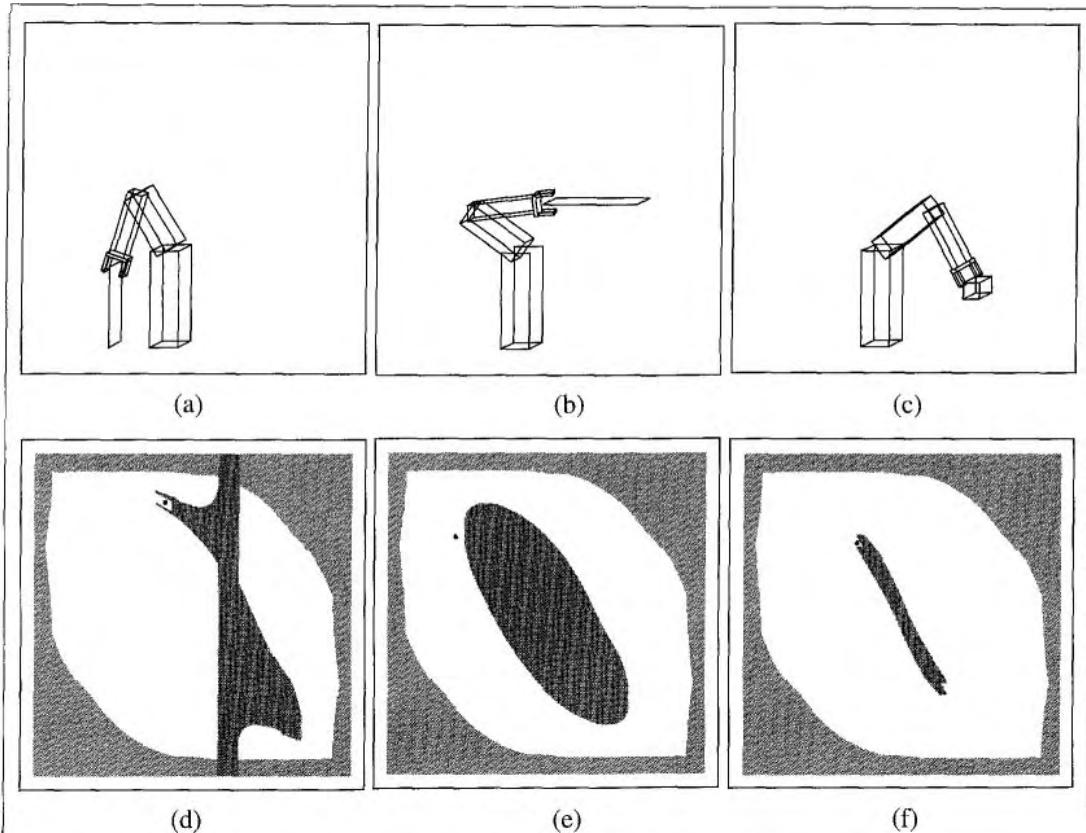


Figure 25.25 Diagrams for Exercise 25.5.

- e. Figure 25.25(d) illustrates that a single planar obstacle can decompose the workspace into two disconnected regions. What is the maximum number of disconnected regions that can be created by inserting a planar obstacle into an obstacle-free, connected workspace, for a 2DOF robot? Give an example, and argue why no larger number of disconnected regions can be created. How about a non-planar obstacle?

25.6 Consider the simplified robot shown in Figure 25.26. Suppose the robot's Cartesian coordinates are known at all times, as are those of its target location. However, the locations of the obstacles are unknown. The robot can sense obstacles in its immediate proximity, as illustrated in this figure. For simplicity, let us assume the robot's motion is noise-free, and the state space is discrete. Figure 25.26 is only one example; in this exercise you are required to address all possible grid worlds with a valid path from the start to the goal location.

- Design a deliberate controller that guarantees that the robot always reaches its target location if at all possible. The deliberate controller can memorize measurements in form of a map that is being acquired as the robot moves. Between individual moves, it may spend arbitrary time deliberating.
- Now design a reactive controller for the same task. This controller may not memorize past sensor measurements. (It may not build a map!) Instead, it has to make all decisions

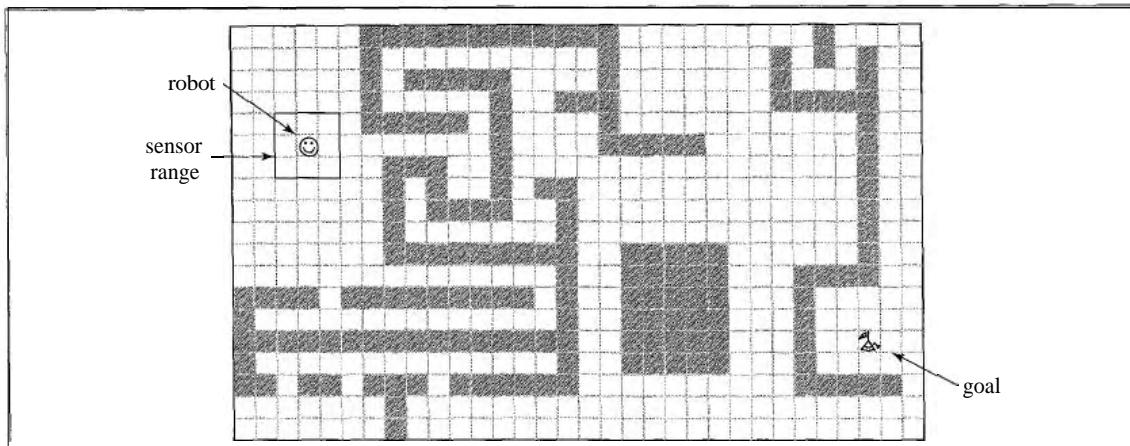


Figure 25.26 Simplified robot in a maze. See Exercise 25.6.

based on the current measurement, which includes knowledge of its own location and that of the goal. The time to make a decision must be independent of the environment size or the number of past time steps. What is the maximum number of steps that it may take for your robot to arrive at the goal?

- c. How will your controllers from (a) and (b) perform if any of the following six conditions apply: continuous state space, noise in perception, noise in motion, noise in both perception and motion, unknown location of the goal (the goal can be detected only when within sensor range), or moving obstacles. For each condition and each controller, give an example of a situation where the robot fails (or explain why it cannot).

25.7 In Figure 25.22(b), we encountered an augmented finite state machine for the control of a single leg of a hexapod robot. In this exercise, the aim is to design an AFSM that, when combined with six copies of the individual leg controllers, results in efficient, stable locomotion. For this purpose, you have to augment the individual leg controller to pass messages to your new AFSM, and to wait until other messages arrive. Argue why your controller is efficient, in that it does not unnecessarily waste energy (e.g., by sliding legs), and in that it propels the robot at reasonably high speeds. Prove that your controller satisfies the stability condition given on page 906.

25.8 (This exercise was first devised by Michael Genesereth and Nils Nilsson. It works for first graders through graduate students.) Humans are so adept at basic tasks such as picking up cups or stacking blocks that they often forget how complex these tasks are. In this exercise you will discover the complexity and recapitulate the last 30 years of developments in robotics. First, pick a task, such as building an arch out of three blocks. Then, build a robot out of four humans as follows:

Brain. The job of the Brain is to come up with a plan to achieve the goal, and to direct the hands in the execution of the plan. The Brain receives input from the Eyes, but *cannot see the scene directly*. The brain is the only one who knows what the goal is.

Eyes. The Eyes' job is to report a brief description of the scene to the Brain. The Eyes should stand a few feet away from the working environment, and can provide qualitative

descriptions (such as "There is a red box standing on top of a green box, which is on its side") or quantitative descriptions ("The green box is about two feet to the left of the blue cylinder"). Eyes can also answer questions from the Brain such as, "Is there a gap between the Left Hand and the red box?" If you have a video camera, point it at the scene and allow the eyes to look at the viewfinder of the video camera, but not directly at the scene.

Left hand and right hand. One person plays each Hand. The two Hands stand next to each other; the Left Hand uses only his or her left hand, and the Right Hand only his or her right hand. The Hands execute only simple commands from the Brain—for example, "Left Hand, move two inches forward." They cannot execute commands other than motions; for example, "Pick up the box" is not something a Hand can do. To discourage cheating, you might want to have the hands wear gloves, or have them operate tongs. The Hands must be blindfolded. The only sensory capability they have is the ability to tell when their path is blocked by an immovable obstacle such as a table or the other Hand. In such cases, they can beep to inform the Brain of the difficulty.

26

PHILOSOPHICAL FOUNDATIONS

In which we consider what it means to think and whether artifacts could and should ever do so.

As we mentioned in Chapter 1, philosophers have been around for much longer than computers and have been trying to resolve some questions that relate to AI: How do minds work? Is it possible for machines to act intelligently in the way that people do, and if they did, would they have minds? What are the ethical implications of intelligent machines? For the first 25 chapters of this book, we have considered questions from AI itself; now we consider the philosopher's agenda for one chapter.

WEAK AI

First, some terminology: the assertion that machines could possibly act intelligently (or, perhaps better, act *as if* they were intelligent) is called the **weak AI** hypothesis by philosophers, and the assertion that machines that do so are *actually* thinking (as opposed to *simulating* thinking) is called the **strong AI** hypothesis.

