

```

function ADABoost(examples, L, K) returns a weighted-majority hypothesis
  inputs: examples, set of  $N$  labeled examples  $(x_1, y_1), \dots, (x_N, y_N)$ 
    L, a learning algorithm
    K, the number of hypotheses in the ensemble
  local variables: w, a vector of  $N$  example weights, initially  $1/N$ 
    h, a vector of  $K$  hypotheses
    z, a vector of  $K$  hypothesis weights

  for  $k = 1$  to  $K$  do
    h[ $k$ ]  $\leftarrow L(\text{examples}, \mathbf{w})$ 
    error  $\leftarrow 0$ 
    for  $j = 1$  to  $N$  do
      if h[ $k$ ]( $x_j$ )  $\neq y_j$  then error  $\leftarrow$  error + w[ $j$ ]
    for  $j = 1$  to  $N$  do
      if h[ $k$ ]( $x_j$ )  $= y_j$  then w[ $j$ ]  $\leftarrow \mathbf{w}[j] \cdot \text{error}/(1 - \text{error})$ 
    w  $\leftarrow \text{NORMALIZE}(\mathbf{w})$ 
    z[ $k$ ]  $\leftarrow \log(1 - \text{error})/\text{error}$ 
  return WEIGHTED-MAJORITY(h, z)

```

Figure 18.34 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**.

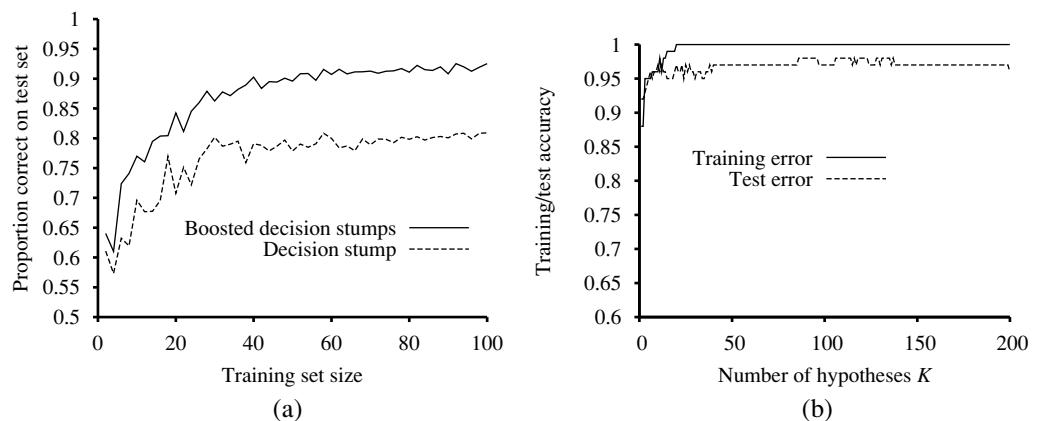


Figure 18.35 (a) Graph showing the performance of boosted decision stumps with $K = 5$ versus unboosted decision stumps on the restaurant data. (b) The proportion correct on the training set and the test set as a function of K , the number of hypotheses in the ensemble. Notice that the test set accuracy improves slightly even after the training accuracy reaches 1, i.e., after the ensemble fits the data exactly.

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.

18.10.1 Online Learning

So far, everything we have done in this chapter has relied on the assumption that the data are i.i.d. (independent and identically distributed). On the one hand, that is a sensible assumption: if the future bears no resemblance to the past, then how can we predict anything? On the other hand, it is too strong an assumption: it is rare that our inputs have captured all the information that would make the future truly independent of the past.

ONLINE LEARNING

In this section we examine what to do when the data are not i.i.d.; when they can change over time. In this case, it matters *when* we make a prediction, so we will adopt the perspective called **online learning**: an agent receives an input x_j from nature, predicts the corresponding y_j , and then is told the correct answer. Then the process repeats with x_{j+1} , and so on. One might think this task is hopeless—if nature is adversarial, all the predictions may be wrong. It turns out that there are some guarantees we can make.

RANDOMIZED
WEIGHTED
MAJORITY
ALGORITHM

Let us consider the situation where our input consists of predictions from a panel of experts. For example, each day a set of K pundits predicts whether the stock market will go up or down, and our task is to pool those predictions and make our own. One way to do this is to keep track of how well each expert performs, and choose to believe them in proportion to their past performance. This is called the **randomized weighted majority algorithm**. We can describe it more formally:

1. Initialize a set of weights $\{w_1, \dots, w_K\}$ all to 1.
2. Receive the predictions $\{\hat{y}_1, \dots, \hat{y}_K\}$ from the experts.
3. Randomly choose an expert k^* , in proportion to its weight: $P(k) = w_k / (\sum_{k'} w_{k'})$.
4. Predict \hat{y}_{k^*} .
5. Receive the correct answer y .
6. For each expert k such that $\hat{y}_k \neq y$, update $w_k \leftarrow \beta w_k$

REGRET

Here β is a number, $0 < \beta < 1$, that tells how much to penalize an expert for each mistake.

We measure the success of this algorithm in terms of **regret**, which is defined as the number of additional mistakes we make compared to the expert who, in hindsight, had the best prediction record. Let M^* be the number of mistakes made by the best expert. Then the number of mistakes, M , made by the random weighted majority algorithm, is bounded by¹⁵

$$M < \frac{M^* \ln(1/\beta) + \ln K}{1 - \beta}.$$

¹⁵ See (Blum, 1996) for the proof.

This bound holds for *any* sequence of examples, even ones chosen by adversaries trying to do their worst. To be specific, when there are $K = 10$ experts, if we choose $\beta = 1/2$ then our number of mistakes is bounded by $1.39M^* + 4.6$, and if $\beta = 3/4$ by $1.15M^* + 9.2$. In general, if β is close to 1 then we are responsive to change over the long run; if the best expert changes, we will pick up on it before too long. However, we pay a penalty at the beginning, when we start with all experts trusted equally; we may accept the advice of the bad experts for too long. When β is closer to 0, these two factors are reversed. Note that we can choose β to get asymptotically close to M^* in the long run; this is called **no-regret learning** (because the average amount of regret per trial tends to 0 as the number of trials increases).

Online learning is helpful when the data may be changing rapidly over time. It is also useful for applications that involve a large collection of data that is constantly growing, even if changes are gradual. For example, with a database of millions of Web images, you wouldn't want to train, say, a linear regression model on all the data, and then retrain from scratch every time a new image is added. It would be more practical to have an online algorithm that allows images to be added incrementally. For most learning algorithms based on minimizing loss, there is an online version based on minimizing regret. It is a bonus that many of these online algorithms come with guaranteed bounds on regret.

To some observers, it is surprising that there are such tight bounds on how well we can do compared to a panel of experts. To others, the really surprising thing is that when panels of human experts congregate—predicting stock market prices, sports outcomes, or political contests—the viewing public is so willing to listen to them pontificate and so unwilling to quantify their error rates.

18.11 PRACTICAL MACHINE LEARNING

We have introduced a wide range of machine learning techniques, each illustrated with simple learning tasks. In this section, we consider two aspects of practical machine learning. The first involves finding algorithms capable of learning to recognize handwritten digits and squeezing every last drop of predictive performance out of them. The second involves anything but—pointing out that obtaining, cleaning, and representing the data can be at least as important as algorithm engineering.

18.11.1 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 18.36.

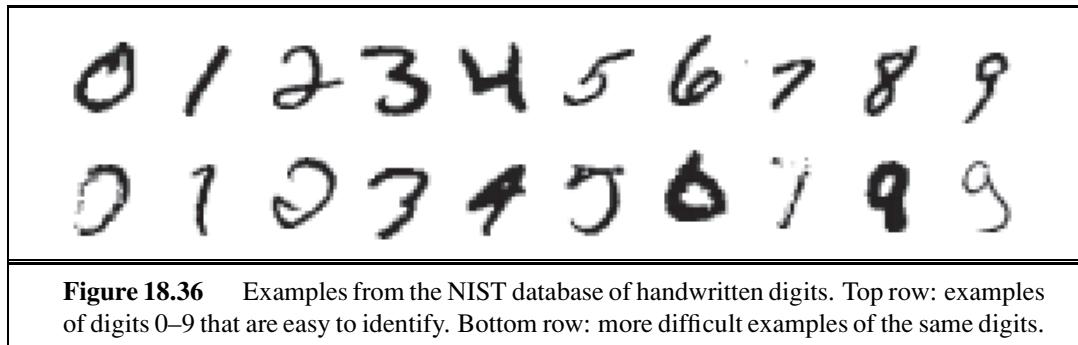


Figure 18.36 Examples from the NIST database of handwritten digits. Top row: examples of digits 0–9 that are easy to identify. Bottom row: more difficult examples of the same digits.

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 run time 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 the majority ruling. The test error rate was 0.7%.

A **support vector machine** (see Section 18.9) 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.

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 (Belongie *et al.*, 2002). 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%.

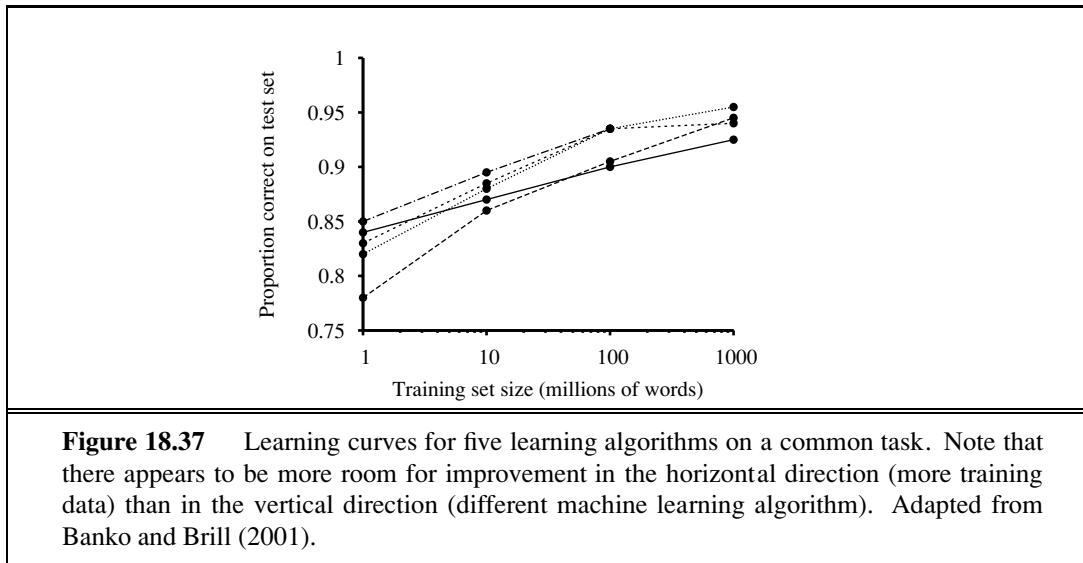
The following figure summarizes the error rates, run time performance, memory requirements, and amount of training time for the seven algorithms we have discussed. It also adds another measure, the percentage of digits that must be rejected to achieve 0.5% error. For example, if the SVM is allowed to 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
Run time (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		

18.11.2 Case study: Word senses and house prices

In a textbook we need to deal with simple, toy data to get the ideas across: a small data set, usually in two dimensions. But in practical applications of machine learning, the data set is usually large, multidimensional, and messy. The data are not handed to the analyst in a prepackaged set of (x, y) values; rather the analyst needs to go out and acquire the right data. There is a task to be accomplished, and most of the engineering problem is deciding what data are necessary to accomplish the task; a smaller part is choosing and implementing an



appropriate machine learning method to process the data. Figure 18.37 shows a typical real-world example, comparing five learning algorithms on the task of word-sense classification (given a sentence such as “The bank folded,” classify the word “bank” as “money-bank” or “river-bank”). The point is that machine learning researchers have focused mainly on the vertical direction: Can I invent a new learning algorithm that performs better than previously published algorithms on a standard training set of 1 million words? But the graph shows there is more room for improvement in the horizontal direction: instead of inventing a new algorithm, all I need to do is gather 10 million words of training data; even the *worst* algorithm at 10 million words is performing better than the *best* algorithm at 1 million. As we gather even more data, the curves continue to rise, dwarfing the differences between algorithms.

Consider another problem: the task of estimating the true value of houses that are for sale. In Figure 18.13 we showed a toy version of this problem, doing linear regression of house size to asking price. You probably noticed many limitations of this model. First, it is measuring the wrong thing: we want to estimate the selling price of a house, not the asking price. To solve this task we’ll need data on actual sales. But that doesn’t mean we should throw away the data about asking price—we can use it as one of the input features. Besides the size of the house, we’ll need more information: the number of rooms, bedrooms and bathrooms; whether the kitchen and bathrooms have been recently remodeled; the age of the house; we’ll also need information about the lot, and the neighborhood. But how do we define neighborhood? By zip code? What if part of one zip code is on the “wrong” side of the highway or train tracks, and the other part is desirable? What about the school district? Should the *name* of the school district be a feature, or the *average test scores*? In addition to deciding what features to include, we will have to deal with missing data; different areas have different customs on what data are reported, and individual cases will always be missing some data. If the data you want are not available, perhaps you can set up a social networking site to encourage people to share and correct data. In the end, this process of

deciding what features to use, and how to use them, is just as important as choosing between linear regression, decision trees, or some other form of learning.

That said, one *does* have to pick a method (or methods) for a problem. There is no guaranteed way to pick the best method, but there are some rough guidelines. Decision trees are good when there are a lot of discrete features and you believe that many of them may be irrelevant. Nonparametric methods are good when you have a lot of data and no prior knowledge, and when you don't want to worry too much about choosing just the right features (as long as there are fewer than 20 or so). However, nonparametric methods usually give you a function h that is more expensive to run. Support vector machines are often considered the best method to try first, provided the data set is not too large.

18.12 SUMMARY

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

- Learning takes many forms, depending on the nature of the agent, the component to be improved, and the available feedback.
- If the available feedback provides the correct answer for example inputs, then the learning problem is called **supervised learning**. The task is to learn a function $y = h(x)$. Learning a discrete-valued function is called **classification**; learning a continuous function is called **regression**.
- Inductive learning involves finding a hypothesis that agrees well 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.
- When there are multiple models to choose from, **cross-validation** can be used to select a model that will generalize well.
- Sometimes not all errors are equal. A **loss function** tells us how bad each error is; the goal is then to minimize loss over a validation set.
- **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.
- **Linear regression** is a widely used model. The optimal parameters of a linear regression model can be found by gradient descent search, or computed exactly.
- A linear classifier with a hard threshold—also known as a **perceptron**—can be trained by a simple weight update rule to fit data that are **linearly separable**. In other cases, the rule fails to converge.

- **Logistic regression** replaces the perceptron’s hard threshold with a soft threshold defined by a logistic function. Gradient descent works well even for noisy data that are not linearly separable.
- **Neural networks** represent complex nonlinear functions with a network of linear-threshold units. termMultilayer 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.
- **Nonparametric models** use all the data to make each prediction, rather than trying to summarize the data first with a few parameters. Examples include **nearest neighbors** and **locally weighted regression**.
- **Support vector machines** find linear separators with **maximum margin** to improve the generalization performance of the classifier. **Kernel methods** implicitly transform the input data into a high-dimensional space where a linear separator may exist, even if the original data are non-separable.
- Ensemble methods such as **boosting** often perform better than individual methods. In **online learning** we can aggregate the opinions of experts to come arbitrarily close to the best expert’s performance, even when the distribution of the data is constantly shifting.

BIBLIOGRAPHICAL AND HISTORICAL NOTES

Chapter 1 outlined the history of philosophical investigations into inductive learning. William of Ockham¹⁶ (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 (although he did say “Pluralitas non est ponenda sine necessitate,” or “plurality shouldn’t be posited without necessity”). A similar sentiment was expressed by Aristotle in 350 B.C. in *Physics* book I, chapter VI: “For the more limited, if adequate, is always preferable.”

The first notable use of decision trees was in EPAM, the “Elementary Perceiver And Memorizer” (Feigenbaum, 1961), which was a simulation of human concept learning. ID3 (Quinlan, 1979) added the crucial idea of choosing the attribute with maximum entropy; it is the basis for the decision tree algorithm in this chapter. Information theory 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.

¹⁶ The name is often misspelled as “Occam,” perhaps from the French rendering, “Guillaume d’Occam.”

Cross-validation was first introduced by Larson (1931), and in a form close to what we show by Stone (1974) and Golub *et al.* (1979). The regularization procedure is due to Tikhonov (1963). Guyon and Elisseeff (2003) introduce a journal issue devoted to the problem of feature selection. Banko and Brill (2001) and Halevy *et al.* (2009) discuss the advantages of using large amounts of data. It was Robert Mercer, a speech researcher who said in 1985 “There is no data like more data.” (Lyman and Varian, 2003) estimate that about 5 exabytes (5×10^{18} bytes) of data was produced in 2002, and that the rate of production is doubling every 3 years.

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, 2009) 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, 2007) 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.

The theory of PAC-learning was inaugurated by Leslie Valiant (1984). His 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).

Linear regression with squared error loss goes back to Legendre (1805) and Gauss (1809), who were both working on predicting orbits around the sun. The modern use of multivariate regression for machine learning is covered in texts such as Bishop (2007). Ng (2004) analyzed the differences between L_1 and L_2 regularization.

KOLMOGOROV
COMPLEXITY

MINIMUM
DESCRIPTION
LENGTH

UNIFORM
CONVERGENCE
THEORY
VC DIMENSION

The term **logistic function** comes from Pierre-François Verhulst (1804–1849), a statistician who used the curve to model population growth with limited resources, a more realistic model than the unconstrained geometric growth proposed by Thomas Malthus. Verhulst called it the *courbe logistique*, because of its relation to the logarithmic curve. The term **regression** is due to Francis Galton, nineteenth century statistician, cousin of Charles Darwin, and initiator of the fields of meteorology, fingerprint analysis, and statistical correlation, who used it in the sense of regression to the mean. The term **curse of dimensionality** comes from Richard Bellman (1961).

Logistic regression can be solved with gradient descent, or with the Newton-Raphson method (Newton, 1671; Raphson, 1690). A variant of the Newton method called L-BFGS is sometimes used for large-dimensional problems; the L stands for “limited memory,” meaning that it avoids creating the full matrices all at once, and instead creates parts of them on the fly. BFGS are authors’ initials (Byrd *et al.*, 1995).