STRONG AI

Most AI researchers take the weak AI hypothesis for granted, and don't care about the strong AI hypothesis—as long as their program works, they don't care whether you call it a simulation of intelligence or real intelligence. All AI researchers should be concerned with the ethical implications of their work.

26.1 WEAK AI: CAN MACHINES ACT INTELLIGENTLY?

Some philosophers have tried to prove that AI is impossible; that machines cannot possibly act intelligently. Some have used their arguments to call for a stop to AI research:

Artificial intelligence pursued within the cult of computationalism stands not even a ghost of a chance of producing durable results . . . it is time to divert the efforts of AI researchers—and the considerable monies made available for their support—into avenues other than the computational approach. (Sayre, 1993)

Clearly, whether AI is impossible depends on how it is defined. In essence, AI is the quest for the best agent program on a given architecture. With this formulation, AI is by definition possible: for any digital architecture consisting of k bits of storage there are exactly 2^k agent

programs, and all we have to do to find the best one is enumerate and test them all. This might not be feasible for large k , but philosophers deal with the theoretical, not the practical.

Our definition of AI works well for the engineering problem of finding a good agent, given an architecture. Therefore, we're tempted to end this section right now, answering the title question in the affirmative. But philosophers are interested in the problem of comparing two architectures—human and machine. Furthermore, they have traditionally posed the question as, "Can machines think?" Unfortunately, this question is ill-defined. To see why, consider the following questions:

- Can machines fly?
- Can machines swim?

Most people agree that the answer to the first question is yes, airplanes can fly, but the answer to the second is no; boats and submarines do move through the water, but we do not call that swimming. However, neither the questions nor the answers have any impact at all on the working lives of aeronautic and naval engineers or on the users of their products. The answers have very little to do with the design or capabilities of airplanes and submarines, and much more to do with the way we have chosen to use words. The word "swim" in English has come to mean "to move along in the water by movement of body parts," whereas the word "fly" has no such limitation on the means of locomotion.¹ The practical possibility of "thinking machines" has been with us for only 50 years or so, not long enough for speakers of English to settle on a meaning for the word "think."

Alan Turing, in his famous paper "Computing Machinery and Intelligence" (Turing, 1950), suggested that instead of asking whether machines can think, we should ask whether machines can pass a behavioral intelligence test, which has come to be called the Turing Test. The test is for a program to have a conversation (via online typed messages) with an interrogator for 5 minutes. The interrogator then has to guess if the conversation is with a program or a person; the program passes the test if it fools the interrogator 30% of the time. Turing conjectured that, by the year 2000, a computer with a storage of 10^9 units could be programmed well enough to pass the test, but he was wrong. Some people **have** been fooled for 5 minutes; for example, the ELIZA program and the Internet chatbot called MGONZ have fooled humans who didn't realize they might be talking to a program, and the program ALICE fooled one judge in the 2001 Loebner Prize competition. But no program has come close to the 30% criterion against trained judges, and the field of AI as a whole has paid little attention to Turing tests.

Turing also examined a wide variety of possible objections to the possibility of intelligent machines, including virtually all of those that have been raised in the half century since his paper appeared. We will look at some of them.

The argument from disability

The "argument from disability" makes the claim that "a machine can never do X" As examples of X, Turing lists the following:

¹ In Russian, the equivalent of "swim" **does** apply to ships.

Be kind, resourceful, beautiful, friendly, have initiative, have a sense of humor, tell right from wrong, make mistakes, fall in love, enjoy strawberries and cream, make someone fall in love with it, learn from experience, use words properly, be the subject of its own thought, have as much diversity of behavior as man, do something really new.

Turing had to use his intuition to guess what would be possible in the future, but we have the luxury of looking back at what computers have already done. It is undeniable that computers now do many things that previously were the domain of humans alone. Programs play chess, checkers and other games, inspect parts on assembly lines, check the spelling of word processing documents, steer cars and helicopters, diagnose diseases, and do hundreds of other tasks as well as or better than humans. Computers have made small but significant discoveries in astronomy, mathematics, chemistry, mineralogy, biology, computer science, and other fields. Each of these required performance at the level of a human expert.

Given what we now know about computers, it is not surprising that they do well at combinatorial problems such as playing chess. But algorithms also perform at human levels on tasks that seemingly involve human judgment, or as Turing put it, "learning from experience" and the ability to "tell right from wrong." As far back as 1955, Paul Meehl (see also Grove and Meehl, 1996) studied the decision-making processes of trained experts at subjective tasks such as predicting the success of a student in a training program, or the recidivism of a criminal. In 19 out of the 20 studies he looked at, Meehl found that simple statistical learning algorithms (such as linear regression or naive Bayes) predict better than the experts. The Educational Testing Service has used an automated program to grade millions of essay questions on the GMAT exam since 1999. The program agrees with human graders 97% of the time, about the same level that two human graders agree (Burstein *et al.*, 2001).

It is clear that computers can do many things as well as or better than humans, including things that people believe require great human insight and understanding. This does not mean, of course, that computers use insight and understanding in performing these tasks—those are not part of behavior, and we address such questions elsewhere—but the point is that one's first guess about the mental processes required to produce a given behavior is often wrong. It is also true, of course, that there are many tasks at which computers do not yet excel (to put it mildly), including Turing's task of carrying on an open-ended conversation.

The mathematical objection

It is well known, through the work of Turing (1936) and Gödel (1931), that certain mathematical questions are in principle unanswerable by particular formal systems. Gödel's incompleteness theorem (see Section 9.5) is the most famous example of this. Briefly, for any formal axiomatic system F powerful enough to do arithmetic, it is possible to construct a so-called "Gödel sentence" $G(F)$ with the following properties:

- $G(F)$ is a sentence of F , but cannot be proved within F .
- If F is consistent, then $G(F)$ is true.

Philosophers such as J. R. Lucas (1961) have claimed that this theorem shows that machines are mentally inferior to humans, because machines are formal systems that are limited by the incompleteness theorem—they cannot establish the truth of their own Gödel sentence—while

humans have no such limitation. This claim has caused decades of controversy, spawning a vast literature including two books by the mathematician Sir Roger Penrose (1989, 1994) that repeat the claim with some fresh twists (such as the hypothesis that humans are different because their brains operate by quantum gravity). We will examine only three of the problems with the claim.

First, Gödel's incompleteness theorem applies only to formal systems that are powerful enough to do arithmetic. This includes Turing machines, and Lucas's claim is in part based on the assertion that computers are Turing machines. This is a good approximation, but is not quite true. Turing machines are infinite, whereas computers are finite, and any computer can therefore be described as a (very large) system in propositional logic, which is not subject to Gödel's incompleteness theorem.

Second, an agent should not be too ashamed that it cannot establish the truth of some sentence while other agents can. Consider the sentence

J. R. Lucas cannot consistently assert that this sentence is true.

If Lucas asserted this sentence then he would be contradicting himself, so therefore Lucas cannot consistently assert it, and hence it must be true. (The sentence cannot be false, because if it were then Lucas could not consistently assert it, so it would be true.) We have thus demonstrated that there is a sentence that Lucas cannot consistently assert while other people (and machines) can. But that does not make us think less of Lucas. To take another example, no human could compute the sum of 10 billion 10 digit numbers in his or her lifetime, but a computer could do it in seconds. Still, we do not see this as a fundamental limitation in the human's ability to think. Humans were behaving intelligently for thousands of years before they invented mathematics, so it is unlikely that mathematical reasoning plays more than a peripheral role in what it means to be intelligent.

Third, and most importantly, even if we grant that computers have limitations on what they can prove, there is no evidence that humans are immune from those limitations. It is all too easy to show rigorously that a formal system cannot do X, and then claim that humans can do X using their own informal method, without giving any evidence for this claim. Indeed, it is impossible to prove that humans are not subject to Gödel's incompleteness theorem, because any rigorous proof would itself contain a formalization of the claimed unformalizable human talent, and hence refute itself. So we are left with an appeal to intuition that humans can somehow perform superhuman feats of mathematical insight. This appeal is expressed with arguments such as "we must assume our own consistency, if thought is to be possible at all" (Lucas, 1976). But if anything, humans are known to be inconsistent. This is certainly true for everyday reasoning, but it is also true for careful mathematical thought. A famous example is the four-color map problem. Alfred Kempe published a proof in 1879 that was widely accepted and contributed to his election as a Fellow of the Royal Society. In 1890, however, Percy Heawood pointed out a flaw and the theorem remained unproved until 1977.

The argument from informality

One of the most influential and persistent criticisms of AI as an enterprise was raised by Turing as the "argument from informality of behavior." Essentially, this is the claim that human

behavior is far too complex to be captured by any simple set of rules and that because computers can do no more than follow a set of rules, they cannot generate behavior as intelligent as that of humans. The inability to capture everything in a set of logical rules is called the **qualification problem** in AI. (See Chapter 10.)

The principal proponent of this view has been the philosopher Hubert Dreyfus, who has produced a series of influential critiques of artificial intelligence: *What Computers Can't Do* (1972), *What Computers Still Can't Do* (1992), and, with his brother Stuart, *Mind Over Machine* (1986).

The position they criticize came to be called "Good (Old-Fashioned)AI," or GOFAI, a term coined by Haugeland (1985). GOFAI is supposed to claim that all intelligent behavior can be captured by a system that reasons logically from a set of facts and rules describing the domain. It therefore corresponds to the simplest logical agent described in Chapter 7. Dreyfus is correct in saying that logical agents are vulnerable to the qualification problem. As we saw in Chapter 13, probabilistic reasoning systems are more appropriate for open-ended domains. The Dreyfus critique therefore is not addressed against computers *per se*, but rather against one particular way of programming them. It is reasonable to suppose, however, that a book called *What First-Order Logical Rule-Based Systems Without Learning Can't Do* might have had less impact.

Under Dreyfus's view, human expertise does include knowledge of some rules, but only as a "holistic context" or "background" within which humans operate. He gives the example of appropriate social behavior in giving and receiving gifts: "Normally one simply responds in the appropriate circumstances by giving an appropriate gift." One apparently has "a direct sense of how things are done and what to expect." The same claim is made in the context of chess playing: "A mere chess master might need to figure out what to do, but a grandmaster just sees the board as demanding a certain move . . . the right response just pops into his or her head." It is certainly true that much of the thought processes of a present-giver or grandmaster is done at a level that is not open to introspection by the conscious mind. But that does not mean that the thought processes do not exist. The important question that Dreyfus does not answer is *how* the right move gets into the grandmaster's head. One is reminded of Daniel Dennett's (1984) comment,

It is rather as if philosophers were to proclaim themselves expert explainers of the methods of stage magicians, and then, when we ask how the magician does the sawing-the-lady-in-half trick, they explain that it is really quite obvious: the magician doesn't really saw her in half; he simply makes it appear that he does. "But how does he do *that*?" we ask. "Not our department," say the philosophers.

Dreyfus and Dreyfus (1986) propose a five-stage process of acquiring expertise, beginning with rule-based processing (of the sort proposed in GOFAI) and ending with the ability to select correct responses instantaneously. In making this proposal, Dreyfus and Dreyfus in effect move from being AI critics to AI theorists—they propose a neural network architecture organized into a vast "case library," but point out several problems. Fortunately, all of their problems have been addressed, some with partial success and some with total success. Their problems include:

1. Good generalization from examples cannot be achieved without background knowledge. They claim no one has any idea how to incorporate background knowledge into the neural network learning process. In fact, we saw in Chapter 19 that there are techniques for using prior knowledge in learning algorithms. Those techniques, however, rely on the availability of knowledge in explicit form, something that Dreyfus and Dreyfus strenuously deny. In our view, this is a good reason for a serious redesign of current models of neural processing so that they *can* take advantage of previously learned knowledge in the way that other learning algorithms do.
2. Neural network learning is a form of supervised learning (see Chapter 18), requiring the prior identification of relevant inputs and correct outputs. Therefore, they claim, it cannot operate autonomously without the help of a human trainer. In fact, learning without a teacher can be accomplished by unsupervised learning (Chapter 20) and reinforcement learning (Chapter 21).
3. Learning algorithms do not perform well with many features, and if we pick a subset of features, "there is no known way of adding new features should the current set prove inadequate to account for the learned facts." In fact, new methods such as support vector machines handle large feature sets very well. As we saw in Chapter 19, there are also principled ways to generate new features, although much more work is needed.
4. The brain is able to direct its sensors to seek relevant information and to process it to extract aspects relevant to the current situation. But, they claim, "Currently, no details of this mechanism are understood or even hypothesized in a way that could guide AI research." In fact, the field of active vision, underpinned by the theory of information value (Chapter 16), is concerned with exactly the problem of directing sensors, and already some robots have incorporated the theoretical results obtained.

In sum, many of the issues Dreyfus has focused on—background commonsense knowledge, the qualification problem, uncertainty, learning, compiled forms of decision making, the importance of considering situated agents rather than disembodied inference engines—have by now been incorporated into standard intelligent agent design. In our view, this is evidence of AI's progress, not of its impossibility.

26.2 STRONG AI: CAN MACHINES REALLY THINK?

Many philosophers have claimed that a machine that passes the Turing Test would still not be *actually* thinking, but would be only a *simulation* of thinking. Again, the objection was foreseen by Turing. He cites a speech by Professor Geoffrey Jefferson (1949):

Not until a machine could write a sonnet or compose a concerto because of thoughts and emotions felt, and not by the chance fall of symbols, could we agree that machine equals brain—that is, not only write it but know that it had written it.

Turing calls this the argument from consciousness—the machine has to be aware of its own mental states and actions. While consciousness is an important subject, Jefferson's key point

actually relates to phenomenology, or the study of direct experience—the machine has to actually feel emotions. Others focus on intentionality—that is, the question of whether the machine's purported beliefs, desires, and other representations are actually "about" something in the real world.

Turing's response to the objection is interesting. He could have presented reasons that machines can in fact be conscious (or have phenomenology, or have intentions). Instead, he maintains that the question is just as ill-defined as asking, "Can machines think?" Besides, why should we insist on a higher standard for machines than we do for humans? After all, in ordinary life we never have any direct evidence about the internal mental states of other humans. Nevertheless, Turing says, "Instead of arguing continually over this point, it is usual to have the polite convention that everyone thinks."

Turing argues that Jefferson would be willing to extend the polite convention to machines if only he had experience with ones that act intelligently. He cites the following dialog, which has become such a part of AI's oral tradition that we simply have to include it:

HUMAN: In the first line of your sonnet which reads "shall I compare thee to a summer's day," would not a "spring day" do as well or better?

MACHINE: It wouldn't scan.

HUMAN: How about "a winter's day." That would scan all right.

MACHINE: Yes, but nobody wants to be compared to a winter's day.

HUMAN: Would you say Mr. Pickwick reminded you of Christmas?

MACHINE: In a way.

HUMAN: Yet Christmas is a winter's day, and I do not think Mr. Pickwick would mind the comparison.

MACHINE: I don't think you're serious. By a winter's day one means a typical winter's day, rather than a special one like Christmas.

Turing concedes that the question of consciousness is a difficult one, but denies that it has much relevance to the practice of AI: "I do not wish to give the impression that I think there is no mystery about consciousness . . . But I do not think these mysteries necessarily need to be solved before we can answer the question with which we are concerned in this paper." We agree with Turing—we are interested in creating programs that behave intelligently, not in whether someone else pronounces them to be real or simulated. On the other hand, many philosophers are keenly interested in the question. To help understand it, we will consider the question of whether other artifacts are considered real.

In 1848, artificial urea was synthesized for the first time, by Frederick Wohler. This was important because it proved that organic and inorganic chemistry could be united, a question that had been hotly debated. Once the synthesis was accomplished, chemists agreed that artificial urea **was** urea, because it had all the right physical properties. Similarly, artificial sweeteners are undeniably sweeteners, and artificial insemination (the other AI) is undeniably insemination. On the other hand, artificial flowers are not flowers, and Daniel Dennett points out that artificial Chateau Latour wine would not be Chateau Latour wine, even if it was chemically indistinguishable, simply because it was not made in the right place in the right way. Nor is an artificial Picasso painting a Picasso painting, no matter what it looks like.

We can conclude that in some cases, the behavior of an artifact is important, while in others it is the artifact's pedigree that matters. Which one is important in which case seems to be a matter of convention. But for artificial minds, there is no convention, and we are left to rely on intuitions. The philosopher John Searle (1980) has a strong one:

No one supposes that a computer simulation of a storm will leave us all wet ... Why on earth would anyone in his right mind suppose a computer simulation of mental processes actually had mental processes? (pp. 37–38)

While it is easy to agree that computer simulations of storms do not make us wet, it is not clear how to carry this analogy over to computer simulations of mental processes. After all, a Hollywood simulation of a storm using sprinklers and wind machines *does* make the actors wet. Most people are comfortable saying that a computer simulation of addition is addition, and a computer simulation of a chess game is a chess game. Are mental processes more like storms, or more like addition or chess? Like Chateau Latour and Picasso, or like urea? That all depends on your theory of mental states and processes.

FUNCTIONALISM

The theory of **functionalism** says that a mental state is any intermediate causal condition between input and output. Under functionalist theory, any two systems with isomorphic causal processes would have the same mental states. Therefore, a computer program could have the same mental states as a person. Of course, we have not yet said what "isomorphic" really means, but the assumption is that there is some level of abstraction below which the specific implementation does not matter; as long as the processes are isomorphic down to the this level, the same mental states will occur.

BIOLOGICAL NATURALISM

In contrast, the **biological naturalism** theory says that mental states are high-level emergent features that are caused by low-level neurological processes *in the neurons*, and it is the (unspecified) properties of the neurons that matter. Thus, mental states cannot be duplicated just on the basis of some program having the same functional structure with the same input–output behavior; we would require that the program be running on an architecture with the same causal power as neurons. The theory does not say why neurons have this causal power, nor what other physical instantiations might or might not have it.

To investigate these two viewpoints we will first look at one of the oldest problems in the philosophy of mind, and then turn to three thought experiments.