Nearest-neighbors 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. Gionis *et al.* (1999) introduced locality-sensitive hashing, which has revolutionized the retrieval of similar objects in high-dimensional spaces, particularly in computer vision. Andoni and Indyk (2006) provide a recent survey of LSH and related methods.

The ideas behind kernel machines come from Aizerman *et al.* (1964) (who also introduced the kernel trick), but the full development of the theory is due to Vapnik and his colleagues (Boser *et al.*, 1992). SVMs were made practical with the introduction of the soft-margin classifier for handling noisy data in a paper that won the 2008 ACM Theory and Practice Award (Cortes and Vapnik, 1995), and of the Sequential Minimal Optimization (SMO) algorithm for efficiently solving SVM problems using quadratic programming (Platt, 1999). SVMs have proven to be very popular and effective for tasks such as text categorization (Joachims, 2001), computational genomics (Cristianini and Hahn, 2007), and natural language processing, such as the handwritten digit recognition of DeCoste and Schölkopf (2002). As part of this process, many new kernels have been designed that work with strings, trees, and other nonnumerical data types. A related technique that also uses the kernel trick to implicitly represent an exponential feature space is the voted perceptron (Freund and Schapire, 1999; Collins and Duffy, 2002). Textbooks on SVMs include Cristianini and Shawe-Taylor (2000) and Schölkopf and Smola (2002). A friendlier exposition appears in the *AI Magazine* article by Cristianini and Schölkopf (2002). Bengio and LeCun (2007) show some of the limitations of SVMs and other local, nonparametric methods for learning functions that have a global structure but do not have local smoothness.

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 this chapter originated with theoretical work by Schapire (1990). The ADABOOST algorithm was developed by Freund and Schapire

(1996) and analyzed theoretically by Schapire (2003). Friedman *et al.* (2000) explain boosting from a statistician’s viewpoint. Online learning is covered in a survey by Blum (1996) and a book by Cesa-Bianchi and Lugosi (2006). Dredze *et al.* (2008) introduce the idea of confidence-weighted online learning for classification: in addition to keeping a weight for each parameter, they also maintain a measure of confidence, so that a new example can have a large effect on features that were rarely seen before (and thus had low confidence) and a small effect on common features that have already been well-estimated.

The literature on neural networks is rather too large (approximately 150,000 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). (As mentioned in Chapter 1, John McCarthy has pointed to the work of Nicolas Rashevsky (1936, 1938) as the earliest mathematical model of neural learning.) 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). 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 a 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).

The probabilistic interpretation of neural networks has several sources, including Baum and Wilczek (1988) and Bridle (1990). The role of the sigmoid function is discussed by Jordan (1995). Bayesian parameter learning for neural networks was proposed by MacKay

(1992) and is explored further by Neal (1996). The capacity of neural networks to represent functions was investigated by Cybenko (1988, 1989), who showed that two hidden layers are enough to represent any function and a single layer is enough to represent any *continuous* function. The “optimal brain damage” method for removing useless connections is by LeCun *et al.* (1989), and Sietsma and Dow (1988) show how to remove useless units. The tiling algorithm for growing larger structures is due to Mézard and Nadal (1989). LeCun *et al.* (1995) survey a number of algorithms for handwritten digit recognition. Improved error rates since then were reported by Belongie *et al.* (2002) for shape matching and DeCoste and Schölkopf (2002) for virtual support vectors. At the time of writing, the best test error rate reported is 0.39% by Ranzato *et al.* (2007) using a convolutional neural network.

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.¹⁷ 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, a result that should not be surprising in the light of our discussion of regularization.

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 20.3.1. 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 labeled training data, and the second phase, although supervised, is efficient. See Bishop (1995) for more details.

RADIAL BASIS
FUNCTION

HOPFIELD NETWORK

ASSOCIATIVE
MEMORY

Recurrent networks, in which units are linked in cycles, were mentioned in the chapter but not explored in depth. **Hopfield networks** (Hopfield, 1982) are probably the best-understood class of recurrent networks. They use *bidirectional* connections with *symmetric* weights (i.e., $w_{i,j} = w_{j,i}$), all of the units are both input and output units, the activation function g is the sign function, and the activation levels can only be ± 1 . A Hopfield network functions as an **associative memory**: after the network trains on a set of examples, a new stimulus will cause it to settle into an activation pattern corresponding to the example in the training set that *most closely resembles* the new stimulus. For example, if the training set consists of a set of photographs, and the new stimulus is a small piece of one of the photographs, then the network activation levels will reproduce the photograph from which the piece was taken. Notice that the original photographs are not stored separately in the network; each

¹⁷ 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.

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 (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.)

For neural nets, Bishop (1995), Ripley (1996), and Haykin (2008) are the leading texts. The field of computational neuroscience is covered by Dayan and Abbott (2001).

The approach taken in this chapter was influenced by the excellent course notes of David Cohn, Tom Mitchell, Andrew Moore, and Andrew Ng. There are several top-notch textbooks in Machine Learning (Mitchell, 1997; Bishop, 2007) and in the closely allied and overlapping fields of pattern recognition (Ripley, 1996; Duda *et al.*, 2001), statistics (Wasserman, 2004; Hastie *et al.*, 2001), data mining (Hand *et al.*, 2001; Witten and Frank, 2005), computational learning theory (Kearns and Vazirani, 1994; Vapnik, 1998) and information theory (Shannon and Weaver, 1949; MacKay, 2002; Cover and Thomas, 2006). Other books concentrate on implementations (Segaran, 2007; Marsland, 2009) and comparisons of algorithms (Michie *et al.*, 1994). Current research in machine learning is published in the annual proceedings of the International Conference on Machine Learning (ICML) and the conference on Neural Information Processing Systems (NIPS), in *Machine Learning* and the *Journal of Machine Learning Research*, and in mainstream AI journals.

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. Describe the percepts and actions of the infant, and the types of learning the infant must do. Describe the subfunctions the infant is trying to learn in terms of inputs and outputs, and available example data.

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 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?

18.4 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.

CLASS PROBABILITY

- 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.

18.5 Suppose that an attribute splits the set of examples E into subsets E_k and that each subset has p_k positive examples and n_k negative examples. Show that the attribute has strictly positive information gain unless the ratio $p_k/(p_k + n_k)$ is the same for all k .

18.6 Consider the following data set comprised of three binary input attributes (A_1 , A_2 , and A_3) and one binary output:

Example	A_1	A_2	A_3	Output y
\mathbf{x}_1	1	0	0	0
\mathbf{x}_2	1	0	1	0
\mathbf{x}_3	0	1	0	0
\mathbf{x}_4	1	1	1	1
\mathbf{x}_5	1	1	0	1

Use the algorithm in Figure 18.5 (page 702) to learn a decision tree for these data. Show the computations made to determine the attribute to split at each node.

18.7 A decision *graph* is a generalization of a decision tree that allows nodes (i.e., attributes used for splits) to have multiple parents, rather than just a single parent. The resulting graph must still be acyclic. Now, consider the XOR function of *three* binary input attributes, which produces the value 1 if and only if an odd number of the three input attributes has value 1.

- a. Draw a minimal-sized decision *tree* for the three-input XOR function.
- b. Draw a minimal-sized decision *graph* for the three-input XOR function.

18.8 This exercise considers χ^2 pruning of decision trees (Section 18.3.5).

- a. Create a data set with two input attributes, such that the information gain at the root of the tree for both attributes is zero, but there is a decision tree of depth 2 that is consistent with all the data. What would χ^2 pruning do on this data set if applied bottom up? If applied top down?
- b. Modify DECISION-TREE-LEARNING to include χ^2 -pruning. You might wish to consult Quinlan (1986) or Kearns and Mansour (1998) for details.



18.9 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 \mathbf{x} has a missing value for attribute A and that the decision tree tests for A at a node that \mathbf{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 .

18.10 In Section 18.3.6, 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.

18.11 Suppose you are running a learning experiment on a new algorithm for Boolean classification. You have a data set consisting of 100 positive and 100 negative examples. You plan to use leave-one-out cross-validation and compare your algorithm to a baseline function, 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 every time. Can you explain why?

18.12 Construct a *decision list* to classify the data below. Select tests to be as small as possible (in terms of attributes), breaking ties among tests with the same number of attributes by selecting the one that classifies the greatest number of examples correctly. If multiple tests have the same number of attributes and classify the same number of examples, then break the tie using attributes with lower index numbers (e.g., select A_1 over A_2).

Example	A_1	A_2	A_3	A_4	y
\mathbf{x}_1	1	0	0	0	1
\mathbf{x}_2	1	0	1	1	1
\mathbf{x}_3	0	1	0	0	1
\mathbf{x}_4	0	1	1	0	0
\mathbf{x}_5	1	1	0	1	1
\mathbf{x}_6	0	1	0	1	0
\mathbf{x}_7	0	0	1	1	1
\mathbf{x}_8	0	0	1	0	0

18.13 Prove that a decision list can represent the same function as a decision tree while using at most as many rules as there are leaves in the decision tree for that function. Give an example of a function represented by a decision list using strictly fewer rules than the number of leaves in a minimal-sized decision tree for that same function.

18.14 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 .

18.15 Suppose a 7-nearest-neighbors regression search returns $\{7, 6, 8, 4, 7, 11, 100\}$ as the 7 nearest y values for a given x value. What is the value of \hat{y} that minimizes the L_1 loss function on this data? There is a common name in statistics for this value as a function of the y values; what is it? Answer the same two questions for the L_2 loss function.

18.16 Figure 18.31 showed how a circle at the origin can be linearly separated by mapping from the features (x_1, x_2) to the two dimensions (x_1^2, x_2^2) . But what if the circle is not located at the origin? What if it is an ellipse, not a circle? The general equation for a circle (and hence the decision boundary) is $(x_1 - a)^2 + (x_2 - b)^2 - r^2 = 0$, and the general equation for an ellipse is $c(x_1 - a)^2 + d(x_2 - b)^2 - 1 = 0$.

- a. Expand out the equation for the circle and show what the weights w_i would be for the decision boundary in the four-dimensional feature space (x_1, x_2, x_1^2, x_2^2) . Explain why this means that any circle is linearly separable in this space.
- b. Do the same for ellipses in the five-dimensional feature space $(x_1, x_2, x_1^2, x_2^2, x_1 x_2)$.

18.17 Construct a support vector machine that computes the XOR function. Use values of $+1$ and -1 (instead of 1 and 0) for both inputs and outputs, so that an example looks like $([-1, 1], 1)$ or $([-1, -1], -1)$. Map the input $[x_1, x_2]$ into a space consisting of 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.

18.18 Consider an ensemble learning algorithm that uses simple majority voting among K 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 K and ϵ , and evaluate it for the cases where $K = 5, 10$, 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 ?

18.19 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.

18.20 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?

18.21 Section 18.6.4 (page 725) noted that the output of the logistic function could be interpreted as a *probability* p assigned by the model to the proposition that $f(\mathbf{x}) = 1$; the probability that $f(\mathbf{x}) = 0$ is therefore $1 - p$. Write down the probability p as a function of \mathbf{x} and calculate the derivative of $\log p$ with respect to each weight w_i . Repeat the process for $\log(1 - p)$. These calculations give a learning rule for minimizing the negative-log-likelihood

loss function for a probabilistic hypothesis. Comment on any resemblance to other learning rules in the chapter.

18.22 Suppose you had a neural network with linear activation functions. That is, for each unit the output is some constant c times the weighted sum of the inputs.

- a. Assume that the network has one hidden layer. For a given assignment to the weights \mathbf{w} , write down equations for the value of the units in the output layer as a function of \mathbf{w} and the input layer \mathbf{x} , without any explicit mention of 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), but this time do it for a network with any number of hidden layers.
- c. Suppose a network with one hidden layer and linear activation functions has n input and output nodes and h hidden nodes. What effect does the transformation in part (a) to a network with no hidden layers have on the total number of weights? Discuss in particular the case $h \ll n$.

18.23 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 it to zero.)

18.24 The neural network whose learning performance is measured in Figure 18.25 has four hidden nodes. This number was chosen somewhat arbitrarily. Use a cross-validation method to find the best number of hidden nodes.

18.25 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 labeled 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 $N - 1$.

19 KNOWLEDGE IN LEARNING

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

PRIOR KNOWLEDGE

In all of the approaches to learning described in the previous chapter, 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 perhaps a preference for simpler hypotheses. 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.

19.1.1 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 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 (page 700) 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 example is given by a literal using the goal predicate, in this case

$$\text{WillWait}(X_1) \quad \text{or} \quad \neg\text{WillWait}(X_1).$$

The complete training set can thus be expressed as the conjunction of all the example descriptions and goal literals.

The aim of inductive learning in general is to find a hypothesis that classifies the examples well and generalizes well to new examples. Here we are concerned with hypotheses expressed in logic; each hypothesis h_j will have the form

$$\forall x \text{ } \text{Goal}(x) \Leftrightarrow C_j(x),$$

where $C_j(x)$ is a candidate definition—some expression involving the attribute predicates. For example, a decision tree can be interpreted as a logical expression of this form. Thus, the tree in Figure 18.6 (page 702) expresses the following logical definition (which we will call h_r for future reference):

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

EXTENSION

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 \mathcal{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_j is consistent with the entire training set, it has to be consistent with each example in the training set. What would it mean for it to be inconsistent with an example? There are two possible ways that this can happen:

FALSE NEGATIVE

- An example can be a **false negative** for the hypothesis, if the hypothesis says it should be negative but in fact it is positive. For instance, the new example X_{13} described by $\text{Patrons}(X_{13}, \text{Full}) \wedge \neg\text{Hungry}(X_{13}) \wedge \dots \wedge \text{WillWait}(X_{13})$ would be a false negative for the hypothesis h_r given earlier. From h_r and the example description, we can deduce both $\text{WillWait}(X_{13})$, 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.

19.1.2 Current-best-hypothesis search

CURRENT-BEST-HYPOTHESIS

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

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

¹ The terms “false positive” and “false negative” are used in medicine to describe erroneous results from lab tests. A result is a false positive if it indicates that the patient has the disease when in fact no disease is present.

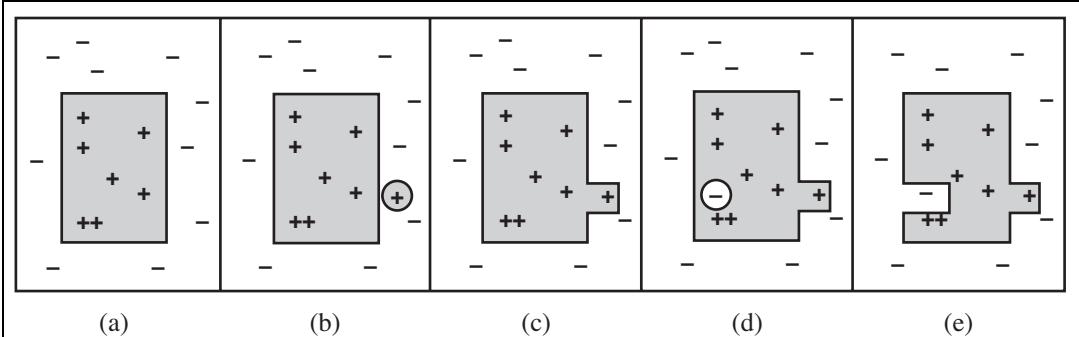


Figure 19.1 (a) A consistent hypothesis. (b) A false negative. (c) The hypothesis is generalized. (d) A false positive. (e) The hypothesis is specialized.

```

function CURRENT-BEST-LEARNING(examples, h) returns a hypothesis or fail
  if examples is empty then
    return h
  e  $\leftarrow$  FIRST(examples)
  if e is consistent with h then
    return CURRENT-BEST-LEARNING(REST(examples), h)
  else if e is a false positive for h then
    for each h' in specializations of h consistent with examples seen so far do
      h''  $\leftarrow$  CURRENT-BEST-LEARNING(REST(examples), h')
      if h''  $\neq$  fail then return h''
    else if e is a false negative for h then
      for each h' in generalizations of h consistent with examples seen so far do
        h''  $\leftarrow$  CURRENT-BEST-LEARNING(REST(examples), h')
        if h''  $\neq$  fail then return h''
    return fail
  
```

Figure 19.2 The current-best-hypothesis learning algorithm. It searches for a consistent hypothesis that fits all the examples and backtracks when no consistent specialization/generalization can be found. To start the algorithm, any hypothesis can be passed in; it will be specialized or generalized as needed.

SPECIALIZATION

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. 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.

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

$$\forall x \ C_2(x) \Rightarrow C_1(x).$$

DROPPING CONDITIONS

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 $\text{Alternate}(x) \wedge \text{Patrons}(x, \text{Some})$, 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. The attribute $\text{Alternate}(X_1)$ 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 , while continuing to classify X_1 as positive. 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:

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

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

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

```
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
     $V \leftarrow \{h \in V : h \text{ is consistent with } e\}$ 
```

Figure 19.3 The version space learning algorithm. It finds a subset of V that is consistent with all the *examples*.

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 examples and a large space, however, some difficulties arise:

1. Checking all the previous examples 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.

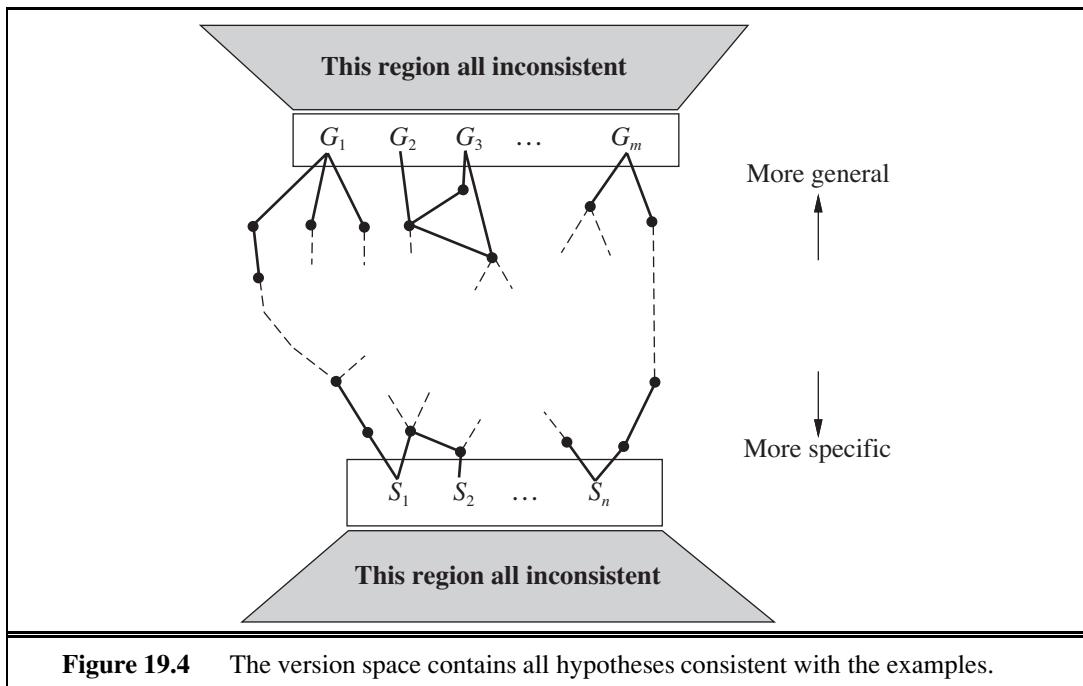
19.1.3 Least-commitment search

Backtracking arises because the current-best-hypothesis approach has to *choose* a particular hypothesis as its best guess even though it does not have enough data yet to be sure of the choice. What we can do instead is to keep around all and only those hypotheses that are consistent with all the data so far. Each new example 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).