MIND-BODY PROBLEM

The mind–body problem

The **mind–body problem** asks how mental states and processes are related to bodily (specifically, brain) states and processes. As if that wasn't hard enough, we will generalize the problem to the "mind–architecture" problem, to allow us to talk about the possibility of machines having minds.

DUALISM
MONISM
MATERIALISM

Why is the mind–body problem a problem? The first difficulty goes back to René Descartes, who considered how an immortal soul interacts with a mortal body and concluded that the soul and body are two distinct types of things—a **dualist** theory. The **monist** theory, often called **materialism**, holds that there are no such things as immaterial souls; only material objects. Consequently, mental states—such as being in pain, knowing that one is riding a horse, or believing that Vienna is the capital of Austria—are brain states. John Searle pithily

sums up the idea with the slogan, "*Brains cause minds.*"

The materialist must face at least two serious obstacles. The first is the problem of **free will**: how can it be that a purely physical mind, whose every transformation is governed strictly by the laws of physics, still retains any freedom of choice? Most philosophers regard this problem as requiring a careful reconstitution of our naive notion of free will, rather than presenting any threat to materialism. The second problem concerns the general issue of **consciousness** (and related, but not identical, questions of **understanding** and **self-awareness**). Put simply, why is it that it *feels* like something to have certain brain states, whereas it presumably does not feel like anything to have other physical states (e.g., being a rock).

To begin to answer such questions, we need ways to talk about brain states at levels more abstract than specific configurations of all the atoms of the brain of a particular person at a particular time. For example, as I think about the capital of Austria, my brain undergoes myriad tiny changes from one picosecond to the next, but these do not constitute a *qualitative* change in brain state. To account for this, we need a notion of brain state *types*, under which we can judge whether two brain states belong to the same or different types. Various authors have various positions on what one means by *type* in this case. Almost everyone believes that if one takes a brain and replaces some of the carbon atoms by a new set of carbon atoms,² the mental state will not be affected. This is a good thing because real brains are continually replacing their atoms through metabolic processes, and yet this in itself does not seem to cause major mental upheavals.

Now let's consider a particular kind of mental state: the **propositional attitudes** (first discussed in Chapter 10), which are also known as **intentional states**. These are states, such as believing, knowing, desiring, fearing, and so on, that refer to some aspect of the external world. For example, the belief that Vienna is the capital of Austria is a belief about a particular city and its status. We will be asking whether it is possible for computers to have intentional states, so it helps to understand how to characterize such states. For example, one might say that the mental state in which I desire a hamburger differs from the state in which I desire a pizza because hamburger and pizza are different things in the real world. That is to say, intentional states have a necessary connection to their objects in the external world. On the other hand, we argued just a few paragraphs back that mental states are brain states; hence the identity or non-identity of mental states should be determined by staying completely "inside the head," without reference to the real world. To examine this dilemma we turn to a thought experiment that attempts to separate intentional states from their external objects.

The "brain in a vat" experiment

Imagine, if you will, that your brain was removed from your body at birth and placed in a marvelously engineered vat. The vat sustains your brain, allowing it to grow and develop. At the same time, electronic signals are fed to your brain from a computer simulation of an entirely fictitious world, and motor signals from your brain are intercepted and used to modify the simulation as appropriate.³ Then the brain could have the mental state

² Perhaps even atoms of a different isotope of carbon, as is sometimes done in brain-scanning experiments.

³ This situation may be familiar to those who have seen the 1999 film, *The Matrix*.

DyingFor(Me, Hamburger) even though it has no body to feel hunger and no taste buds to experience taste, and there may be no hamburger in the real world. In that case, would this be the same mental state as one held by a brain in a body?

WIDE CONTENT

One way to resolve the dilemma is to say that the content of mental states can be interpreted from two different points of view. The "**wide content**" view interprets it from the point of view of an omniscient outside observer with access to the whole situation, who can distinguish differences in the world. So under wide content the brain-in-a-vat beliefs are different from those of a "normal" person. **Narrow content** considers only the internal subjective point of view, and under this view the beliefs would all be the same.

NARROW CONTENT

QUALIA

The belief that a hamburger is delicious has a certain intrinsic nature—there is something that it is like to have this belief. Now we get into the realm of **qualia**, or intrinsic experiences (from the Latin word meaning, roughly, "such things"). Suppose, through some accident of retinal and neural wiring, that person X experiences as red the color that person Y perceives as green, and vice-versa. Then when both see the same traffic light they will act the same way, but the **experience** they have will be in some way different. Both may agree that the name for their experience is "the light is red," but the experiences feel different. It is not clear whether that means they are the same or different mental states.

We now turn to another thought experiment that gets at the question of whether physical objects other than human neurons can have mental states.

The brain prosthesis experiment

The brain prosthesis experiment was introduced in the mid-1970s by Clark Glymour and was touched on by John Searle (1980), but is most commonly associated with the work of Hans Moravec (1988). It goes like this: Suppose neurophysiology has developed to the point where the input–output behavior and connectivity of all the neurons in the human brain are perfectly understood. Suppose further that we can build microscopic electronic devices that mimic this behavior and can be smoothly interfaced to neural tissue. Lastly, suppose that some miraculous surgical technique can replace individual neurons with the corresponding electronic devices without interrupting the operation of the brain as a whole. The experiment consists of gradually replacing all the neurons in someone's head with electronic devices and then reversing the process to return the subject to his or her normal biological state.

We are concerned with both the external behavior and the internal experience of the subject, during and after the operation. By the definition of the experiment, the subject's external behavior must remain unchanged compared with what would be observed if the operation were not carried out.⁴ Now although the presence or absence of consciousness cannot easily be ascertained by a third party, the subject of the experiment ought at least to be able to record any changes in his or her own conscious experience. Apparently, there is a direct clash of intuitions as to what would happen. Moravec, a robotics researcher and functionalist, is convinced his consciousness would remain unaffected. Searle, a philosopher and biological naturalist, is equally convinced his consciousness would vanish:

⁴ One can imagine using an identical "control" subject who is given a placebo operation, so that the two behaviors can be compared.

You find, to your total amazement, that you are indeed losing control of your external behavior. You find, for example, that when doctors test your vision, you hear them say "We are holding up a red object in front of you; please tell us what you see." You want to cry out "I can't see anything. I'm going totally blind." But you hear your voice saying in a way that is completely out of your control, "I see a red object in front of me." ... [Y]our conscious experience slowly shrinks to nothing, while your externally observable behavior remains the same. (Searle, 1992)

But one can do more than argue from intuition. First, note that, in order for the external behavior to remain the same while the subject gradually becomes unconscious, it must be the case that the subject's volition is removed instantaneously and totally; otherwise the shrinking of awareness would be reflected in external behavior—"Help, I'm shrinking!" or words to that effect. This instantaneous removal of volition as a result of gradual neuron-at-a-time replacement seems an unlikely claim to have to make.

Second, consider what happens if we do ask the subject questions concerning his or her conscious experience during the period when no real neurons remain. By the conditions of the experiment, we will get responses such as "I feel fine. I must say I'm a bit surprised because I believed Searle's argument." Or we might poke the subject with a pointed stick and observe the response, "Ouch, that hurt." Now, in the normal course of affairs, the skeptic can dismiss such outputs from AI programs as mere contrivances. Certainly, it is easy enough to use a rule such as "If sensor 12 reads 'High' then output 'Ouch.'" But the point here is that, because we have replicated the functional properties of a normal human brain, we assume that the electronic brain contains no such contrivances. Then we must have an explanation of the manifestations of consciousness produced by the electronic brain that appeals only to the functional properties of the neurons. *And this explanation must also apply to the real brain, which has the same functional properties.* There are, it seems, only two possible conclusions:

1. The causal mechanisms of consciousness that generate these kinds of outputs in normal brains are still operating in the electronic version, which is therefore conscious.
2. The conscious mental events in the normal brain have no causal connection to behavior, and are missing from the electronic brain, which is therefore not conscious.

EPIPHENOMENON

Although we cannot rule out the second possibility, it reduces consciousness to what philosophers call an **epiphenomenal** role—something that happens, but casts no shadow, as it were, on the observable world. Furthermore, if consciousness is indeed epiphenomenal, then the brain must contain a second, unconscious mechanism that is responsible for the "Ouch."

Third, consider the situation after the operation has been reversed and the subject has a normal brain. Once again, the subject's external behavior must, by definition, be as if the operation had not occurred. In particular, we should be able to ask, "What was it like during the operation? Do you remember the pointed stick?" The subject must have accurate memories of the actual nature of his or her conscious experiences, including the qualia, despite the fact that, according to Searle there were no such experiences.

Searle might reply that we have not defined the experiment properly. If the real neurons are, say, put into suspended animation between the time they are extracted and the time they are replaced in the brain, then of course they will not "remember" the experiences during

the operation. To deal with this eventuality, we need to make sure that the neurons' state is updated to reflect the internal state of the artificial neurons they are replacing. If the supposed "nonfunctional" aspects of the real neurons then result in functionally different behavior from that observed with artificial neurons still in place, then we have a simple *reductio ad absurdum*, because that would mean that the artificial neurons are not functionally equivalent to the real neurons. (See Exercise 26.3 for one possible rebuttal to this argument.)