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 already. 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 are 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 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:

BOUNDARY SET

G-SET

S-SET

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” between 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.

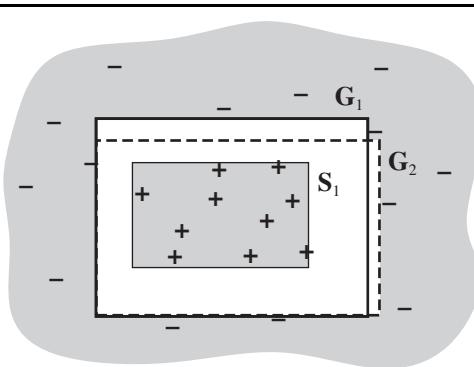


Figure 19.5 The extensions of the members of G and S . No known examples lie in between the two sets of boundaries.

We need to worry about the members S_i and G_i of the S - and G -sets. For each one, the new example 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 example until one of three things happens:

1. We have exactly one hypothesis 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 and have 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.

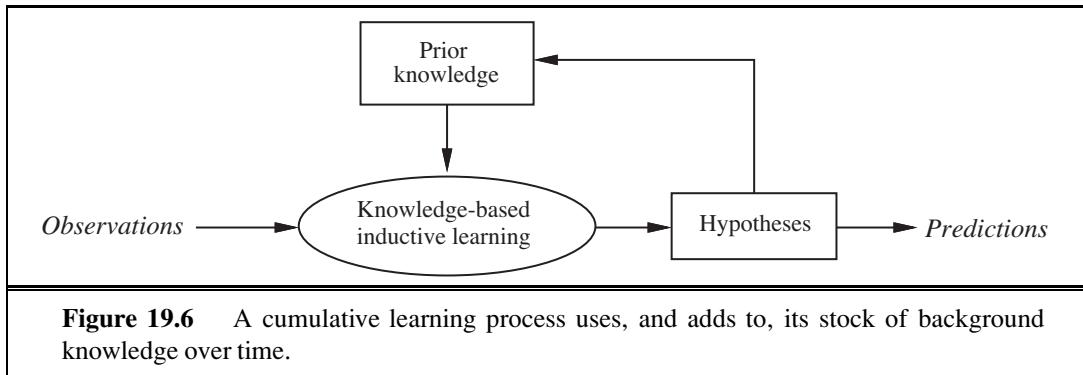
There are two principal drawbacks to the version-space approach:

- 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 or G-set may grow exponentially in the number of attributes, even though efficient learning algorithms exist for those hypothesis spaces.

To date, no completely successful solution has been found for the problem of noise. The problem of disjunction can be addressed by allowing only 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.

GENERALIZATION
HIERARCHY

The pure version space algorithm was first applied in the Meta-DENDRAL system, which was designed to learn rules for predicting how molecules would break into pieces in a mass spectrometer (Buchanan and Mitchell, 1978). Meta-DENDRAL was able to generate rules that were sufficiently novel to warrant publication in a journal of analytical chemistry—the first real scientific knowledge generated by a computer program. It was also used in the elegant LEX system (Mitchell *et al.*, 1983), which was able to learn to solve symbolic integration problems by studying its own successes and failures. Although version space methods are probably not practical in most real-world learning problems, mainly because of noise, they provide a good deal of insight into the logical structure of hypothesis space.



19.2 KNOWLEDGE IN LEARNING

The preceding section described the simplest setting for inductive learning. To understand the role of prior knowledge, we need to talk about the logical relationships among hypotheses, example descriptions, and classifications. Let *Descriptions* denote the conjunction of all the example descriptions in the training set, and let *Classifications* denote the conjunction of all the example classifications. Then a *Hypothesis* that “explains the observations” must satisfy the following property (recall that \models means “logically entails”):

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

ENTAILMENT
CONSTRAINT

We call this kind of relationship an **entailment constraint**, in which *Hypothesis* is the “unknown.” Pure inductive learning means solving this constraint, where *Hypothesis* is drawn from some predefined hypothesis space. For example, if we consider a decision tree as a logical formula (see Equation (19.1) on page 769), then a decision tree that is consistent with all the examples will satisfy Equation (19.3). If we place *no* restrictions on the logical form of the hypothesis, of course, then *Hypothesis* = *Classifications* also satisfies the constraint. Ockham’s razor tells us to prefer *small*, consistent hypotheses, so we try to do better than simply memorizing the examples.



This simple knowledge-free picture of inductive learning persisted until the early 1980s. The modern approach is to design agents that *already know something* and are trying to learn some more. This may not sound like a terrifically deep insight, but it makes quite a difference to the way we design agents. It might also have some relevance to our theories about how science itself works. The general idea is shown schematically in Figure 19.6.

An autonomous learning agent that uses background knowledge must somehow obtain 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.

19.2.1 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 traveler 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*.

19.2.2 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 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 example*. The agent *could have* derived the example from what it already knew, although that might have required an unreasonable amount of computation. EBL is now viewed as a method for converting first-principles theories into useful, special-purpose knowledge. We describe algorithms for EBL in Section 19.3.

The situation of our traveler in Brazil is quite different, for she cannot necessarily explain why Fernando speaks the way he does, unless she knows her papal bulls. Moreover, the same generalization would be forthcoming from a traveler entirely ignorant of colonial history. The relevant prior knowledge in this case is that, within any given country, most people tend to speak the same language; on the other hand, Fernando is not assumed to be the name of all Brazilians because this kind of regularity does not hold for names. Similarly, the freshman physics student also would be hard put to explain the particular values that she discovers for the conductance and density of copper. She does know, however, that the material of which an object is composed and its temperature together determine its conductance. 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:

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

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

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

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

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 9.17). If we differentiate an expression such as X^2 with respect to X , we obtain $2X$. (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 $\text{ASK}(\text{Derivative}(X^2, X) = d, KB)$, with solution $d = 2X$.

Anyone who knows differential calculus can see this solution “by inspection” as a result of practice in solving such problems. A student encountering such problems for the first time, or a program with no experience, will have a much more difficult job. Application of the standard rules of differentiation eventually yields the expression $1 \times (2 \times (X^{(2-1)}))$, and eventually this simplifies to $2X$. In the authors’ logic programming implementation, this takes 136 proof steps, of which 99 are on dead-end branches in the proof. After such an experience, we would like the program to solve the same problem much more quickly the next time it arises.

MEMOIZATION

The technique of **memoization** has long been used in computer science to speed up programs by saving the results of computation. The basic idea of memo functions is to accumulate a database of input–output pairs; when the function is called, it first checks the database to see whether it can avoid solving the problem from scratch. Explanation-based learning takes this a good deal further, by creating *general* rules that cover an entire class of cases. In the case of differentiation, memoization would remember that the derivative of X^2 with respect to X is $2X$, but would leave the agent to calculate the derivative of Z^2 with respect to Z from scratch. We would like to be able to extract the general rule that for any arithmetic unknown u , the derivative of u^2 with respect to u is $2u$. (An even more general rule for u^n can also be produced, but the current example suffices to make the point.) In logical terms, this is expressed by the rule

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

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*, 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.



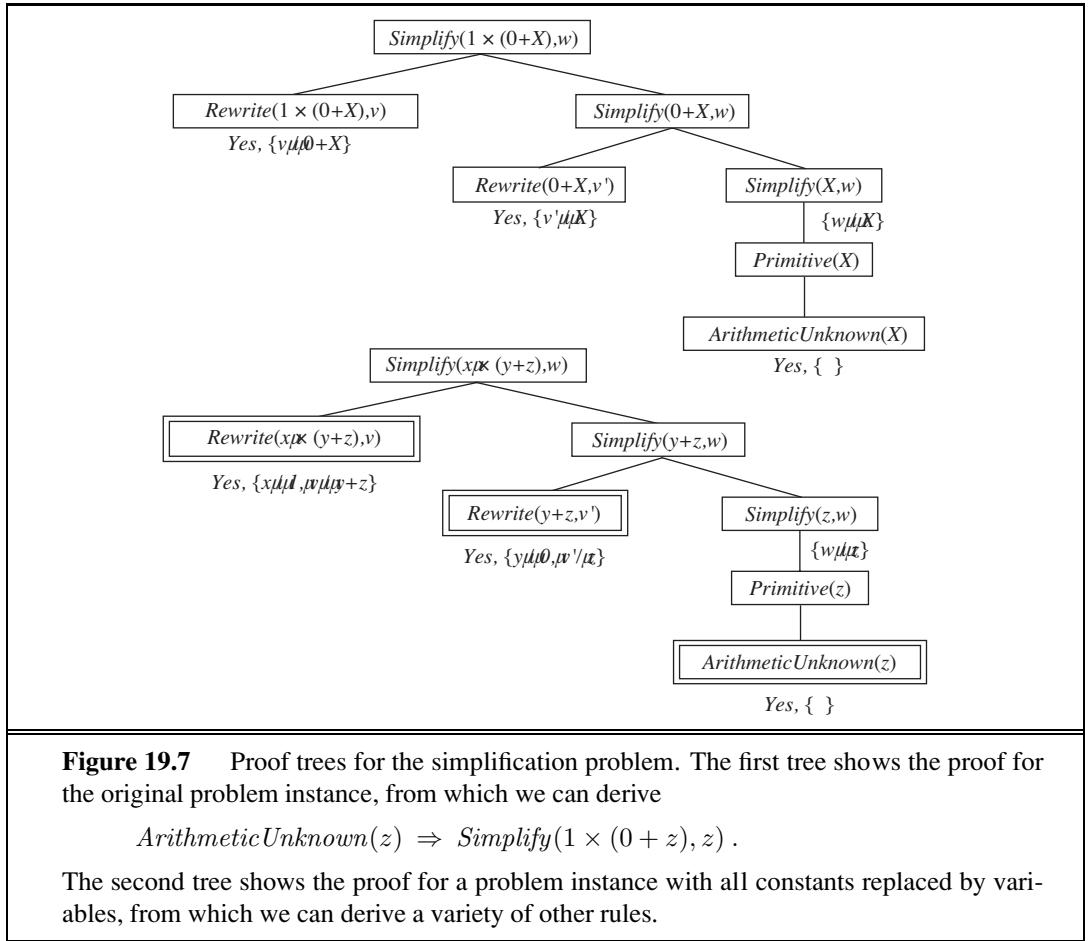
19.3.1 Extracting general rules from examples

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:

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

The proof that the answer is X is shown in the top half of Figure 19.7. The EBL method actually constructs two proof trees simultaneously. The second proof tree uses a *variabilized* goal in which the constants from the original goal are replaced by variables. As the original proof proceeds, the variabilized proof proceeds in step, using *exactly the same rule applications*. This could cause some of the variables to become instantiated. For example, in order to use the rule $\text{Rewrite}(1 \times u, u)$, 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:

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

Notice that the first two conditions on the left-hand side are true *regardless of the value of z*. We can therefore drop them from the rule, yielding

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

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 from the left-hand side that are true regardless of the values of the variables in the goal.

19.3.2 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

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

This rule is as valid as, but *more general* than, the rule using *ArithmeticUnknown*, because it covers cases where z is a number. We can extract a still more general rule by pruning after the step *Simplify*($y + z, w$), yielding the rule

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

In general, a rule can be extracted from *any partial subtree* of the generalized proof tree. Now we have a problem: which of these rules do we choose?

The choice of which rule to generate comes down to the question of efficiency. There are three factors involved in the analysis of efficiency gains from EBL:

1. Adding large numbers of rules can slow down the reasoning process, because the inference mechanism must still check those rules even in cases where they do not yield a solution. In other words, it increases the **branching factor** in the search space.
2. To compensate for the slowdown in reasoning, the derived rules must offer significant increases in speed for the cases that they do cover. These increases come about mainly because the derived rules avoid dead ends that would otherwise be taken, but also because they shorten the proof itself.
3. Derived rules should be as general as possible, so that they apply to the largest possible set of cases.

OPERATIONALITY

A common approach to ensuring that derived rules are efficient is to insist on the **operationality** of each subgoal in the rule. A subgoal is operational if it is “easy” to solve. For example, the subgoal *Primitive*(z) is easy to solve, requiring at most two steps, whereas the subgoal *Simplify*($y + z, w$) could lead to an arbitrary amount of inference, depending on the values of y and z . If a test for operationality is carried out at each step in the construction of the generalized proof, then we can prune the rest of a branch as soon as an operational subgoal is found, keeping just the operational subgoal as a conjunct of the new rule.

Unfortunately, there is usually a tradeoff between operationality and generality. More specific subgoals are generally easier to solve but cover fewer cases. Also, operationality is a matter of degree: one or two steps is definitely operational, but what about 10 or 100?

Finally, the cost of solving a given subgoal depends on what other rules are available in the knowledge base. It can go up or down as more rules are added. Thus, EBL systems really face a very complex optimization problem in trying to maximize the efficiency of a given initial knowledge base. It is sometimes possible to derive a mathematical model of the effect on overall efficiency of adding a given rule and to use this model to select the best rule to add. The analysis can become very complicated, however, especially when recursive rules are involved. One promising approach is to address the problem of efficiency empirically, simply by adding several rules and seeing which ones are useful and actually speed things up.



Empirical analysis of efficiency is actually at the heart of EBL. What we have been calling loosely the “efficiency of a given knowledge base” is actually the average-case complexity on a distribution of problems. *By generalizing from past example problems, EBL makes the knowledge base more efficient for the kind of problems that it is reasonable to expect.* This works as long as the distribution of past examples is roughly the same as for future examples—the same assumption used for PAC-learning in Section 18.5. If the EBL system is carefully engineered, it is possible to obtain significant speedups. For example, a very large Prolog-based natural language system designed for speech-to-speech translation between Swedish and English was able to achieve real-time performance only by the application of EBL to the parsing process (Samuelsson and Rayner, 1991).

19.4 LEARNING USING RELEVANCE INFORMATION

Our traveler in Brazil seems to be able to make a confident generalization concerning the language spoken by other Brazilians. The inference is sanctioned by her background knowledge, namely, that people in a given country (usually) speak the same language. We can express this in first-order logic as follows:²

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

(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):

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

² We assume for the sake of simplicity that a person speaks only one language. Clearly, the rule would have to be amended for countries such as Switzerland and India.

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:

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

The corresponding generalizations follow logically from the determinations and observations.

19.4.1 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, $\log(|\mathcal{H}|)$ examples are required to converge to a reasonable hypothesis, where $|\mathcal{H}|$ 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, $|\mathcal{H}| = O(2^{2^n})$, 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 $O(2^{n-d})$.

19.4.2 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 goal predicate. For example, suppose we have the following examples of conductance measurements on material samples:

```

function MINIMAL-CONSISTENT-DET( $E, A$ ) returns a set of attributes
  inputs:  $E$ , a set of examples
     $A$ , a set of attributes, of size  $n$ 

  for  $i = 0$  to  $n$  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

```

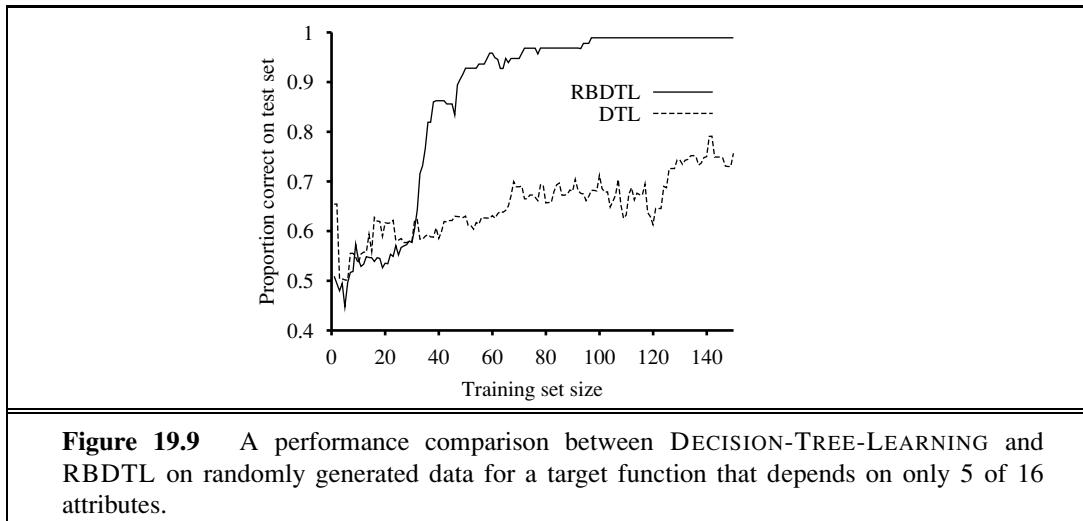
Figure 19.8 An algorithm for finding a minimal consistent determination.

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

The minimal consistent determination is $Material \wedge Temperature \succ Conductance$. There is a nonminimal but consistent determination, namely, $Mass \wedge Size \wedge Temperature \succ Conductance$. This is consistent with the examples because mass and size determine density and, in our data set, we do not have two different materials with the same density. As usual, we would need a larger sample set in order to eliminate a nearly correct hypothesis.

There are several possible algorithms for finding minimal consistent determinations. The most obvious approach is to conduct a search through the space of determinations, checking all determinations with one predicate, two predicates, and so on, until a consistent determination is found. We will assume a simple attribute-based representation, like that used for decision tree learning in Chapter 18. A determination d will be represented by the set of attributes on the left-hand side, because the target predicate is assumed to be 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 hypotheses as logic programs.³ It has gained popularity for three reasons. First, ILP offers a rigorous approach to 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.

19.5.1 An example

Recall from Equation (19.5) that the general knowledge-based induction problem is to “solve” the entailment constraint

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

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 8.14, 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

³ It might be appropriate at this point for the reader to refer to Chapter 7 for some of the underlying concepts, including Horn clauses, conjunctive normal form, unification, and resolution.

complete set of *Classifications* contains $20 \times 20 = 400$ conjuncts of the form

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

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

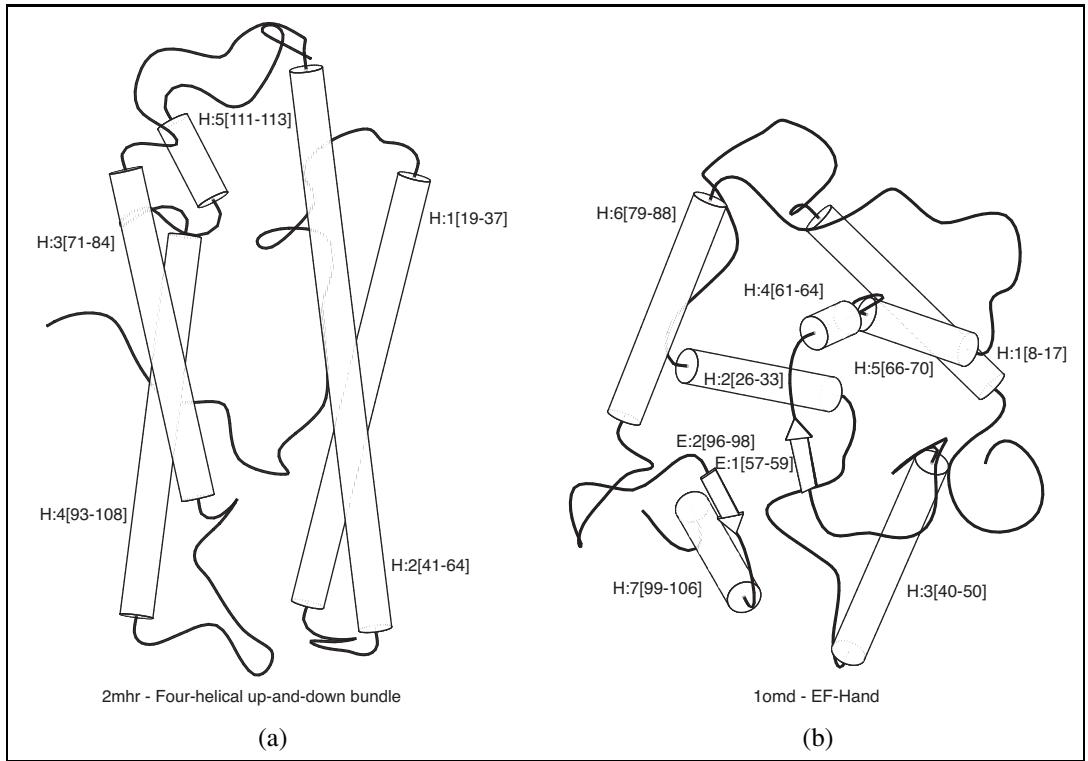


Figure 19.10 (a) and (b) show positive and negative examples, respectively, of the “four-helical up-and-down bundle” concept in the domain of protein folding. Each example structure is coded into a logical expression of about 100 conjuncts such as *TotalLength(D2mhr, 118) \wedge NumberHelices(D2mhr,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

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

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 “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.”

for *Hypothesis* is the following:

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

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

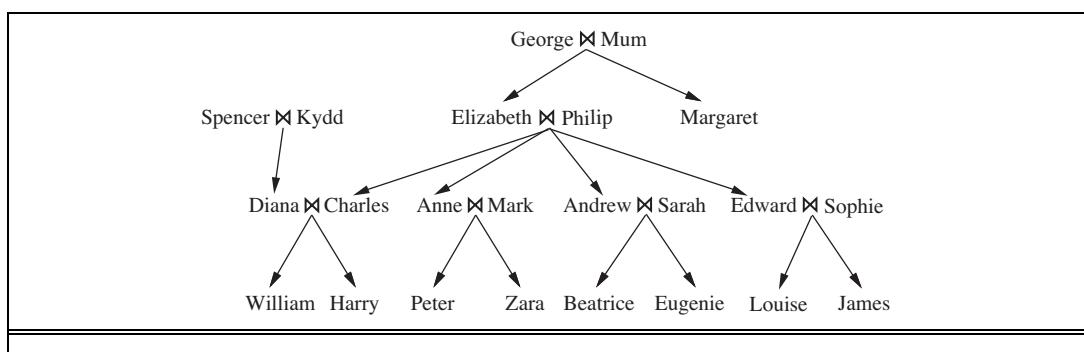


Figure 19.11 A typical family tree.

“*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. 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.

19.5.2 Top-down inductive learning methods

The first approach to ILP works by starting with a very general rule and gradually specializing it so that it fits the data. This is essentially what happens in decision-tree learning, where a decision tree is gradually grown until it is consistent with the observations. To do ILP we use first-order literals instead of attributes, and the hypothesis is a set of clauses instead of a decision tree. This section describes FOIL (Quinlan, 1990), one of the first ILP programs.

Suppose we are trying to learn a definition of the *Grandfather*(x, y) predicate, using the same family data as before. As with decision-tree learning, we can divide the examples into positive and negative examples. Positive examples are

$$\langle \text{George}, \text{Anne} \rangle, \langle \text{Philip}, \text{Peter} \rangle, \langle \text{Spencer}, \text{Harry} \rangle, \dots$$

and negative examples are

$$\langle \text{George}, \text{Elizabeth} \rangle, \langle \text{Harry}, \text{Zara} \rangle, \langle \text{Charles}, \text{Philip} \rangle, \dots$$

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 *Grandfather*(x, y) as the head. The clauses must classify the 12 positive examples as instances of the *Grandfather*(x, y) relationship, while ruling out the 388 negative examples. The clauses are Horn clauses, with the extension that negated literals are allowed in the body of a clause and are interpreted 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:

$$\begin{aligned} \text{Father}(x, y) &\Rightarrow \text{Grandfather}(x, y) . \\ \text{Parent}(x, z) &\Rightarrow \text{Grandfather}(x, y) . \\ \text{Father}(x, z) &\Rightarrow \text{Grandfather}(x, y) . \end{aligned}$$

(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, the solution is a set of Horn clauses, each of which implies the target predicate. For example, if we didn't have the *Parent* predicate in our vocabulary, then the solution might be

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

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

Note that each of these clauses covers some of the positive examples, that together they cover all the positive examples, and that NEW-CLAUSE is designed in such a way that no clause will incorrectly cover a negative example. 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 the negative examples. Then the positive examples covered by the clause are removed from the training set, and the process continues until no positive examples remain. The two main subroutines to be explained are NEW-LITERALS, which constructs all possible new literals to add to the clause, and CHOOSE-LITERAL, which selects a literal to add.

NEW-LITERALS takes a clause and constructs all possible “useful” literals that could be added to the clause. Let us use as an example the clause