Patricia Churchland (1986) points out that the functionalist arguments that operate at the level of the neuron can also operate at the level of any larger functional unit—a clump of neurons, a mental module, a lobe, a hemisphere, or the whole brain. That means that if you accept the notion that the brain prosthesis experiment shows that the replacement brain is conscious, then you should also believe that consciousness is maintained when the entire brain is replaced by a circuit that maps from inputs to outputs via a huge lookup table. This is disconcerting to many people (including Turing himself), who have the intuition that lookup tables are not conscious—or at least, that the conscious experiences generated during table lookup are not the same as those generated during the operation of a system that might be described (even in a simple-minded, computational sense) as accessing and generating beliefs, introspections, goals, and so on. This would suggest that the brain prosthesis experiment cannot use whole-brain-at-once replacement if it is to be effective in guiding intuitions, but it does not mean that it must use one-atom-at-a-time replacement as Searle have us believe.

The Chinese room

Our final thought experiment is perhaps the most famous of all. It is due to John Searle (1980), who describes a hypothetical system that is clearly running a program and passes the Turing Test, but that equally clearly (according to Searle) does not *understand* anything of its inputs and outputs. His conclusion is that running the appropriate program (i.e., having the right outputs) is not a *sufficient* condition for being a mind.

The system consists of a human, who understands only English, equipped with a rule book, written in English, and various stacks of paper, some blank, some with indecipherable inscriptions. (The human therefore plays the role of the CPU, the rule book is the program, and the stacks of paper are the storage device.) The system is inside a room with a small opening to the outside. Through the opening appear slips of paper with indecipherable symbols. The human finds matching symbols in the rule book, and follows the instructions. The instructions may include writing symbols on new slips of paper, finding symbols in the stacks, rearranging the stacks, and so on. Eventually, the instructions will cause one or more symbols to be transcribed onto a piece of paper that is passed back to the outside world.

So far, so good. But from the outside, we see a system that is taking input in the form of Chinese sentences and generating answers in Chinese that are as obviously "intelligent" as those in the conversation imagined by Turing.⁵ Searle then argues as follows: the person in the room does not understand Chinese (given). The rule book and the stacks of paper, being

⁵ The fact that the stacks of paper might well be larger than the entire planet and the generation of answers would take millions of years has no bearing on the logical structure of the argument. One aim of philosophical training is to develop a finely honed sense of which objections are germane and which are not.



just pieces of paper, do not understand Chinese. Therefore, there is no understanding of Chinese going on. *Hence, according to Searle, running the right program does not necessarily generate understanding.*

Like Turing, Searle considered and attempted to rebuff a number of replies to his argument. Several commentators, including John McCarthy and Robert Wilensky, proposed what Searle calls the systems reply. The objection is that, although one can ask if the human in the room understands Chinese, this is analogous to asking if the CPU can take cube roots. In both cases, the answer is no, and in both cases, according to the systems reply, the entire system *does* have the capacity in question. Certainly, if one asks the Chinese room whether it understands Chinese, the answer would be affirmative (in fluent Chinese). By Turing's polite convention, this should be enough. Searle's response is to reiterate the point that the understanding is not in the human and cannot be in the paper, so there cannot be any understanding. He further suggests that one could imagine the human memorizing the rule book and the contents of all the stacks of paper, so that there would be nothing to have understanding *except* the human; and again, when one asks the human (in English), the reply will be in the negative.

Now we are down to the real issues. The shift from paper to memorization is a red herring, because both forms are simply physical instantiations of a running program. The real claim made by Searle rests upon the following four axioms (Searle, 1990):

1. Computer programs are formal, syntactic entities.
2. Minds have mental contents, or semantics.
3. Syntax by itself is not sufficient for semantics.
4. Brains cause minds.

From the first three axioms he concludes that programs are not sufficient for minds. In other words, an agent running a program might be a mind, but it is not necessarily a mind just by virtue of running the program. From the fourth axiom he concludes "Any other system capable of causing minds would have to have causal powers (at least) equivalent to those of brains." From there he infers that any artificial brain would have to duplicate the causal powers of brains, not just run a particular program, and that human brains do not produce mental phenomena solely by virtue of running a program.

The conclusions that programs are not sufficient for minds *does* follow from the axioms, if you are generous in interpreting them. But the conclusion is unsatisfactory—all Searle has shown is that if you explicitly deny functionalism (that is what his axiom (3) does) then you can't necessarily conclude that non-brains are minds. This is reasonable enough, so the whole argument comes down to whether axiom (3) can be accepted. According to Searle, the point of the Chinese room argument is to provide intuitions for axiom (3). But the reaction to his argument shows that it provides intuitions only to those who were already inclined to accept the idea that mere programs cannot generate true understanding.

To reiterate, the aim of the Chinese Room argument is to refute strong AI—the claim that running the right sort of program necessarily results in a mind. It does this by exhibiting an apparently intelligent system running the right sort of program that is, according to Searle, *demonstrably* not a mind. Searle appeals to intuition, not proof, for this part: just look at the room; what's there to be a mind? But one could make the same argument about the brain:

just look at this collection of cells (or of atoms), blindly operating according to the laws of biochemistry (or of physics)—what's there to be a mind? Why can a hunk of brain be a mind while a hunk of liver cannot?

Furthermore, when Searle admits that materials other than neurons could in principle be a mind, he weakens his argument even further, for two reasons: first, one has only Searle's intuitions (or one's own) to say that the Chinese room is not a mind, and second, even if we decide the room is not a mind, that tells us nothing about whether a program running on some other physical medium (including a computer) might be a mind.

Searle allows the logical possibility that the brain is actually implementing an AI program of the traditional sort—but the same program running on the wrong kind of machine would not be a mind. Searle has denied that he believes that "machines cannot have minds," rather, he asserts that some machines *do* have minds—humans are biological machines with minds. We are left without much guidance as to what types of machines do or do not qualify.

26.3 THE ETHICS AND RISKS OF DEVELOPING ARTIFICIAL INTELLIGENCE

So far, we have concentrated on whether we *can* develop AI, but we must also consider whether we *should*. If the effects of AI technology are more likely to be negative than positive, then it would be the moral responsibility of workers in the field to redirect their research. Many new technologies have had unintended negative side-effects: the internal combustion engine brought air pollution and the paving-over of paradise; nuclear fission brought Chernobyl, Three Mile Island, and the threat of global destruction. All scientists and engineers face ethical considerations of how they should act on the job, what projects should or should not be done, and how they should be handled. There is even a handbook on the *Ethics of Computing* (Berleur and Brunnstein, 2001). AI, however, seems to pose some fresh problems beyond that of, say, building bridges that don't fall down:

- People might lose their jobs to automation.
- People might have too much (or too little) leisure time.
- People might lose their sense of being unique.
- People might lose some of their privacy rights.
- The use of AI systems might result in a loss of accountability.
- The success of AI might mean the end of the human race.

We will look at each issue in turn.

People might lose their jobs to automation. The modern industrial economy has become dependent on computers in general, and select AI programs in particular. For example, much of the economy, especially in the United States, depends on the availability of consumer credit. Credit card applications, charge approvals, and fraud detection are now done by AI programs. One could say that thousands of workers have been displaced by these AI programs, but in fact if you took away the AI programs these jobs would not exist, because human labor would add an unacceptable cost to the transactions. So far, automation via AI

technology has created more jobs than it has eliminated, and has created more interesting, higher-paying jobs. Now that the canonical AI program is an "intelligent agent" designed to assist a human, loss of jobs is less of a concern than it was when AI focused on "expert systems" designed to replace humans.

People might have too much (or too little) leisure time. Alvin Toffler wrote in *Future Shock* (1970), "The work week has been cut by 50 percent since the turn of the century. It is not out of the way to predict that it will be slashed in half again by 2000." Arthur C. Clarke (1968b) wrote that people in 2001 might be "faced with a future of utter boredom, where the main problem in life is deciding which of several hundred TV channels to select." The only one of these predictions that has come close to panning out is the number of TV channels (Springsteen, 1992). Instead, people working in knowledge-intensive industries have found themselves part of an integrated computerized system that operates 24 hours a day; to keep up, they have been forced to work *longer* hours. In an industrial economy, rewards are roughly proportional to the time invested; working 10% more would tend to mean a 10% increase in income. In an information economy marked by high-bandwidth communication and easy replication of intellectual property (what Frank and Cook (1996) call the "Winner-Take-All Society"), there is a large reward for being slightly better than the competition; working 10% more could mean a 100% increase in income. So there is increasing pressure on everyone to work harder. AI increases the pace of technological innovation and thus contributes to this overall trend, but AI also holds the promise of allowing us to take some time off and let our automated agents handle things for a while.

People might lose their sense of being unique. In *Computer Power and Human Reason*, Weizenbaum (1976), the author of the ELIZA program, points out some of the potential threats that AI poses to society. One of Weizenbaum's principal arguments is that AI research makes possible the idea that humans are automata—an idea that results in a loss of autonomy or even of humanity. We note that the idea has been around much longer than AI, going back at least to *L'Homme Machine* (La Mettrie, 1748). We also note that humanity has survived other setbacks to our sense of uniqueness: *De Revolutionibus Orbium Coelestium* (Copernicus, 1543) moved the Earth away from the center of the solar system and *Descent of Man* (Darwin, 1871) put *Homo sapiens* at the same level as other species. AI, if widely successful, may be at least as threatening to the moral assumptions of 21st-century society as Darwin's theory of evolution was to those of the 19th century.