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

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 $\text{Mother}(z, u)$, $\text{Married}(z, z)$, $\neg\text{Male}(y)$, and $\text{Grandfather}(v, x)$ are allowed, whereas $\text{Married}(u, v)$ 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

```

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 positive 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 a set of examples
  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-LITERALS and CHOOSE-LITERAL are explained in the text.

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 704) to decide which literal to add. The exact details are not 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.

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.

19.5.3 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* \wedge *Hypothesis* \wedge *Descriptions*, then one must be able to prove this fact by resolution (because resolution is complete). If we can “run the proof backward,” then we can find a *Hypothesis* such that the proof goes through. The key, then, is to find a way to invert the resolution process.

We will show a backward proof process for inverse resolution that consists of individual backward steps. Recall that an ordinary resolution step takes two clauses C_1 and C_2 and resolves them to produce the **resolvent** C . An inverse resolution step takes a resolvent C and produces two clauses C_1 and C_2 , such that C is the result of resolving C_1 and C_2 . Alternatively, it may take a resolvent C and clause C_1 and produce a clause C_2 such that C is the result of resolving C_1 and C_2 .

The early steps in an inverse resolution process are shown in Figure 19.13, where we focus on the positive example *Grandparent(George, Anne)*. The process begins at the end of the proof (shown at the bottom of the figure). We take the resolvent C to be empty clause (i.e. a contradiction) and C_2 to be $\neg\text{Grandparent}(\text{George}, \text{Anne})$, which is the negation of the goal example. The first inverse step takes C and C_2 and generates the clause *Grandparent(George, Anne)* for C_1 . The next step takes this clause as C and the clause *Parent(Elizabeth, Anne)* as C_2 , and generates the clause

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

as C_1 . The final step treats this clause as the resolvent. With *Parent(George, Elizabeth)* as C_2 , one possible clause C_1 is the hypothesis

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

Now we have a resolution proof that the hypothesis, descriptions, and background knowledge entail the classification *Grandparent(George, Anne)*.

Clearly, inverse resolution involves a search. Each inverse resolution step is nondeterministic, because for any C , there can be many or even an infinite number of clauses C_1 and C_2 that resolve to C . For example, instead of choosing $\neg\text{Parent}(\text{Elizabeth}, y) \vee \text{Grandparent}(\text{George}, y)$ 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. Linear resolution produces proof trees that have a linear branching structure—the whole tree follows one line, with only single clauses branching off that line (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

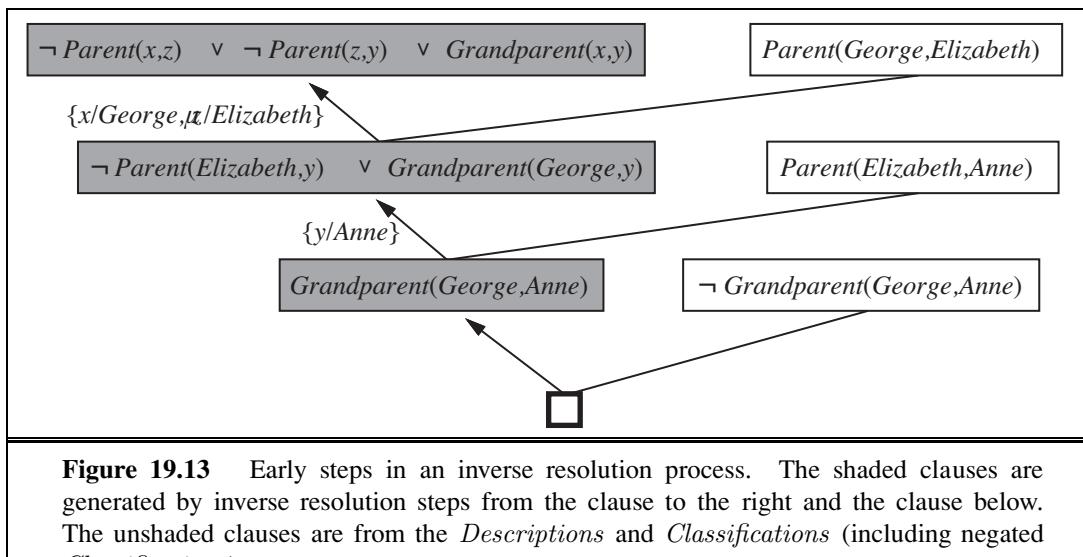
$$\text{Background} \wedge \text{Hypothesis} \wedge \text{Descriptions} \models \text{Classifications}$$

to the logically equivalent form

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

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 (Lavrauc and Dzeroski, 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 10.

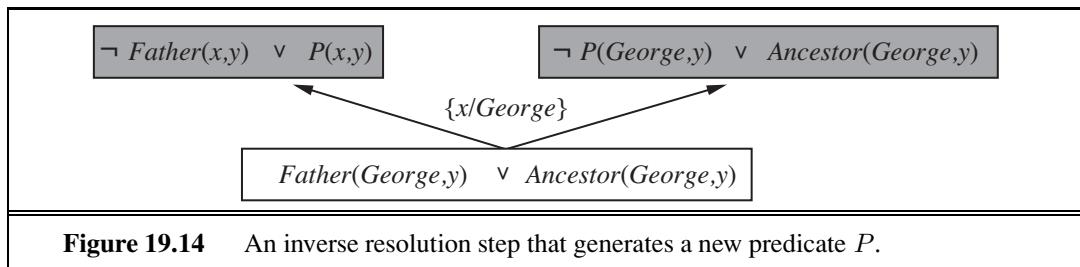
19.5.4 Making discoveries with inductive logic programming

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.

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 that allow it to outperform linear regression, neural nets, and decision trees. Most impressively, King *et al.* (2009) endowed a robot with the ability to perform molecular biology experiments and extended ILP techniques to include experiment design, thereby creating an autonomous scientist that actually discovered new knowledge about the functional genomics of yeast. For all these examples it appears that the ability both to represent relations and to use background knowledge contribute to ILP's high performance. The fact that the rules found by ILP can be interpreted by humans contributes to the acceptance of these techniques in biology journals rather than just computer science journals.

ILP has made contributions to other sciences besides biology. One of the most important is natural language processing, where ILP has been used to extract complex relational information from text. These results are summarized in Chapter 23.

19.6 SUMMARY

This chapter has investigated various ways in which prior knowledge can help an agent to learn from new experiences. Because much prior knowledge is expressed in terms of relational models rather than attribute-based models, we have also covered systems that allow learning of relational models. The important points are:

- The use of prior knowledge in learning leads to a picture of **cumulative learning**, in which learning agents improve their learning ability as they acquire more knowledge.
- Prior knowledge helps learning by eliminating otherwise consistent hypotheses and by

“filling in” the explanation of examples, thereby allowing for shorter hypotheses. These contributions often result in faster learning from fewer examples.

- Understanding the different logical roles played by prior knowledge, as expressed by **entailment constraints**, helps to define a variety of learning techniques.
- **Explanation-based learning** (EBL) extracts general rules from single examples by *explaining* the examples and generalizing the explanation. It provides a deductive method for turning first-principles knowledge into useful, efficient, special-purpose expertise.
- **Relevance-based learning** (RBL) uses prior knowledge in the form of determinations to identify the relevant attributes, thereby generating a reduced hypothesis space and speeding up learning. RBL also allows deductive generalizations from single examples.
- **Knowledge-based inductive learning** (KBIL) finds inductive hypotheses that explain sets of observations with the help of background knowledge.
- **Inductive logic programming** (ILP) techniques perform KBIL on knowledge that is expressed in first-order logic. ILP methods can learn relational knowledge that is not expressible in attribute-based systems.
- ILP can be done with a top-down approach of refining a very general rule or through a bottom-up approach of inverting the deductive process.
- ILP methods naturally generate new predicates with which concise new theories can be expressed and show promise as general-purpose scientific theory formation systems.

BIBLIOGRAPHICAL AND HISTORICAL NOTES

Although the use of prior knowledge in learning would seem to be a natural topic for philosophers of science, little formal work was done until quite recently. *Fact, Fiction, and Forecast*, by the philosopher Nelson Goodman (1954), refuted the earlier supposition that induction was simply a matter of seeing enough examples of some universally quantified proposition and then adopting it as a hypothesis. Consider, for example, the hypothesis “All emeralds are grue,” where *grue* means “green if observed before time t , but blue if observed thereafter.” At any time up to t , we might have observed millions of instances confirming the rule that emeralds are grue, and no disconfirming instances, and yet we are unwilling to adopt the rule. This can be explained only by appeal to the role of relevant prior knowledge in the induction process. Goodman proposes a variety of different kinds of prior knowledge that might be useful, including a version of determinations called **overhypotheses**. Unfortunately, Goodman’s ideas were never pursued in machine learning.

The **current-best-hypothesis** approach 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.*, 1986).

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).

Initial enthusiasm for EBL was tempered by Minton's finding (1988) that, without extensive extra work, EBL could easily slow down a program significantly. Formal probabilistic analysis of the expected payoff of EBL can be found in Greiner (1989) and Subramanian and Feldman (1990). An excellent survey of early work on EBL 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 rediscovered and given a full logical analysis by Davies and Russell (Davies, 1985; Davies and Russell, 1987). Their role as prior knowledge in inductive learning was explored by Russell and Grosop (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, due to Almuallim and Dietterich (1991). Tadepalli (1993) describes a very 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. The inverse resolution method also appears in (Russell, 1986), with a simple algorithm given in a footnote. 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. 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 Lavrauc and Duzeroski (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 intractable. However, with better understanding of the importance of syntactic restrictions on clauses, positive results have been obtained even for clauses with recursion (Duzeroski *et al.*, 1992). Learnability results for ILP are surveyed by Kietz and Duzeroski (1994) and Cohen and Page (1995).

DISCOVERY SYSTEM

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 784 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. Design and denomination determine the mass of a coin.
- b. For a given program, input determines output.
- c. Climate, food intake, exercise, and metabolism determine weight gain and loss.
- d. 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 LEARNING PROBABILISTIC MODELS

In which we view learning as a form of uncertain reasoning from observations.

Chapter 13 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, by formulating 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. Some of the material in this chapter is fairly mathematical, although the general lessons can be understood without plunging into the details. It may benefit the reader to review Chapters 13 and 14 and peek at 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 in this chapter are probabilistic theories of how the domain works, including logical theories as a special case.

Consider a simple example. Our favorite Surprise candy comes in two flavors: cherry (yum) and lime (ugh). The manufacturer has a peculiar sense of humor and wraps each piece of candy in the same opaque wrapper, regardless of flavor. The candy is sold in very large bags, of which there are known to be five kinds—again, indistinguishable from the outside:

- h_1 : 100% cherry,
- h_2 : 75% cherry + 25% lime,
- h_3 : 50% cherry + 50% lime,
- h_4 : 25% cherry + 75% lime,
- h_5 : 100% lime .

Given a new bag of candy, the random variable H (for *hypothesis*) denotes the type of the bag, with possible values h_1 through h_5 . H is not directly observable, of course. As the pieces of candy are opened and inspected, data are revealed— D_1, D_2, \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 \mathbf{d} ; then the probability of each hypothesis is obtained by Bayes’ rule:

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

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

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

HYPOTHESIS PRIOR
LIKELIHOOD

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, $P(\mathbf{d} | h_i)$.

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.** (see page 708), 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_5 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 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.

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 20.2.

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.

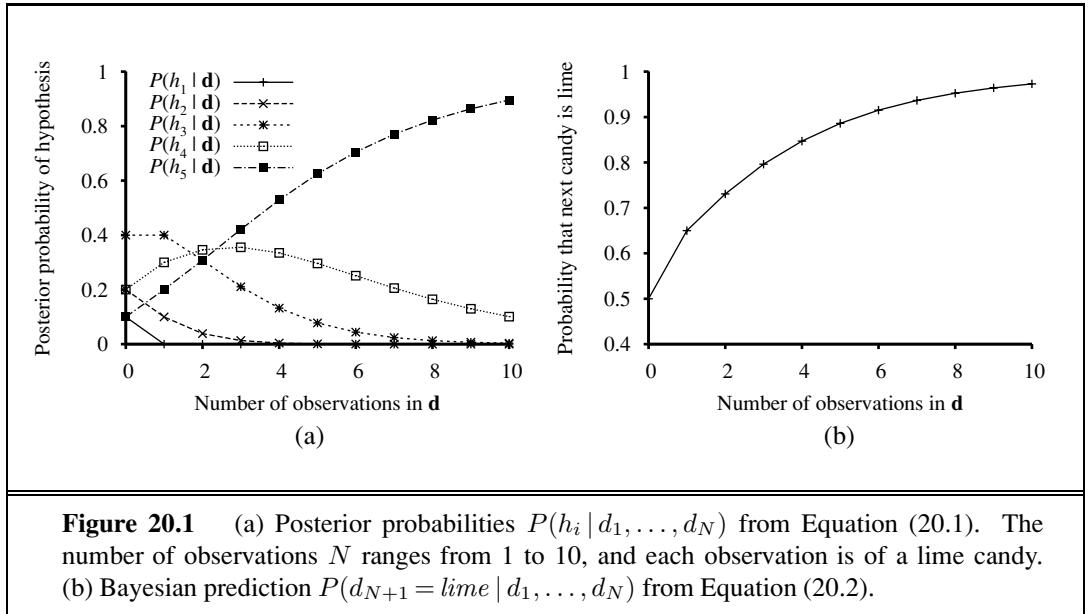


Figure 20.1 (a) Posterior probabilities $P(h_i | d_1, \dots, d_N)$ from Equation (20.1). The number of observations N ranges from 1 to 10, and each observation is of a lime candy. (b) Bayesian prediction $P(d_{N+1} = \text{lime} | d_1, \dots, d_N)$ from Equation (20.2).



The example shows that *the Bayesian prediction eventually agrees with the true hypothesis*. 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, under certain technical conditions, eventually vanish. This happens 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 important, the Bayesian prediction is *optimal*, whether the data set be small or large. Given the hypothesis prior, any other prediction is expected to 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(b). 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 learning, 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 tradeoff between the complexity of a hypothesis and its degree of fit to the data.



We can see the effect of this tradeoff 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 tradeoff 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 18.3.4, 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. Whereas MAP learning expresses simplicity by assigning higher probabilities to simpler hypotheses, MDL expresses it directly by counting the bits in a binary encoding of the hypotheses and data.

MAXIMUM-LIKELIHOOD

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.

20.2 LEARNING WITH COMPLETE DATA

DENSITY ESTIMATION

The general task of learning a probability model, given data that are assumed to be generated from that model, is called **density estimation**. (The term applied originally to probability density functions for continuous variables, but is used now for discrete distributions too.)

COMPLETE DATA

This section covers the simplest case, where we have **complete data**. Data are complete when each data point contains values for every variable in the probability model being learned. We focus on **parameter learning**—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. We will also look briefly at the problem of learning structure and at nonparametric density estimation.

20.2.1 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; the fraction could be anywhere between 0 and 1. In that case, we have a continuum of hypotheses. The **parameter** in this case, which we call θ , is the proportion of cherry candies, and the hypothesis is h_θ . (The proportion of limes is just $1 - \theta$.) If we assume that all proportions are equally likely *a priori*, then a maximum-likelihood approach is reasonable. If we model the situation with a Bayesian network, we need just one random variable, *Flavor* (the flavor of a randomly chosen candy from the bag). It has values *cherry* and *lime*, where the probability of *cherry* is θ (see Figure 20.2(a)). Now suppose we unwrap N candies, of which c are cherries and $\ell = N - c$ are limes. According to Equation (20.3), the likelihood of this particular data set is

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

LOG LIKELIHOOD

The maximum-likelihood hypothesis is given by the value of θ that maximizes this expression. The same value is obtained by maximizing the **log likelihood**,

$$L(\mathbf{d} | h_\theta) = \log P(\mathbf{d} | h_\theta) = \sum_{j=1}^N \log P(d_j | h_\theta) = c \log \theta + \ell \log(1 - \theta).$$

(By taking logarithms, we reduce the product to a sum over the data, which is usually easier to maximize.) To find the maximum-likelihood value of θ , we differentiate L with respect to θ and set the resulting expression to zero:

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

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, a method with broad applicability:

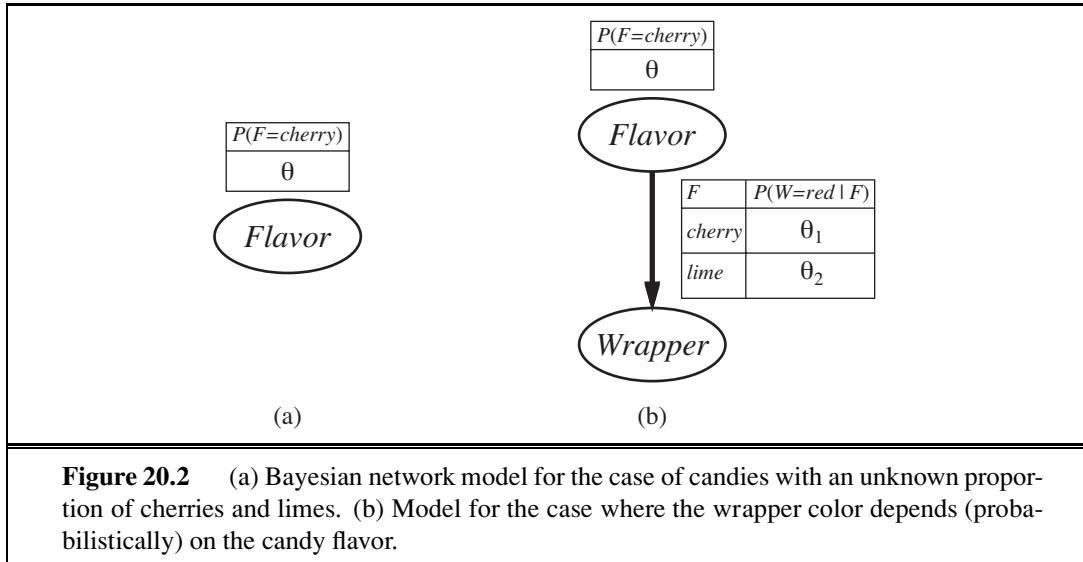


Figure 20.2 (a) Bayesian network model for the case of candies with an unknown proportion of cherries and limes. (b) Model for the case where the wrapper color depends (probabilistically) on the candy flavor.

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.

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 0.



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 513):

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

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

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

This looks pretty horrible, but taking logarithms helps:

$$L = [c \log \theta + \ell \log(1 - \theta)] + [r_c \log \theta_1 + g_c \log(1 - \theta_1)] + [r_\ell \log \theta_2 + g_\ell \log(1 - \theta_2)].$$

The benefit of taking logs is clear: the log likelihood is the sum of three terms, each of which contains a single parameter. When we take derivatives with respect to each parameter and set them to zero, we get three independent equations, each containing just one parameter:

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

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.* (See Exercise 20.6 for the nontabulated case, where each parameter affects several conditional probabilities.) 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.



20.2.2 Naive Bayes models

Probably the most common Bayesian network model used in machine learning is the **naive Bayes** model first introduced on page 499. 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 class *Flavor* and just one attribute, *Wrapper*.) 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 20.4) is one of the most effective

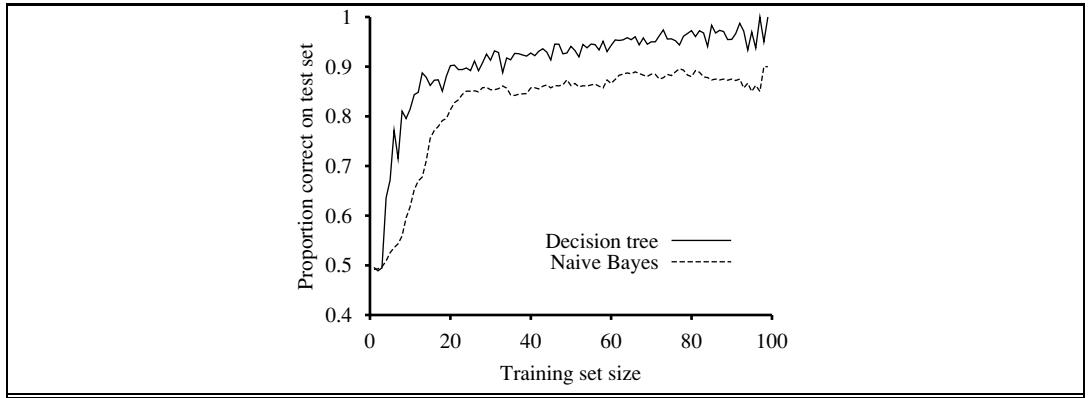


Figure 20.3 The learning curve for naive Bayes learning applied to the restaurant problem from Chapter 18; the learning curve for decision-tree learning is shown for comparison.



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 systems have no difficulty with noisy or missing data and can give probabilistic predictions when appropriate.

20.2.3 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 the parameters of continuous models from data. The principles for maximum-likelihood learning are identical in the continuous and discrete cases.

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.

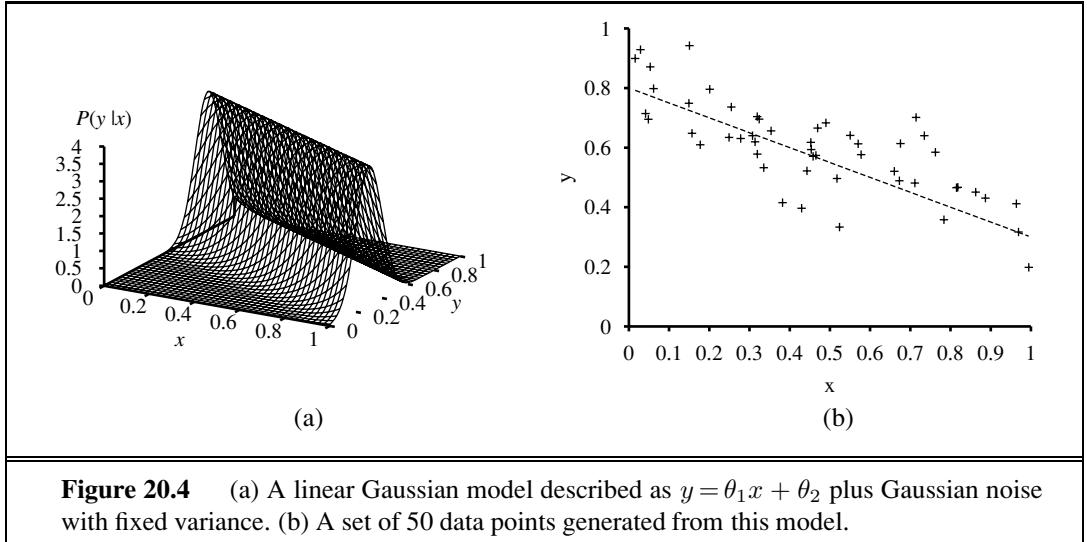


Figure 20.4 (a) A linear Gaussian model described as $y = \theta_1 x + \theta_2$ plus Gaussian noise with fixed variance. (b) A set of 50 data points generated from this model.

Now consider a linear Gaussian model with one continuous parent X and a continuous child Y . As explained on page 520, Y has a Gaussian distribution whose mean depends linearly on the value of X and whose standard deviation is fixed. To learn the conditional 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 20.5), we can find the maximum-likelihood values of the parameters. The point here is different. 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 $(y - (\theta_1 x + \theta_2))^2$ in the exponent of Equation (20.5). This is the L_2 loss, the squared error between the actual value y and the prediction $\theta_1 x + \theta_2$. This is the quantity minimized by the standard **linear regression** procedure described in Section 18.6. 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*.

20.2.4 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. It is more likely that the bag is a mixture of lime and cherry. The Bayesian approach to parameter learning starts by defining a prior probability distribution over the possible hypotheses. We call this the **hypothesis prior**. Then, as data arrives, the posterior probability distribution is updated.

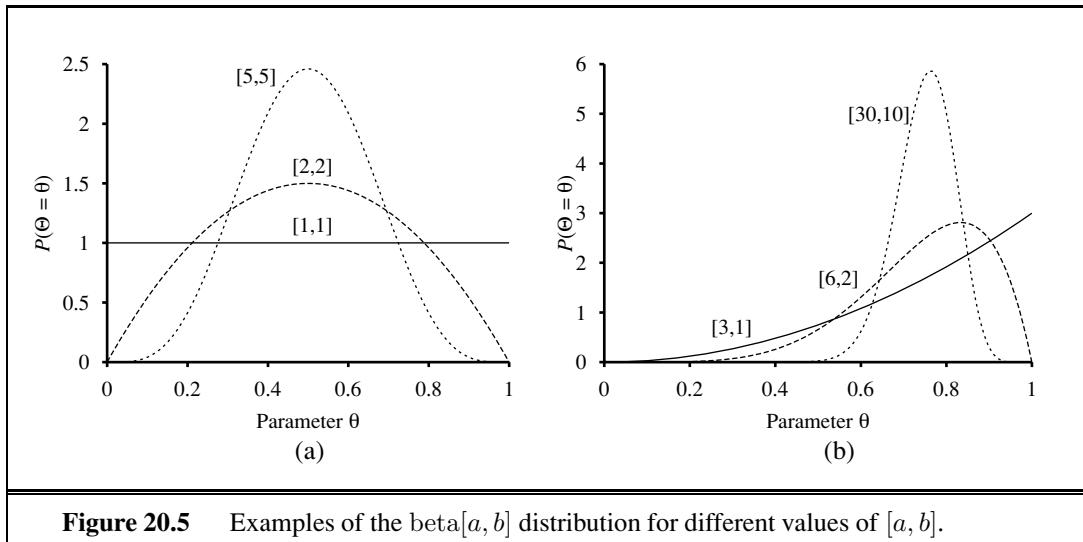


Figure 20.5 Examples of the beta $[a, b]$ distribution for different values of $[a, b]$.

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 that defines the hypothesis space; the hypothesis prior is just the prior distribution $\mathbf{P}(\Theta)$. Thus, $P(\Theta = \theta)$ is the prior probability that the bag has a fraction θ of cherry candies.

If the parameter θ can be any value between 0 and 1, then $\mathbf{P}(\Theta)$ must be a continuous distribution that is nonzero only between 0 and 1 and that integrates to 1. The uniform density $P(\theta) = \text{Uniform}[0, 1](\theta)$ 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**³ a and b such that

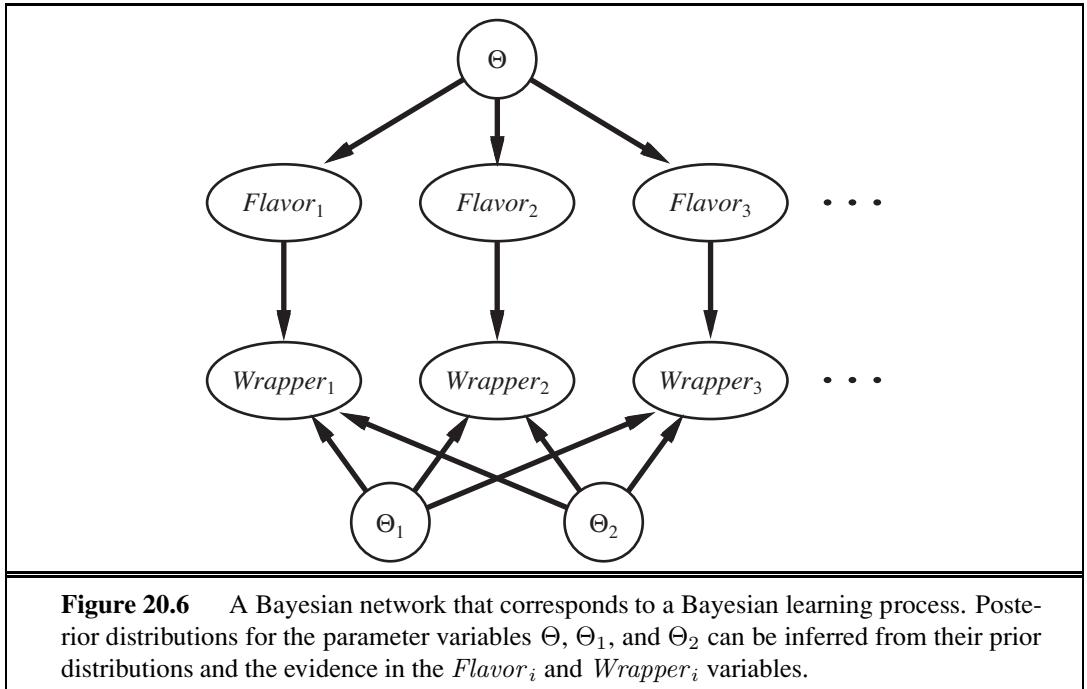
$$\text{beta}[a, b](\theta) = \alpha \theta^{a-1} (1 - \theta)^{b-1}, \quad (20.6)$$

for θ in the range $[0, 1]$. The normalization constant α , which makes the distribution integrate to 1, depends on a and b . (See Exercise 20.7.) 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. In other words, beta is closed under update. The beta family is called the **conjugate prior** for the family of distributions for a Boolean variable.⁴ Let's see how this works. Suppose we observe a cherry candy; then we have

³ They are called hyperparameters because they parameterize a distribution over θ , which is itself a parameter.

⁴ 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).



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

VIRTUAL COUNTS

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 the same answer as maximum-likelihood learning.

PARAMETER INDEPENDENCE

Now let us consider a more complicated case. 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. Figure 20.6 shows how we can incorporate the hypothesis prior and any data into one Bayesian network. The nodes $\Theta, \Theta_1, \Theta_2$ have no parents. But each time we make an observation of a wrapper and corresponding flavor of a piece of candy, we add a node $Flavor_i$, which is dependent on the flavor parameter Θ :

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

We also add a node $Wrapper_i$, which is dependent on Θ_1 and Θ_2 :

$$P(Wrapper_i = red | Flavor_i = cherry, \Theta_1 = \theta_1) = \theta_1$$

$$P(Wrapper_i = red | Flavor_i = lime, \Theta_2 = \theta_2) = \theta_2 .$$

Now, the entire Bayesian learning process can be formulated as an *inference* problem. We add new evidence nodes, then query the unknown nodes (in this case, $\Theta, \Theta_1, \Theta_2$). 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* —the inference algorithm for Bayesian networks. Of course, the nature of these networks is somewhat different from those of Chapter 14 because of the potentially huge number of evidence variables representing the training set and the prevalence of continuous-valued parameter variables.



20.2.5 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 understand how the structure of a Bayes net can be learned from data. This section gives 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 links. 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 in 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. But 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 assess 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 20.8). 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.

20.2.6 Density estimation with nonparametric models

NONPARAMETRIC
DENSITY ESTIMATION

It is possible to learn a probability model without making any assumptions about its structure and parameterization by adopting the nonparametric methods of Section 18.8. The task of **nonparametric density estimation** is typically done in continuous domains, such as that shown in Figure 20.7(a). The figure shows a probability density function on a space defined by two continuous variables. In Figure 20.7(b) we see a sample of data points from this density function. The question is, can we recover the model from the samples?

First we will consider **k -nearest-neighbors** models. (In Chapter 18 we saw nearest-neighbor models for classification and regression; here we see them for density estimation.) Given a sample of data points, to estimate the unknown probability density at a query point \mathbf{x} we can simply measure the density of the data points in the neighborhood of \mathbf{x} . Figure 20.7(b) shows two query points (small squares). For each query point we have drawn the smallest circle that encloses 10 neighbors—the 10-nearest-neighborhood. We can see that the central circle is large, meaning there is a low density there, and the circle on the right is small, meaning there is a high density there. In Figure 20.8 we show three plots of density estimation using k -nearest-neighbors, for different values of k . It seems clear that (b) is about right, while (a) is too spiky (k is too small) and (c) is too smooth (k is too big).

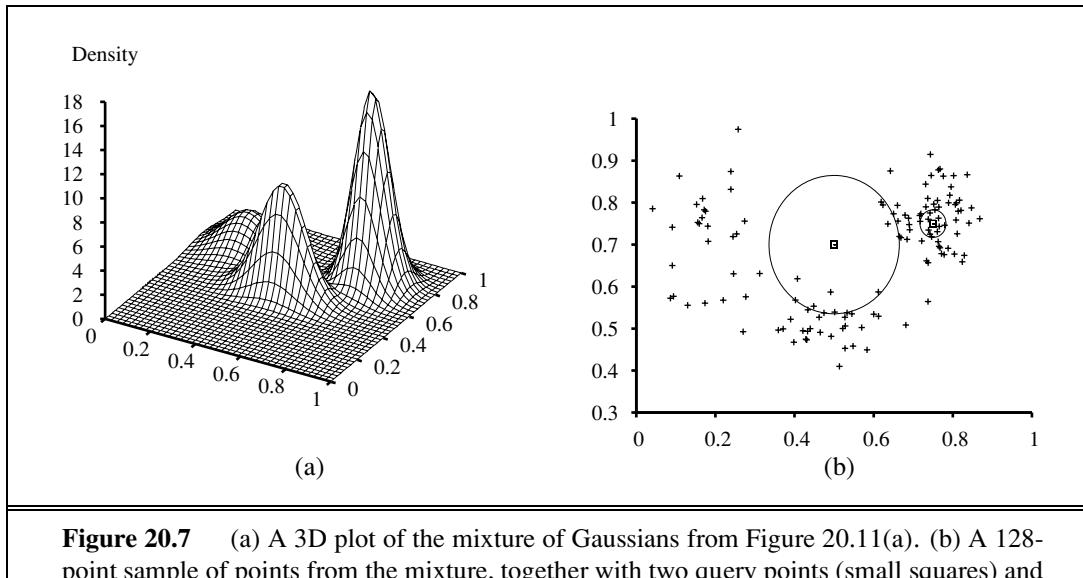


Figure 20.7 (a) A 3D plot of the mixture of Gaussians from Figure 20.11(a). (b) A 128-point sample of points from the mixture, together with two query points (small squares) and their 10-nearest-neighbors (medium and large circles).

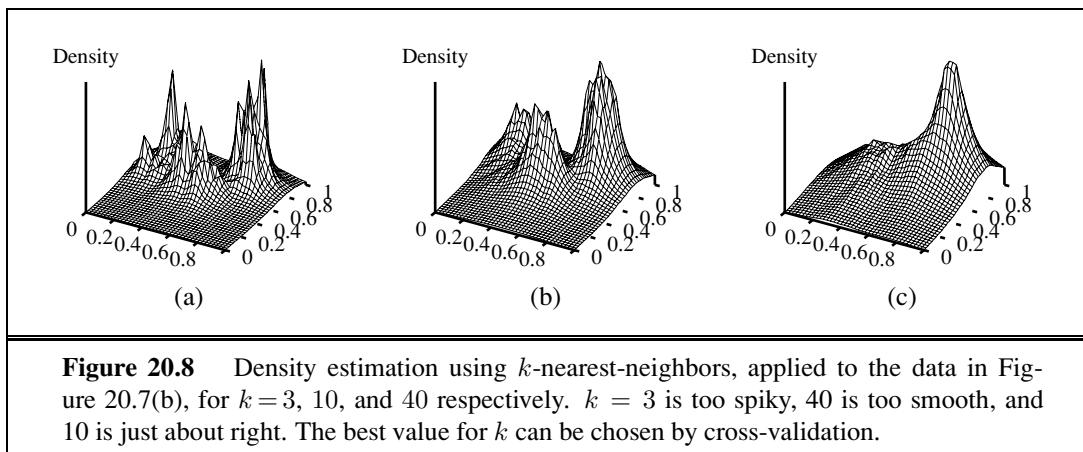


Figure 20.8 Density estimation using k -nearest-neighbors, applied to the data in Figure 20.7(b), for $k = 3, 10$, and 40 respectively. $k = 3$ is too spiky, 40 is too smooth, and 10 is just about right. The best value for k can be chosen by cross-validation.

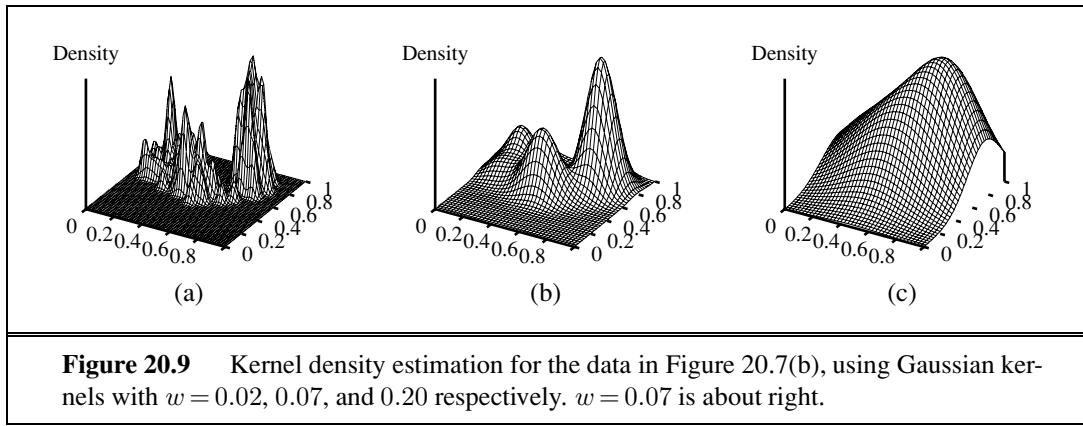


Figure 20.9 Kernel density estimation for the data in Figure 20.7(b), using Gaussian kernels with $w = 0.02, 0.07$, and 0.20 respectively. $w = 0.07$ is about right.

Another possibility is to use **kernel functions**, as we did for locally weighted regression. To apply a kernel model to density estimation, assume that each data point generates its own little density function, using a Gaussian kernel. The estimated density at a query point \mathbf{x} is then the average density as given by each kernel function:

$$P(\mathbf{x}) = \frac{1}{N} \sum_{j=1}^N \mathcal{K}(\mathbf{x}, \mathbf{x}_j) .$$

We will assume spherical Gaussians with standard deviation w along each axis:

$$\mathcal{K}(\mathbf{x}, \mathbf{x}_j) = \frac{1}{(w^2 \sqrt{2\pi})^d} e^{-\frac{D(\mathbf{x}, \mathbf{x}_j)^2}{2w^2}} ,$$

where d is the number of dimensions in \mathbf{x} and D is the Euclidean distance function. We still have the problem of choosing a suitable value for kernel width w ; Figure 20.9 shows values that are too small, just right, and too large. A good value of w can be chosen by using cross-validation.

20.3 LEARNING WITH HIDDEN VARIABLES: THE EM ALGORITHM

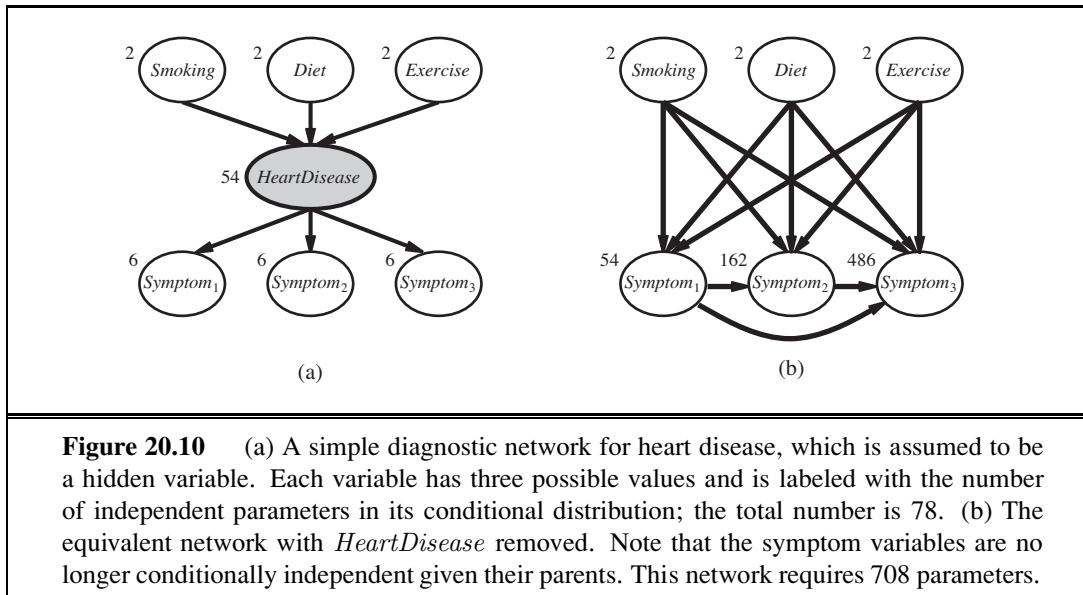
LATENT VARIABLE



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 physician's diagnosis, the treatment applied, and perhaps the outcome of the treatment, but they seldom contain a direct observation of the disease itself! (Note that the *diagnosis* is not the *disease*; it is a causal consequence of the observed symptoms, which are in turn caused by the disease.) One might ask, “If the disease is not observed, why not construct a model without it?” The answer appears in Figure 20.10, 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.

EXPECTATION-MAXIMIZATION

Hidden variables are important, but they do complicate the learning problem. In Figure 20.10(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 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.



20.3.1 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 types 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 and the creation of natural kinds for ordinary objects (see Chapter 12).

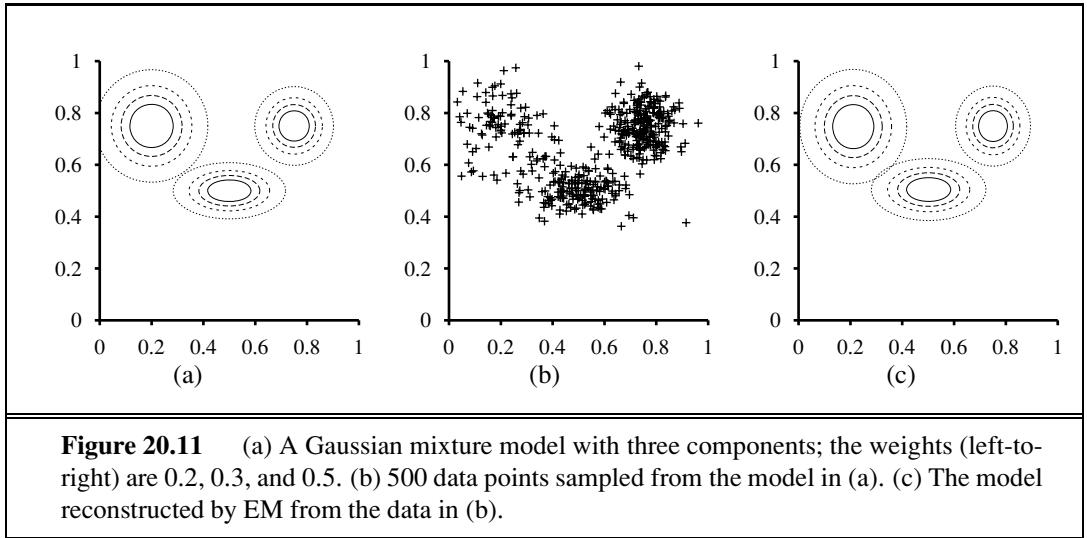
Unsupervised clustering begins with data. Figure 20.11(b) shows 500 data points, each of which specifies the values of two continuous attributes. The data points might correspond to stars, and the attributes might correspond to spectral intensities at two particular frequencies. Next, we need to understand 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 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

MIXTURE DISTRIBUTION COMPONENT

MIXTURE OF GAUSSIANS



$w_i = P(C = i)$ (the weight of each component), μ_i (the mean of each component), and Σ_i (the covariance of each component). Figure 20.11(a) shows a mixture of three Gaussians; this mixture is in fact the source of the data in (b) as well as being the model shown in Figure 20.7(a) on page 815.

The unsupervised clustering problem, then, is to recover a mixture model like the one in Figure 20.11(a) from raw data like that in Figure 20.11(b). 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) (page 809) 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 $n_i = \sum_j p_{ij}$, the effective number of data points currently assigned to component i .
2. **M-step:** Compute the new mean, covariance, and component weights using the following steps in sequence:

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

INDICATOR VARIABLE

where N is the total number of data points. 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 20.11(a) is shown in Figure 20.11(c); it is virtually indistinguishable from the original model from which the data were generated. Figure 20.12(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 (that hold in most cases), 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.

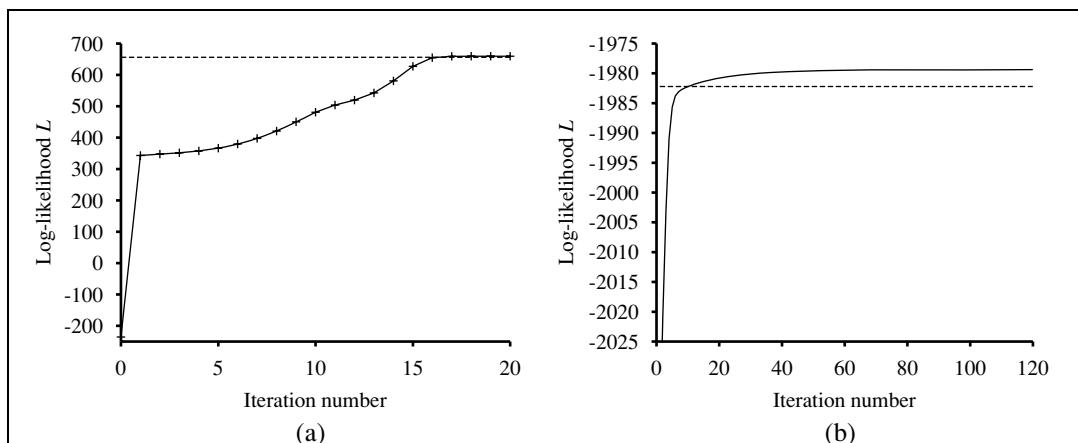


Figure 20.12 Graphs showing the log likelihood of the data, L , as a function of the EM iteration. The horizontal line shows the log likelihood according to the true model. (a) Graph for the Gaussian mixture model in Figure 20.11. (b) Graph for the Bayesian network in Figure 20.13(a).

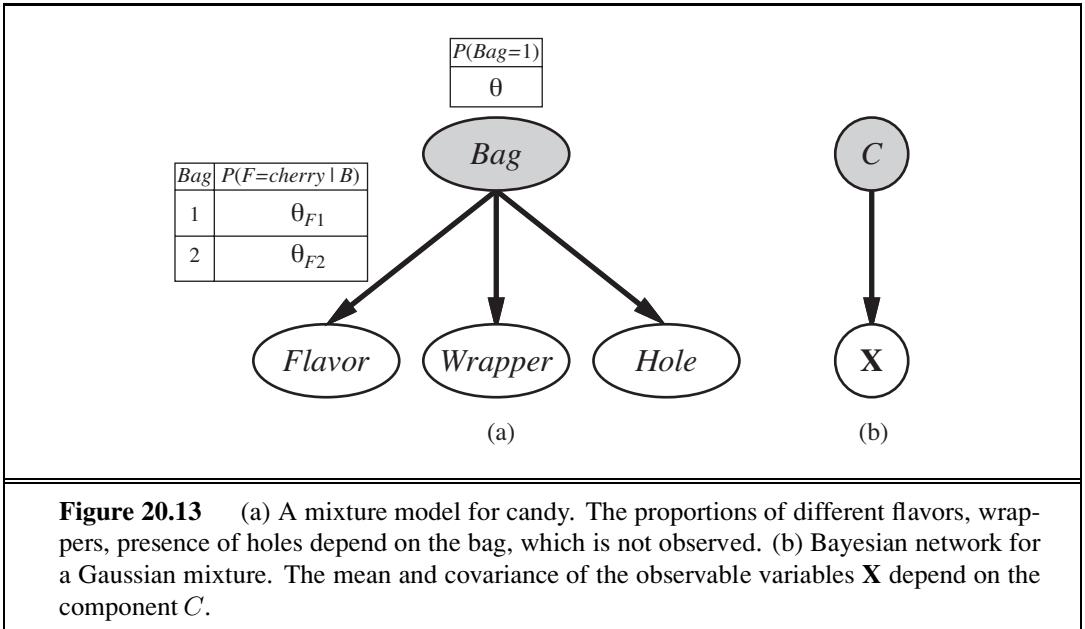


Figure 20.13 (a) A mixture model for candy. The proportions of different flavors, wrappers, presence of holes depend on the bag, which is not observed. (b) Bayesian network for a Gaussian mixture. The mean and covariance of the observable variables X depend on the component C .

Things do not always go as well as Figure 20.12(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. Sensible initialization also helps.

20.3.2 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 20.13 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. 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 or Bag 2 respectively; θ_{W1} and θ_{W2} give the probabilities that the wrapper is red; and θ_{H1} and θ_{H2} give the probabilities that the candy has a hole. Notice that the overall model is a mixture model. (In fact, we can also model the mixture of Gaussians as a Bayesian network, as shown in Figure 20.13(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 as follows:

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

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 arbitrarily choose⁵

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

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:

$$\theta^{(1)} = \frac{1}{N} \sum_{j=1}^N \frac{P(\text{flavor}_j | Bag = 1)P(\text{wrapper}_j | Bag = 1)P(\text{holes}_j | Bag = 1)P(Bag = 1)}{\sum_i P(\text{flavor}_j | Bag = i)P(\text{wrapper}_j | Bag = i)P(\text{holes}_j | Bag = i)P(Bag = i)}.$$

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 $\theta^{(1)} = 0.6124$.

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

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

⁵ It is better in practice to choose them randomly, to avoid local maxima due to symmetry.

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 20.12(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.* Here, “local” means that the CPT for each variable X_i can be learned from posterior probabilities involving just X_i and its parents \mathbf{U}_i . Defining θ_{ijk} to be the CPT parameter $P(X_i = x_{ij} | \mathbf{U}_i = \mathbf{u}_{ik})$, the update is given by the normalized expected counts as follows:

$$\theta_{ijk} \leftarrow \hat{N}(X_i = x_{ij}, \mathbf{U}_i = \mathbf{u}_{ik}) / \hat{N}(\mathbf{U}_i = \mathbf{u}_{ik}).$$

The expected counts are obtained by summing over the examples, computing the probabilities $P(X_i = x_{ij}, \mathbf{U}_i = \mathbf{u}_{ik})$ 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.

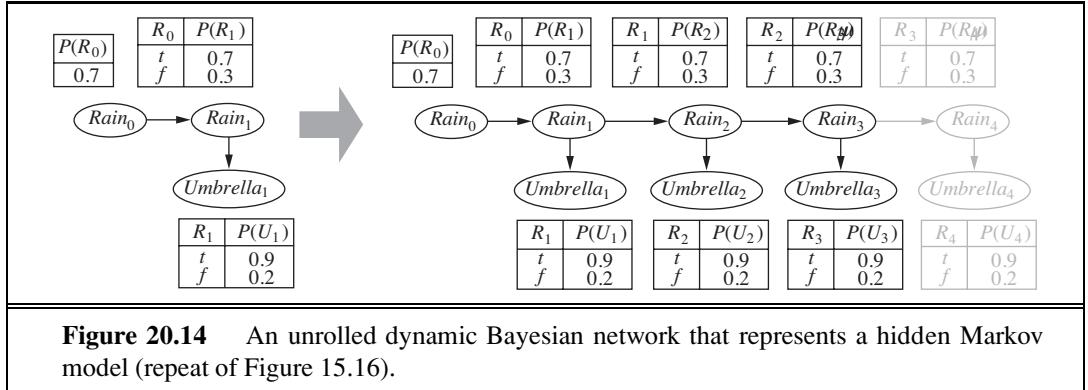
20.3.3 Learning hidden Markov models

Our final application of EM involves learning the transition probabilities in hidden Markov models (HMMs). Recall from Section 15.3 that a hidden Markov model can be represented by a dynamic Bayes net with a single discrete state variable, as illustrated in Figure 20.14. 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 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 :