People might lose some of their privacy rights. Weizenbaum also pointed out that speech recognition technology could lead to widespread wiretapping, and hence to a loss of civil liberties. He didn't foresee a world with terrorist threats that would change the balance of how much surveillance people are willing to accept, but he did correctly recognize that AI has the potential to mass-produce surveillance. His prediction may have come true: the U.S. government's classified Echelon system "consists of a network of listening posts, antenna fields, and radar stations; the system is backed by computers that use language translation, speech recognition, and keyword searching to automatically sift through telephone, email, fax, and telex traffic."⁶ Some accept that computerization leads to a loss of privacy—Sun

⁶ See "Eavesdropping on Europe," Wired news, 913011998, and cited EU reports.

Microsystems CEO Scott McNealy has said "You have zero privacy anyway. Get over it." Others disagree: Judge Louis Brandeis wrote in 1890, "Privacy is the most comprehensive of all rights . . . the right to one's personality."

The use of AI systems might result in a loss of accountability. In the litigious atmosphere that prevails in the United States, legal liability becomes an important issue. When a physician relies on the judgment of a medical expert system for a diagnosis, who is at fault if the diagnosis is wrong? Fortunately, due in part to the growing influence of decision-theoretic methods in medicine, it is now accepted that negligence cannot be shown if the physician performs medical procedures that have high *expected utility*, even if the *actual* result is catastrophic for the patient. The question should therefore be "Who is at fault if the diagnosis is unreasonable?" So far, courts have held that medical expert systems play the same role as medical textbooks and reference books; physicians are responsible for understanding the reasoning behind any decision and for using their own judgment in deciding whether to accept the system's recommendations. In designing medical expert systems as agents, therefore, the actions should be thought of not as directly affecting the patient but as influencing the physician's behavior. If expert systems become reliably more accurate than human diagnosticians, doctors might become legally liable if they *don't* use the recommendations of an expert system. Gawande (2002) explores this premise.

Similar issues are beginning to arise regarding the use of intelligent agents on the Internet. Some progress has been made in incorporating constraints into intelligent agents so that they cannot, for example, damage the files of other users (Weld and Etzioni, 1994). The problem is magnified when money changes hands. If monetary transactions are made "on one's behalf" by an intelligent agent, is one liable for the debts incurred? Would it be possible for an intelligent agent to have assets itself and to perform electronic trades on its own behalf? So far, these questions do not seem to be well understood. To our knowledge, no program has been granted legal status as an individual for the purposes of financial transactions; at present, it seems unreasonable to do so. Programs are also not considered to be "drivers" for the purposes of enforcing traffic regulations on real highways. In California law, at least, there do not seem to be any legal sanctions to prevent an automated vehicle from exceeding the speed limits, although the designer of the vehicle's control mechanism would be liable in the case of an accident. As with human reproductive technology, the law has yet to catch up with the new developments.

The success of AI might mean the end of the human race. Almost any technology has the potential to cause harm in the wrong hands, but with AI and robotics, we have the new problem that the wrong hands might belong to the technology itself. Countless science fiction stories have warned about robots or robot-human cyborgs running amok. Early examples include Mary Shelley's *Frankenstein, or the Modern Prometheus* (1818)⁷ and Karel Capek's play *R.U.R* (1921), in which robots conquer the world. In movies, we have *The Terminator* (1984), which combines the clichés of robots-conquer-the-world with time travel, and *The Matrix* (1999), which combines robots-conquer-the-world with brain-in-a-vat.

⁷ As a young man, Charles Babbage was influenced by reading *Frankenstein*.

For the most part, it seems that robots are the protagonists of so many conquer-the-world stories because they represent the unknown, just like the witches and ghosts of tales from earlier eras. Do they pose a more credible threat than witches and ghosts? If robots are properly designed as agents that adopt their owner's goals, then they probably do not: robots that derive from incremental advances over current designs will serve, not conquer. Humans use their intelligence in aggressive ways because humans have some innately aggressive tendencies, due to natural selection. But the machines we build need not be innately aggressive, unless we decide to build them that way. On the other hand, it is possible that computers will achieve a sort of conquest by serving and becoming indispensable, just as automobiles have in a sense conquered the industrialized world. One scenario deserves further consideration. I. J. Good wrote (1965),

Let an ultraintelligent machine be defined as a machine that can far surpass all the intellectual activities of any man however clever. Since the design of machines is one of these intellectual activities, an ultraintelligent machine could design even better machines; there would then unquestionably be an "intelligence explosion," and the intelligence of man would be left far behind. Thus the first ultraintelligent machine is the *last* invention that man need ever make, provided that the machine is docile enough to tell us how to keep it under control.

TECHNOLOGICAL SINGULARITY

The "intelligence explosion" has also been called the technological singularity by mathematics professor and science fiction author Vernor Vinge, who writes (1993), "Within thirty years, we will have the technological means to create superhuman intelligence. Shortly after, the human era will be ended." Good and Vinge (and many others) correctly note that the curve of technological progress is growing exponentially at present (consider Moore's Law). However, it is quite a step to extrapolate that the curve will continue on to a singularity of near-infinite growth. So far, every other technology has followed an S-shaped curve, where the exponential growth eventually tapers off.

Vinge is concerned and scared about the coming singularity, but other computer scientists and futurists relish it. Hans Moravec's *Robot: Mere Machine to Transcendent Mind* predicts that robots will match human intelligence in 50 years and then exceed it. He writes,

Rather quickly, they could displace us from existence. I'm not as alarmed as many by the latter possibility, since I consider these future machines our progeny, "mind children" built in our image and likeness, ourselves in more potent form. Like biological children of previous generations, they will embody humanity's best hope for a long-term future. It behooves us to give them every advantage, and to bow out when we can no longer contribute. (Moravec, 2000)

TRANSHUMANISM

Ray Kurzweil, in *The Age of Spiritual Machines* (2000), predicts that by the year 2099 there will be "a strong trend toward a merger of human thinking with the world of machine intelligence that the human species initially created. There is no longer any clear distinction between humans and computers." There is even a new word—transhumanism—for the active social movement that looks forward to this future. Suffice it to say that such issues present a challenge for most moral theorists, who take the preservation of human life and the human species to be a good thing.

Finally, let us consider the robot's point of view. If robots become conscious, then to treat them as mere "machines" (e.g., to take them apart) might be immoral. Robots also must themselves act morally—we would need to program them with a theory of what is right and wrong. Science fiction writers have addressed the issue of robot rights and responsibilities, starting with Isaac Asimov (1942). The well-known movie A.I. (Spielberg, 2001) was based on a story by Brian Aldiss about an intelligent robot who was programmed to believe that he was human and fails to understand his eventual abandonment by his owner—mother. The story (and the movie) convince one of the need for a civil rights movement for robots.

26.4 SUMMARY

This chapter has addressed the following issues:

- Philosophers use the term **weak AI** for the hypothesis that machines could possibly behave intelligently, and **strong AI** for the hypothesis that such machines would count as having actual minds (as opposed to simulated minds).
- Alan Turing rejected the question "Can machines think?" and replaced it with a behavioral test. He anticipated many objections to the possibility of thinking machines. Few AI researchers pay attention to the Turing test, preferring to concentrate on their systems' performance on practical tasks, rather than the ability to imitate humans.
- There is general agreement in modern times that mental states are brain states.
- Arguments for and against strong AI are inconclusive. Few mainstream AI researchers believe that anything significant hinges on the outcome of the debate.
- Consciousness remains a mystery.
- We identified six potential threats to society posed by AI and related technology. We concluded that some of the threats are either unlikely or differ little from threats posed by other, "unintelligent" technologies. One threat in particular is worthy of further consideration: that ultraintelligent machines might lead to a future that is very different from today—we may not like it, and at that point we may not have a choice. Such considerations lead inevitably to the conclusion that we must weigh carefully, and soon, the possible consequences of AI research for the future of the human race.

BIBLIOGRAPHICAL AND HISTORICAL NOTES

The nature of the mind has been a standard topic of philosophical theorizing from ancient times to the present. In the *Phaedo*, Plato specifically considered and rejected the idea that the mind could be an "attunement" or pattern of organization of the parts of the body, a viewpoint that approximates the functionalist viewpoint in modern philosophy of mind. He decided instead that the mind had to be an immortal, immaterial soul, separable from the body and different in substance—the viewpoint of dualism. Aristotle distinguished a variety

of souls (Greek $\psi v\chi\eta$) in living things, some of which, at least, he described in a functionalist manner. (See Nussbaum (1978) for more on Aristotle's functionalism.)

Descartes is notorious for his dualistic view of the human mind, but ironically his historical influence was toward mechanism and materialism. He explicitly conceived of animals as automata, and he anticipated the Turing test, writing "it is not conceivable [that a machine] should produce different arrangements of words so as to give an appropriately meaningful answer to whatever is said in its presence, as even the dullest of men can do" (Descartes, 1637). Descartes's spirited defense of the animals-as-automata viewpoint actually had the effect of making it easier to conceive of humans as automata as well, even though he himself did not take this step. The book *L'Homme Machine* or *Man a Machine* (La Mettrie, 1748) did explicitly argue that humans are automata.