$$\theta_{ij} \leftarrow \sum_t \hat{N}(X_{t+1} = j, X_t = i) / \sum_t \hat{N}(X_t = i).$$

The expected counts are computed by an 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 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. The evidence in a murder case is usually obtained *after* the crime (i.e., the transition from state i to state j) has taken place.

20.3.4 The general form of the EM algorithm

We have seen several instances of the EM algorithm. Each involves computing expected values of hidden variables for each example and then recomputing the parameters, using the expected values as if they were observed values. Let \mathbf{x} be all the observed values in all the examples, let \mathbf{Z} denote all the hidden variables for all the examples, and let $\boldsymbol{\theta}$ be all the parameters for the probability model. Then the EM algorithm is

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

This equation is the EM algorithm in a nutshell. The E-step is the computation of the summation, which is the expectation of the log likelihood of the “completed” data with respect to the distribution $P(\mathbf{Z} = \mathbf{z} | \mathbf{x}, \boldsymbol{\theta}^{(i)})$, which is the posterior over the hidden variables, given the data. The M-step is the maximization of this expected log likelihood with respect to the parameters. For mixtures of Gaussians, the hidden variables are the Z_{ij} s, where Z_{ij} is 1 if example j was generated by component i . For Bayes nets, Z_{ij} is the value of unobserved variable X_i in example j . For HMMs, Z_{jt} is the state of the sequence in example j at time t . Starting from the general form, it is possible to derive an EM algorithm for a specific application once the appropriate hidden variables have been identified.

As soon as we understand the general idea of EM, it becomes easy to derive all sorts of variants and improvements. For example, in many cases the E-step—the computation of posteriors over the hidden variables—is intractable, as in large Bayes nets. It turns out that one can use an *approximate* E-step and still obtain an effective learning algorithm. With a sampling algorithm such as MCMC (see Section 14.5), the learning process is very intuitive: each state (configuration of hidden and observed variables) visited by MCMC is treated exactly as if it were a complete observation. Thus, the parameters can be updated directly after each MCMC transition. Other forms of approximate inference, such as variational and loopy methods, have also proved effective for learning very large networks.

20.3.5 Learning Bayes net structures with hidden variables

In Section 20.2.5, we discussed the problem of learning Bayes net structures with complete data. When unobserved variables may be influencing the data that are observed, things get more difficult. In the simplest case, a human expert might tell the learning algorithm that certain hidden variables exist, leaving it to the algorithm to find a place for them in the network structure. For example, an algorithm might try to learn the structure shown in Figure 20.10(a) on page 817, given the information that *HeartDisease* (a three-valued variable) should be included in the model. As in the complete-data case, the overall algorithm has an outer loop that searches over structures and an inner loop that fits the network parameters given the structure.

If the learning algorithm is not told which hidden variables exist, then there are two choices: either pretend that the data is really complete—which may force the algorithm to learn a parameter-intensive model such as the one in Figure 20.10(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 preexisting 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 sample many possible network structures, thereby approximating 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.

For the complete-data case, the inner loop to learn the parameters 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 non-trivial Bayes net structures. Nonetheless, much work remains to be done before we can say that the structure-learning problem is solved.

20.4 SUMMARY

Statistical learning methods range from simple calculation of averages to the construction of complex models such as Bayesian networks. They have applications throughout computer science, engineering, computational biology, neuroscience, psychology, and physics. This chapter has presented some of the basic ideas and given a flavor of the mathematical underpinnings. The main points are as follows:

- **Bayesian learning** methods formulate learning as a form of probabilistic inference, using the observations to update a prior distribution over hypotheses. This approach provides a good way to implement Ockham’s razor, but quickly becomes intractable for complex hypothesis spaces.
- **Maximum a posteriori** (MAP) learning selects a single most likely hypothesis given the data. The hypothesis prior is still used and the method is often more tractable than full Bayesian learning.
- **Maximum-likelihood** learning simply selects the hypothesis that maximizes the likelihood of the data; it is equivalent to MAP learning with a uniform prior. In simple cases such as linear regression and fully observable Bayesian networks, maximum-likelihood solutions can be found easily in closed form. **Naive Bayes** learning is a particularly effective technique that scales well.
- When some variables are hidden, local maximum likelihood solutions can be found using the EM algorithm. Applications include clustering using mixtures of Gaussians, learning Bayesian networks, and learning hidden Markov models.
- Learning the structure of Bayesian networks is an example of **model selection**. This usually involves a discrete search in the space of structures. Some method is required for trading off model complexity against degree of fit.
- **Nonparametric models** represent a distribution using the collection of data points. Thus, the number of parameters grows with the training set. Nearest-neighbors methods look at the examples nearest to the point in question, whereas **kernel** methods form a distance-weighted combination of all the examples.

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. 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 embodied in the TETRAD package for Bayes net learning. 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). Several algorithmic advances since that time have led to quite respectable performance in the complete-data case (Moore and Wong, 2003; Teyssier and Koller, 2005). One important component is an efficient data structure, the AD-tree, for caching counts over all possible combinations of variables and values (Moore and Lee, 1997). 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 Hartley (1958), who described the general idea of what was later called EM and gave several examples. Further impetus came from the Baum–Welch algorithm for HMM learning (Baum and Petrie, 1966), which is a special case of EM. The paper by Dempster, Laird, and Rubin (1977), which presented the EM algorithm in general form and analyzed its convergence, is one of the most cited papers in both computer science and statistics. (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.) 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).

For maximum-likelihood parameter learning in Bayes nets with hidden variables, EM and gradient-based methods were introduced around the same time by Lauritzen (1995), Russell *et al.* (1995), and Binder *et al.* (1997a). The structural EM algorithm was developed by Friedman (1998) and applied to maximum-likelihood learning of Bayes net structures with

CAUSAL NETWORK

DIRICHLET PROCESS

GAUSSIAN PROCESS

latent variables. Friedman and Koller (2003) describe Bayesian structure learning.

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 a 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.

Nonparametric 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. There is also a rapidly growing literature on nonparametric Bayesian methods, originating with the seminal work of Ferguson (1973) on the **Dirichlet process**, which can be thought of as a distribution over Dirichlet distributions. These methods are particularly useful for mixtures with unknown numbers of components. Ghahramani (2005) and Jordan (2005) provide useful tutorials on the many applications of these ideas to statistical learning. The text by Rasmussen and Williams (2006) covers the **Gaussian process**, which gives a way of defining prior distributions over the space of continuous functions.

The material in this chapter brings together work from the fields of statistics and pattern recognition, 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). Bishop (2007) and Hastie *et al.* (2009) provide an excellent introduction to statistical machine learning. For pattern classification, the classic text for many years has been Duda and Hart (1973), now updated (Duda *et al.*, 2001). The annual NIPS (Neural Information Processing Conference) conference, whose proceedings are published as the series *Advances in Neural Information Processing Systems*, is now dominated by Bayesian papers. 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*. Specifically Bayesian venues include the Valencia International Meetings on Bayesian Statistics and the journal *Bayesian Analysis*.

EXERCISES

20.1 The data used for Figure 20.1 on page 804 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 $P(h_i | d_1, \dots, d_N)$ and $P(D_{N+1} = \text{lime} | d_1, \dots, d_N)$. Comment on your results.

20.2 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.

20.3 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?

20.4 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.

20.5 Consider N 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.

20.6 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.)

20.7 This exercise investigates properties of the Beta distribution defined in Equation (20.6).

- 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?

GAMMA FUNCTION

20.8 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.

20.9 Consider a single Boolean random variable Y (the “classification”). Let the prior probability $P(Y = \text{true})$ be π . Let’s try to find π , given a training set $D = (y_1, \dots, y_N)$ with N independent samples of Y . Furthermore, suppose p of the N are positive and n of the N are negative.

- a. Write down an expression for the likelihood of D (i.e., the probability of seeing this particular sequence of examples, given a fixed value of π) in terms of π , p , and n .
- b. By differentiating the log likelihood L , find the value of π that maximizes the likelihood.
- c. Now suppose we add in k Boolean random variables X_1, X_2, \dots, X_k (the “attributes”) that describe each sample, and suppose we assume that the attributes are conditionally independent of each other given the goal Y . Draw the Bayes net corresponding to this assumption.
- d. Write down the likelihood for the data including the attributes, using the following additional notation:
 - α_i is $P(X_i = \text{true}|Y = \text{true})$.
 - β_i is $P(X_i = \text{true}|Y = \text{false})$.
 - p_i^+ is the count of samples for which $X_i = \text{true}$ and $Y = \text{true}$.
 - n_i^+ is the count of samples for which $X_i = \text{false}$ and $Y = \text{true}$.
 - p_i^- is the count of samples for which $X_i = \text{true}$ and $Y = \text{false}$.
 - n_i^- is the count of samples for which $X_i = \text{false}$ and $Y = \text{false}$.

[Hint: consider first the probability of seeing a single example with specified values for X_1, X_2, \dots, X_k and Y .]

- e. By differentiating the log likelihood L , find the values of α_i and β_i (in terms of the various counts) that maximize the likelihood and say in words what these values represent.
- f. Let $k = 2$, and consider a data set with 4 all four possible examples of the XOR function. Compute the maximum likelihood estimates of $\pi, \alpha_1, \alpha_2, \beta_1$, and β_2 .
- g. Given these estimates of $\pi, \alpha_1, \alpha_2, \beta_1$, and β_2 , what are the posterior probabilities $P(Y = \text{true}|x_1, x_2)$ for each example?

20.10 Consider the application of EM to learn the parameters for the network in Figure 20.13(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 821 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).

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, 19, and 20 covered methods that learn functions, logical theories, and probability models from examples. In this chapter, we will study how agents can learn *what to do* in the absence of labeled examples of what to do.



REINFORCEMENT

Consider, for example, the problem of learning to play chess. A supervised learning agent needs to be told the correct move for each position it encounters, but such feedback is seldom available. In the absence of feedback from a teacher, an agent can learn a transition model for its own moves and can perhaps learn to predict the opponent's moves, but *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 (accidentally) checkmates the opponent, and that something bad has happened when it is checkmated—or vice versa, if the game is suicide chess. 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 environments 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. Thus, the agent faces an unknown Markov decision process. We will consider three of the agent designs first introduced in Chapter 2:

- A **utility-based agent** learns a utility function on states and uses it to select actions that maximize the expected outcome utility.
- A **Q-learning** agent learns an **action-utility function**, or **Q-function**, giving the expected utility of taking a given action in a given state.
- A **reflex agent** learns a policy that maps directly from states to actions.

A utility-based agent must also have a model of the environment in order to make decisions, because it must know the states to which its actions will lead. For example, in order to make use of a backgammon evaluation function, a backgammon program must know what its legal moves are *and how they affect the board position*. Only in this way can it apply the utility function to the outcome states. A Q-learning agent, on the other hand, can compare the expected utilities for 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.

Q-LEARNING
Q-FUNCTION

PASSIVE LEARNING

ACTIVE LEARNING
EXPLORATION

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.

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** $P(s' | s, a)$, 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.

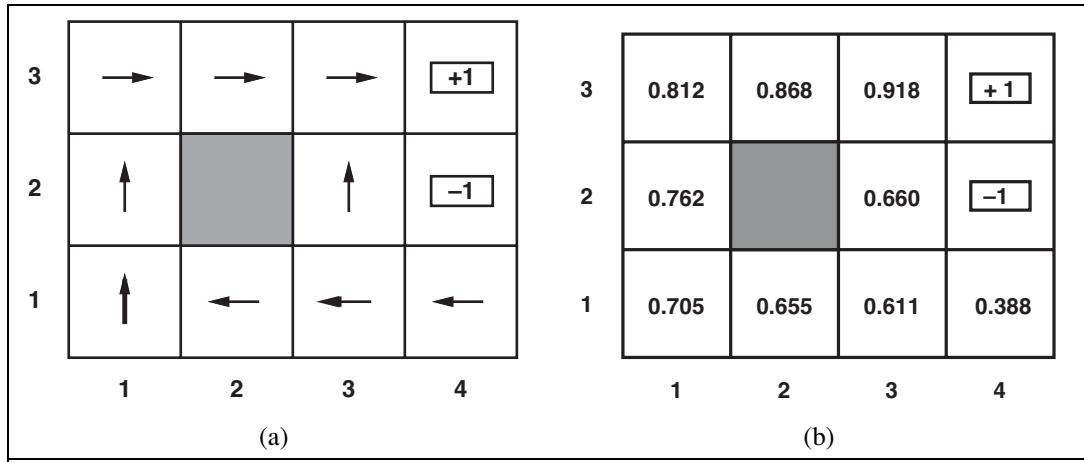


Figure 21.1 (a) A policy π for the 4×3 world; this policy happens to be optimal with rewards of $R(s) = -0.04$ in the nonterminal states and no discounting. (b) The utilities of the states in the 4×3 world, given policy π .

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:

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

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 (17.2) on page 650, we write

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

where $R(s)$ is the reward for a state, S_t (a random variable) is the state reached at time t when executing policy π , and $S_0 = s$. We will include a **discount factor** γ in all of our equations, but for the 4×3 world we will set $\gamma = 1$.

21.2.1 Direct utility estimation

DIRECT UTILITY
ESTIMATION
ADAPTIVE CONTROL
THEORY
REWARD-TO-GO

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 (called the expected **reward-to-go**), and each trial provides a *sample* of this quantity 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. 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)):

$$U^\pi(s) = R(s) + \gamma \sum_{s'} P(s' | s, \pi(s)) U^\pi(s') . \quad (21.2)$$

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 for U in a hypothesis space 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.