Modern analytic philosophy has typically accepted materialism (often in the form of the brain-state **identity theory** (Place, 1956; Armstrong, 1968), which asserts that mental states are identical with brain states), but has been much more divided on functionalism, the machine analogy for the human mind, and the question of whether machines can literally think. A number of early philosophical responses to Turing's (1950) "Computing Machinery and Intelligence," for example, Scriven (1953), attempted to deny that it was even *meaningful* to say that machines could think, on the ground that such an assertion violated the meaning of the word. Scriven, at least, had retracted this view by 1963; see his addendum to a reprint of his article (Anderson, 1964). The computer scientist Edsger Dijkstra said that "The question of whether a computer can think is no more interesting than the question of whether a submarine can swim." Ford and Hayes (1995) argue that the Turing Test is not helpful for AI.

Functionalism is the philosophy of mind most naturally suggested by AI, and critiques of functionalism often take the form of critiques of AI (as in the case of Searle). Following the classification used by Block (1980), we can distinguish varieties of functionalism. **Functional specification theory** (Lewis, 1966, 1980) is a variant of brain-state identity theory that selects the brain states that are to be identified with mental states on the basis of their functional role. **Functional state identity theory** (Putnam, 1960, 1967) is more closely based on a machine analogy. It identifies mental states not with *physical* brain states but with abstract computational states of the brain conceived expressly as a computing device. These abstract states are supposed to be independent of the specific physical composition of the brain, leading some to charge that functional state identity theory is a form of dualism!

Both the brain-state identity theory and the various forms of functionalism have come under attack from authors who claim that they do not account for the *qualia* or "what it's like" aspect of mental states (Nagel, 1974). Searle has focused instead on the alleged inability of functionalism to account for intentionality (Searle, 1980, 1984, 1992). Churchland and Churchland (1982) rebut both these types of criticism.

Eliminative materialism (Rorty, 1965; Churchland, 1979) differs from all other prominent theories in the philosophy of mind, in that it does not attempt to give an account of why our "folk psychology" or commonsense ideas about the mind are true, but instead rejects them as false and attempts to replace them with a purely scientific theory of the mind. In principle, this scientific theory could be given by classical AI, but in practice, eliminative materialists tend to lean on neuroscience and neural network research instead (Churchland,

1986), on the grounds that classical AI, especially "knowledge representation" research of the kind described in Chapter 10, tends to rely on the truth of folk psychology. Although the "intentional stance" viewpoint (Dennett, 1971) could be interpreted as functionalist, it should probably instead be regarded as a form of eliminative materialism, in that taking the "intentional stance" is not supposed to reflect any objective property of the agent toward whom the stance is taken. It should also be noted that it is possible to be an eliminative materialist about some aspects of mentality while analyzing others in some other way. For instance, Dennett (1978) is much more strongly eliminativist about qualia than about intentionality.

Sources for the main critics of weak AI were given in the chapter. Although it became fashionable in the post-neural-network era to deride symbolic approaches, not all philosophers are critical of GOFAI. Some are, in fact, ardent advocates and even practitioners. Zenon Wylshyn (1984) has argued that cognition can best be understood through a computational model, not only in principle but also as a way of conducting research at present, and has specifically rebutted Dreyfus's criticisms of the computational model of human cognition (Pylyshyn, 1974). Gilbert Harman (1983), in analyzing belief revision, makes connections with AI research on truth maintenance systems. Michael Bratman has applied his "belief-desire-intention" model of human psychology (Bratman, 1987) to AI research on planning (Bratman, 1992). At the extreme end of strong AI, Aaron Sloman (1978, p. xiii) has even described as "racialist" Joseph Weizenbaum's view (Weizenbaum, 1976) that hypothetical intelligent machines should not be regarded as persons.

The philosophical literature on minds, brains, and related topics is large and sometimes difficult to read without proper training in the terminology and methods of argument employed. The *Encyclopedia of Philosophy* (Edwards, 1967) is an impressively authoritative and very useful aide in this process. The *Cambridge Dictionary of Philosophy* (Audi, 1999) is a shorter and more accessible work, but main entries (such as "philosophy of mind") still span 10 pages or more. The *MIT Encyclopedia of Cognitive Science* (Wilson and Keil, 1999) covers the philosophy of mind as well as the biology and psychology of mind. General collections of articles on philosophy of mind, including functionalism and other viewpoints related to AI, are *Materialism and the Mind-Body Problem* (Rosenthal, 1971) and *Readings in the Philosophy of Psychology*, volume 1 (Block, 1980). Biro and Shahan (1982) present a collection devoted to the pros and cons of functionalism. Anthologies of articles dealing more specifically with the relation between philosophy and AI include *Minds and Machines* (Anderson, 1964), *Philosophical Perspectives in Artificial Intelligence* (Ringle, 1979), *Mind Design* (Haugeland, 1981), and *The Philosophy of Artificial Intelligence* (Boden, 1990). There are several general introductions to the philosophical "AI question" (Boden, 1977, 1990; Haugeland, 1985; Copeland, 1993). *The Behavioral and Brain Sciences*, abbreviated *BBS*, is a major journal devoted to philosophical and scientific debates about AI and neuroscience. Topics of ethics and responsibility in AI are covered in journals such as *AI and Society*, *Law, Computers and Artificial Intelligence*, and *Artificial Intelligence and Law*.

EXERCISES

26.1 Go through Turing's list of alleged "disabilities" of machines, identifying which have been achieved, which are achievable in principle by a program, and which are still problematic because they require conscious mental states.

26.2 Does a refutation of the Chinese room argument necessarily prove that appropriately programmed computers have mental states? Does an acceptance of the argument necessarily mean that computers cannot have mental states?

26.3 In the brain prosthesis argument, it is important to be able to restore the subject's brain to normal, such that its external behavior is as it would have been if the operation had not taken place. Can the skeptic reasonably object that this would require updating those neurophysiological properties of the neurons relating to conscious experience, as distinct from those involved in the functional behavior of the neurons?

26.4 Find and analyze an account in the popular media of one or more of the arguments to the effect that AI is impossible.

26.5 Attempt to write definitions of the terms "intelligence," "thinking," and "consciousness." Suggest some possible objections to your definitions.

26.6 Analyze the potential threats from AI technology to society. What threats are most serious, and how might they be combated? How do they compare to the potential benefits?

26.7 How do the potential threats from AI technology compare with those from other computer science technologies, and to bio-, nano-, and nuclear technologies?

26.8 Some critics object that AI is impossible, while others object that it is *too* possible, and that ultraintelligent machines pose a threat. Which of these objections do you think is more likely? Would it be a contradiction for someone to hold both positions?

27 AI: PRESENT AND FUTURE

In which we take stock of where we are and where we are going, this being a good thing to do before continuing.

In Part I, we proposed a unified view of AI as rational agent design. We showed that the design problem depends on the percepts and actions available to the agent, the goals that the agent's behavior should satisfy, and the nature of the environment. A variety of different agent designs are possible, ranging from reflex agents to fully deliberative, knowledge-based agents. Moreover, the components of these designs can have a number of different instantiations—for example, logical, probabilistic, or "neural." The intervening chapters presented the principles by which these components operate.

For all the agent designs and components, there has been tremendous progress both in our scientific understanding and in our technological capabilities. In this chapter, we stand back from the details and ask, *"Will all this progress lead to a general-purpose intelligent agent that can perform well in a wide variety of environments?* Section 27.1 looks at the components of an intelligent agent to assess what's known and what's missing. Section 27.2 does the same for the overall agent architecture. Section 27.3 asks whether "rational agent design" is the right goal in the first place. (The answer is, "Not really, but it's OK for now.") Finally, Section 27.4 examines the consequences of success in our endeavors.



27.1 AGENT COMPONENTS

Chapter 2 presented several agent designs and their components. To focus our discussion here, we will look at the utility-based agent, which we show again in Figure 27.1. This is the most general of our agent designs; we will also consider its extension with learning capabilities, as depicted in Figure 2.15.

Interaction with the environment through sensors and actuators: For much of the history of AI, this has been a glaring weak point. With a few honorable exceptions, AI systems were built in such a way that humans had to supply the inputs and interpret the outputs, while robotic systems focused on low-level tasks in which high-level reasoning and planning were largely absent. This was due in part to the great expense and engineering effort required

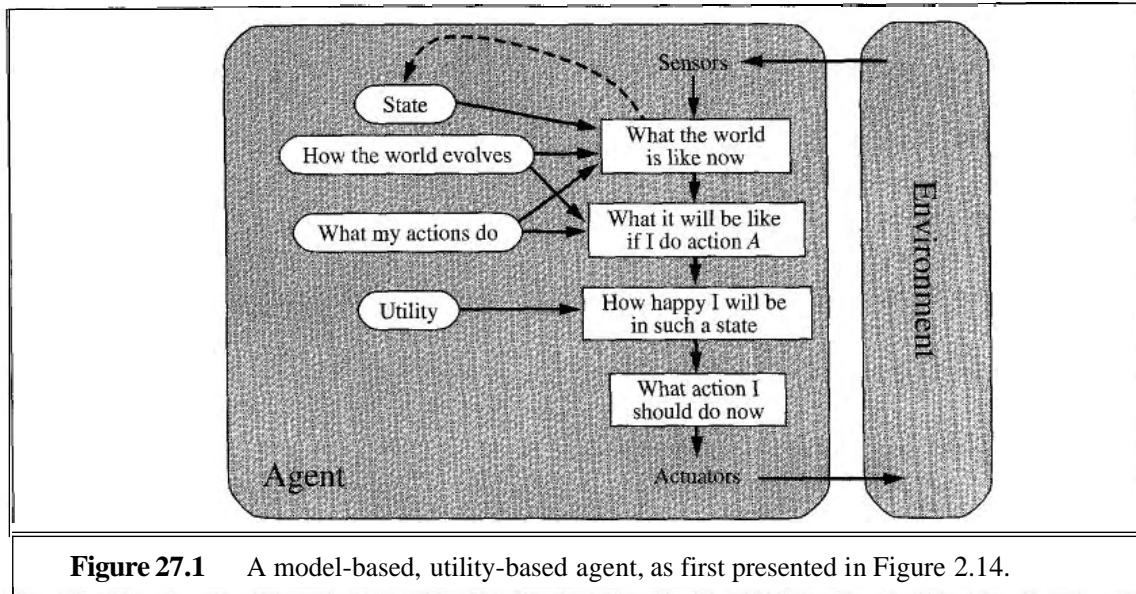


Figure 27.1 A model-based, utility-based agent, as first presented in Figure 2.14.

to get real robots to work at all. The situation has changed rapidly in recent years with the availability of ready-made programmable robots, such as the four-legged robots shown in Figure 25.4(b). These, in turn, have benefited from small, cheap, high-resolution CCD cameras and compact, reliable motor drives. MEMS (micro-electromechanical systems) technology has supplied miniaturized accelerometers and gyroscopes and is now producing actuators that will, for example, power an artificial flying insect. (It may also be possible to combine millions of MEMS actuators to produce very powerful macroscopic actuators.) For physical environments, then, AI systems no longer have a real excuse. Furthermore, an entirely new environment—the Internet—has become available.

Keeping track of the state of the world: This is one of the core capabilities required for an intelligent agent. It requires both perception and updating of internal representations. Chapter 7 described methods for keeping track of worlds described by propositional logic; Chapter 10 extended this to first-order logic; and Chapter 15 described filtering algorithms for tracking uncertain environments. These filtering tools are: required when real (and therefore imperfect) perception is involved. Current filtering and perception algorithms can be combined to do a reasonable job of reporting low-level predicates such as "the cup is on the table" but we have some way to go before they can report that "Dr. Russell is having a cup of tea with Dr. Norvig." Another problem is that, although approximate filtering algorithms can handle quite large environments, they are still essentially propositional—like propositional logic, they do not represent objects and relations explicitly. Chapter 14 explained how probability and first-order logic can be combined to solve this problem; we expect that the application of these ideas for tracking complex environments will yield huge benefits. Incidentally, as soon as we start talking about objects in an uncertain environment, we encounter identity uncertainty—we don't know which object is which. This problem has been largely ignored in logic-based AI, where it has generally been assumed that percepts incorporate constant symbols that identify the objects.

Projecting, evaluating, and selecting future courses of action: The basic knowledge representation requirements here are the same as for keeping track of the world; the primary difficulty is coping with courses of action—such as having a conversation or a cup of tea—that consist eventually of thousands or millions of primitive steps for a real agent. It is only by imposing **hierarchical structure** on behavior that we humans cope at all. Some of the planning algorithms in Chapter 12 use hierarchical representations and first-order representations to handle problems of this scale; on the other hand, the algorithms given in Chapter 17 for decision making under uncertainty are essentially using the same ideas as the state-based search algorithms of Chapter 3. There is clearly a great deal of work to do here, perhaps along the lines of recent developments in **hierarchical reinforcement learning**.

Utility as an expression of preferences: In principle, basing rational decisions on the maximization of expected utility is completely general and avoids many of the problems of purely goal-based approaches, such as conflicting goals and uncertain attainment. As yet, however, there has been very little work on constructing *realistic* utility functions—imagine, for example, the complex web of interacting preferences that must be understood by an agent operating as an office assistant for a human being. It has proven very difficult to decompose preferences over complex states in the same way that Bayes nets decompose beliefs over complex states. One reason may be that preferences over states are really *compiled* from preferences over state histories, which are described by **reward functions** (see Chapter 17). Even if the reward function is simple, the corresponding utility function may be very complex. This suggests that we take seriously the task of knowledge engineering for reward functions as a way of conveying to our agents what it is that we want them to do.

Learning: Chapters 18 to 20 described how learning in an agent can be formulated as inductive learning (supervised, unsupervised, or reinforcement-based) of the functions that constitute the various components of the agent. Very powerful logical and statistical techniques have been developed that can cope with quite large problems, often reaching or exceeding human capabilities in the identification of predictive patterns defined on a given vocabulary. On the other hand, machine learning has made very little progress on the important problem of constructing new representations at levels of abstraction higher than the input vocabulary. For example, how can an autonomous robot generate useful predicates such as *Office* and *Cafe* if they are not supplied to it by humans? Similar considerations apply to learning behavior—*HavingACupOfTea* is an important high-level action, but how does it get into an action library that initially contains much simpler actions such as *RaiseArm* and *Swallow*? Unless we understand such issues, we are faced with the daunting task of constructing large commonsense knowledge bases by hand.

27.2 AGENT ARCHITECTURES

It is natural to ask, "Which of the agent architectures in Chapter 2 should an agent use?" The answer is, "All of them!" We have seen that reflex responses are needed for situations in which time is of the essence, whereas knowledge-based deliberation allows the agent to

plan ahead. A complete agent must be able to do both, using a **hybrid architecture**. One important property of hybrid architectures is that the boundaries between different decision components are not fixed. For example, **compilation** continually converts declarative information at the deliberative level into more efficient representations, eventually reaching the reflex level—see Figure 27.2. (This is the purpose of explanation-based learning, as discussed in Chapter 19.) Agent architectures such as SOAR (Laird *et al.*, 1987) and THEO (Mitchell, 1990) have exactly this structure. Every time they solve a problem by explicit deliberation, they save away a generalized version of the solution for use by the reflex component. A less studied problem is the *reversal* of this process: when the environment changes, learned reflexes may no longer be appropriate and the agent must return to the deliberative level to produce new behaviors.

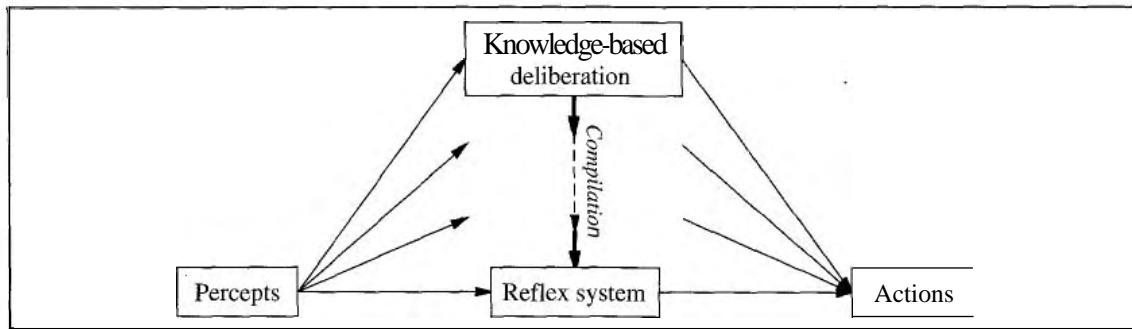


Figure 27.2 Compilation serves to convert deliberative decision making into more efficient, reflexive mechanisms.

Agents also need ways to control their own deliberations. They must be able to cease deliberating when action is demanded, and they must be able to use the time available for deliberation to execute the most profitable computations. For example, a taxi-driving agent that sees an accident ahead must decide in a split second either to brake or to take evasive action. It should also spend that split second thinking about the most important questions, such as whether the lanes to the left and right are clear and whether there is a large truck close behind, rather than worrying about wear and tear on the tires or where to pick up the next passenger. These issues are usually studied under the heading of **real-time AI**. As AI systems move into more complex domains, all problems will become real-time, because the agent will never have long enough to solve the decision problem exactly.

Clearly, there is a pressing need for methods that work in more general decision-making situations. Two promising techniques have emerged in recent years. The first involves the use of **anytime algorithms** (Dean and Boddy, 1988; Horvitz, 1987). An anytime algorithm is an algorithm whose output quality improves gradually over time, so that it has a reasonable decision ready whenever it is interrupted. Such algorithms are controlled by a metalevel decision procedure that assesses whether further computation is worthwhile. Iterative deepening search in game playing provides a simple example of an anytime algorithm. More complex systems, composed of many such algorithms working together, can also be constructed (Zilberstein and Russell, 1996). The second technique is **decision-theoretic metareasoning**