```

function PASSIVE-ADP-AGENT(percept) returns an action
  inputs: percept, a percept indicating the current state  $s'$  and reward signal  $r'$ 
  persistent:  $\pi$ , a fixed policy
     $mdp$ , an MDP with model  $P$ , rewards  $R$ , discount  $\gamma$ 
     $U$ , a table of utilities, initially empty
     $N_{sa}$ , a table of frequencies for state-action pairs, initially zero
     $N_{s'|sa}$ , a table of outcome frequencies given state-action pairs, initially zero
     $s, a$ , the previous state and action, initially null

  if  $s'$  is new then  $U[s'] \leftarrow r'; R[s'] \leftarrow r'$ 
  if  $s$  is not null then
    increment  $N_{sa}[s, a]$  and  $N_{s'|sa}[s', s, a]$ 
    for each  $t$  such that  $N_{s'|sa}[t, s, a]$  is nonzero do
       $P(t | s, a) \leftarrow N_{s'|sa}[t, s, a] / N_{sa}[s, a]$ 
     $U \leftarrow \text{POLICY-EVALUATION}(\pi, U, mdp)$ 
  if  $s'.\text{TERMINAL?}$  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. The POLICY-EVALUATION function solves the fixed-policy Bellman equations, as described on page 657.

21.2.2 Adaptive dynamic programming

ADAPTIVE DYNAMIC
PROGRAMMING

An **adaptive dynamic programming** (or ADP) agent takes advantage of the constraints among the utilities of states by learning the transition model that connects them and solving the corresponding Markov decision process using a dynamic programming method. For a passive learning agent, this means plugging the learned transition model $P(s' | s, \pi(s))$ 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 657), 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 $P(s' | s, a)$ from the frequency with which s' is reached when executing a in s . For example, in the three trials given on page 832, *Right* is executed three times in (1,3) and two out of three times the resulting state is (2,3), so $P((2, 3) | (1, 3), Right)$ is estimated to be 2/3.

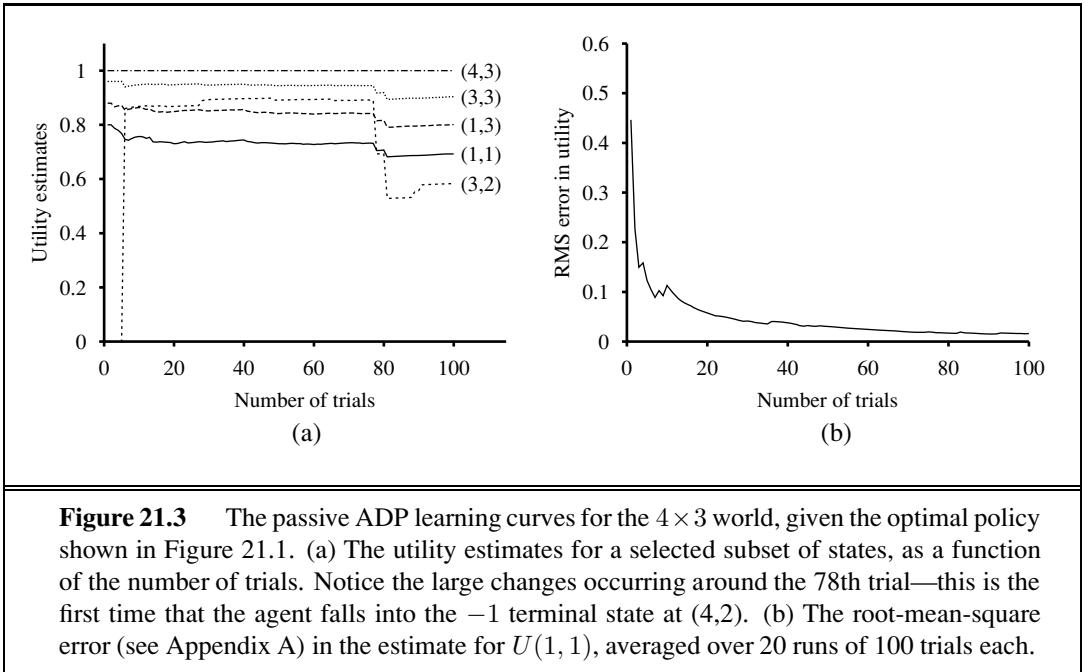


Figure 21.3 The passive ADP learning curves for the 4×3 world, given the optimal policy shown in Figure 21.1. (a) The utility estimates for a selected subset of states, as a function of the number of trials. Notice the large changes occurring around the 78th trial—this is the first time that the agent falls into the -1 terminal state at $(4,2)$. (b) The root-mean-square error (see Appendix A) in the estimate for $U(1, 1)$, averaged over 20 runs of 100 trials each.

The full agent program for a passive ADP agent is shown in Figure 21.2. Its performance on the 4×3 world is shown in Figure 21.3. In terms of how quickly its value estimates improve, the ADP agent is limited only by 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, intractable for large state spaces. In backgammon, for example, it would involve solving roughly 10^{50} equations in 10^{50} unknowns.

A reader familiar with the Bayesian learning ideas of Chapter 20 will have noticed that the algorithm in Figure 21.2 is using maximum-likelihood estimation to learn the transition model; moreover, by choosing a policy based solely on the *estimated* model it is acting *as if* the model were correct. This is not necessarily a good idea! For example, a taxi agent that didn't know about how traffic lights might ignore a red light once or twice without no ill effects and then formulate a policy to ignore red lights from then on. Instead, it might be a good idea to choose a policy that, while not optimal for the model estimated by maximum likelihood, works reasonably well for the whole range of models that have a reasonable chance of being the true model. There are two mathematical approaches that have this flavor.

The first approach, **Bayesian reinforcement learning**, assumes a prior probability $P(h)$ for each hypothesis h about what the true model is; the posterior probability $P(h | \mathbf{e})$ is obtained in the usual way by Bayes' rule given the observations to date. Then, if the agent has decided to stop learning, the optimal policy is the one that gives the highest expected utility. Let u_h^π be the expected utility, averaged over all possible start states, obtained by executing policy π in model h . Then we have

$$\pi^* = \operatorname{argmax}_\pi \sum_h P(h | \mathbf{e}) u_h^\pi.$$

In some special cases, this policy can even be computed! If the agent will continue learning in the future, however, then finding an optimal policy becomes considerably more difficult, because the agent must consider the effects of future observations on its beliefs about the transition model. The problem becomes a POMDP whose belief states are distributions over models. This concept provides an analytical foundation for understanding the exploration problem described in Section 21.3.

ROBUST CONTROL THEORY

The second approach, derived from **robust control theory**, allows for a *set* of possible models \mathcal{H} and defines an optimal robust policy as one that gives the best outcome in the *worst case* over \mathcal{H} :

$$\pi^* = \operatorname{argmax}_{\pi} \min_h u_h^\pi .$$

Often, the set \mathcal{H} will be the set of models that exceed some likelihood threshold on $P(h | \mathbf{e})$, so the robust and Bayesian approaches are related. Sometimes, the robust solution can be computed efficiently. There are, moreover, reinforcement learning algorithms that tend to produce robust solutions, although we do not cover them here.

21.2.3 Temporal-difference learning

Solving the underlying MDP as in the preceding section is not the only way to bring the Bellman equations to bear on the learning problem. Another way is to use the observed transitions to adjust the utilities 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 832. Suppose that, as a result of the first trial, the utility estimates are $U^\pi(1, 3) = 0.84$ and $U^\pi(2, 3) = 0.92$. Now, if this transition occurred all the time, we would expect the utilities to obey the equation

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

so $U^\pi(1, 3)$ would be 0.88. Thus, its current estimate of 0.84 might be a little low and should be increased. More generally, when a transition occurs from state s to state s' , we apply the 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.

All temporal-difference methods work by adjusting the utility estimates towards the ideal equilibrium that holds locally when the utility estimates are correct. 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 $U^\pi(s)$ itself will converge to the

```

function PASSIVE-TD-AGENT(percept) returns an action
  inputs: percept, a percept indicating the current state  $s'$  and reward signal  $r'$ 
  persistent:  $\pi$ , 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
    increment  $N_s[s]$ 
     $U[s] \leftarrow U[s] + \alpha(N_s[s])(r + \gamma U[s'] - U[s])$ 
  if  $s'.\text{TERMINAL?}$  then  $s, a, r \leftarrow \text{null}$  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. The step-size function $\alpha(n)$ is chosen to ensure convergence, as described in the text.

correct value.¹ 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 transition 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 (Equation (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 P . Although the observed transition makes only a local change in P , 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 “pseudoexperience” generated by simulating the current environment model. It is possible to extend the TD approach to use an environment model to generate several pseudoexperiences—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

¹ The technical conditions are given on page 725. In Figure 21.5 we have used $\alpha(n) = 60/(59 + n)$, which satisfies the conditions.

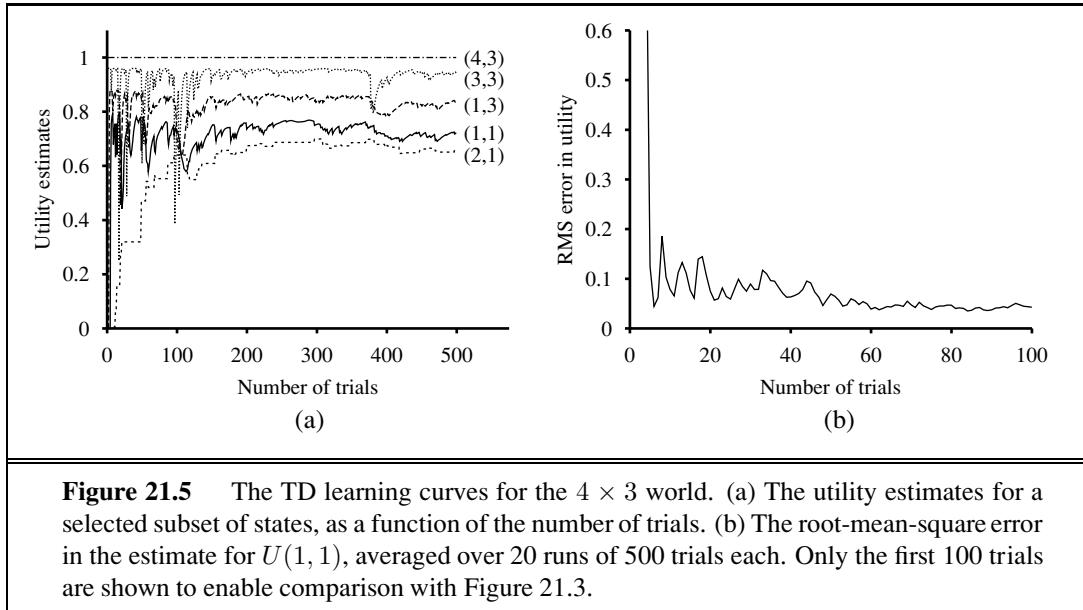


Figure 21.5 The TD learning curves for the 4×3 world. (a) The utility estimates for a selected subset of states, as a function of the number of trials. (b) The root-mean-square error in the estimate for $U(1, 1)$, averaged over 20 runs of 500 trials each. Only the first 100 trials are shown to enable comparison with Figure 21.3.

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. Even though the value iteration algorithm is efficient, it is intractable if we have, say, 10^{100} states. However, many of the necessary adjustments to the state values on each iteration will be 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 P 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 652, which we repeat here for convenience:

$$U(s) = R(s) + \gamma \max_a \sum_{s'} P(s' | s, a) U(s') . \quad (21.4)$$

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?

21.3.1 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(b).) After experimenting with minor variations, from the 276th trial onward it sticks to that policy, never learning the utilities of the other states and never finding the optimal route via (1,2), (1,3), and (2,3). We call this agent the **greedy agent**. Repeated experiments show that the greedy agent *very seldom* converges to the optimal policy for this environment and sometimes converges to really horrendous policies.

GREEDY AGENT

How can it be that choosing the optimal action leads to suboptimal results? The answer is that the learned model is not the same as the true environment; what is optimal in the 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?

EXPLOITATION
EXPLORATION

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.² An agent therefore must make a tradeoff between **exploitation** to maximize its reward—as reflected in its current utility estimates—and **exploration** to maxi-

² Notice the direct analogy to the theory of information value in Chapter 16.

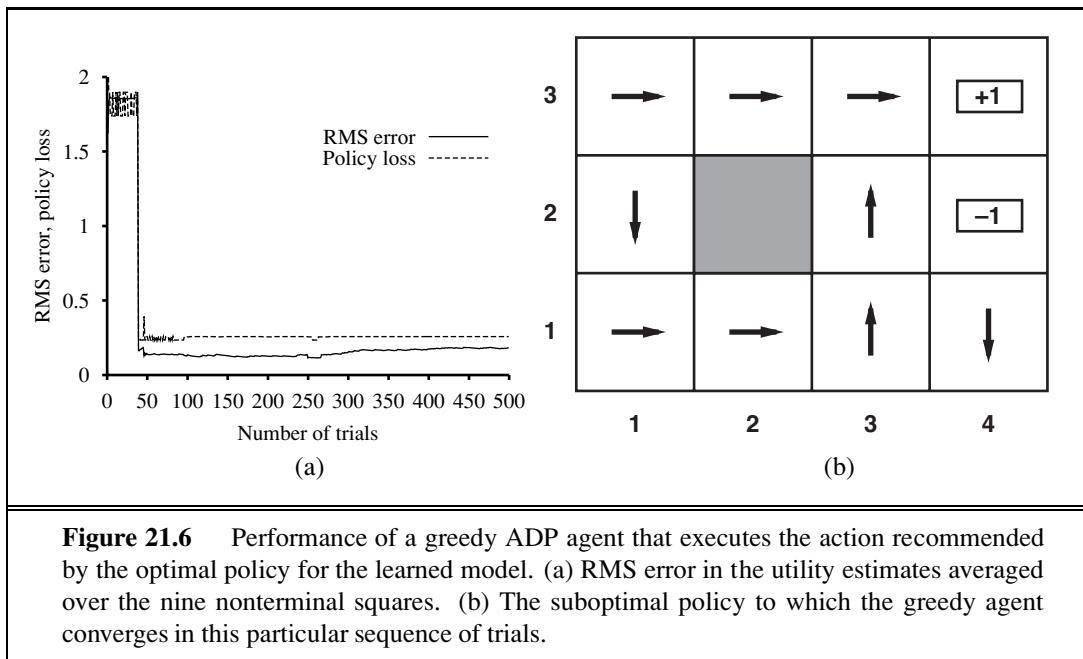


Figure 21.6 Performance of a greedy ADP agent that executes the action recommended by the optimal policy for the learned model. (a) RMS error in the utility estimates averaged over the nine nonterminal squares. (b) The suboptimal policy to which the greedy agent converges in this particular sequence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

mize 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? This question has been studied in depth in the subfield of statistical decision theory that deals with so-called **bandit problems**. (See sidebar.)

BANDIT PROBLEM

GLIE

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 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

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 $P(s' | s, a)$. 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; generally speaking, the higher the expected return and the higher the uncertainty in the utility of a given choice, the better. 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 it can be proven that genetic algorithms allocate coins optimally, given an appropriate set of independence assumptions.

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 $N(s, a)$ 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 (Equation (17.6) on page 652) to incorporate the optimistic estimate. The following equation does this:

$$U^+(s) \leftarrow R(s) + \gamma \max_a f \left(\sum_{s'} P(s' | s, a) U^+(s'), N(s, a) \right). \quad (21.5)$$

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 actions that have not been tried often and have low n). 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.

21.3.2 Learning an action-utility 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

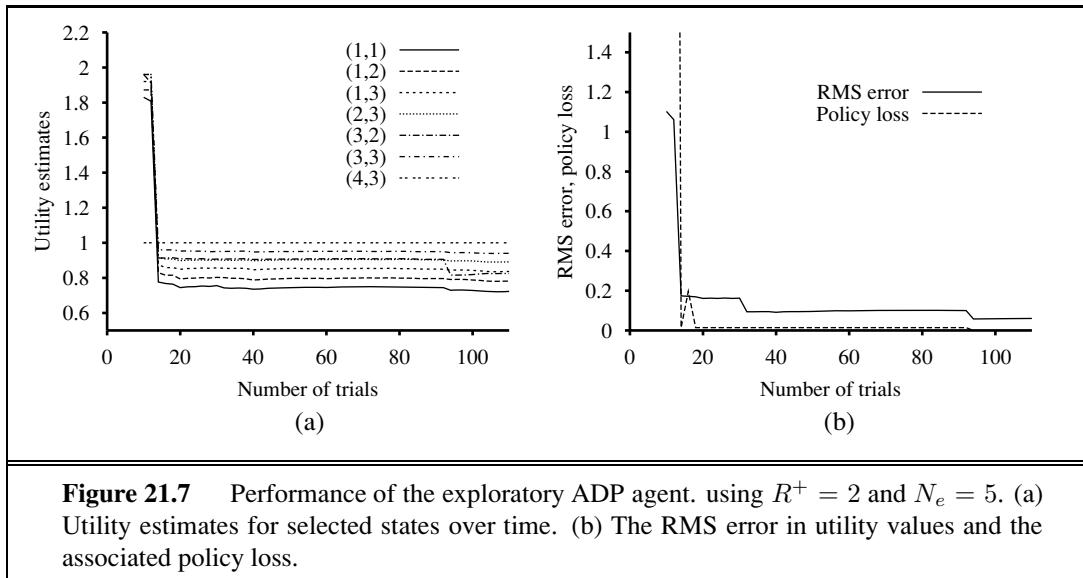


Figure 21.7 Performance of the exploratory ADP agent, using $R^+ = 2$ and $N_e = 5$. (a) Utility estimates for selected states over time. (b) The RMS error in utility values and the associated policy loss.

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**, which learns an action-utility representation instead of learning utilities. We will use the notation $Q(s, a)$ to denote the value of doing action a in state s . Q-values are directly related to utility values as follows:

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



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 of the form $P(s' | s, a)$, either for learning or for 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 Q-values are correct:

$$Q(s, a) = R(s) + \gamma \sum_{s'} P(s' | s, a) \max_{a'} Q(s', a'). \quad (21.7)$$

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 $P(s' | s, a)$. The temporal-difference approach, on the other hand, requires no model of state transitions—all

```

function Q-LEARNING-AGENT(percept) returns an action
  inputs: percept, a percept indicating the current state  $s'$  and reward signal  $r'$ 
  persistent:  $Q$ , a table of action values indexed by state and action, initially zero
     $N_{sa}$ , a table of frequencies for state-action pairs, initially zero
     $s, a, r$ , the previous state, action, and reward, initially null

  if TERMINAL?( $s$ ) then  $Q[s, \text{None}] \leftarrow r'$ 
  if  $s$  is not null then
    increment  $N_{sa}[s, a]$ 
     $Q[s, a] \leftarrow Q[s, a] + \alpha(N_{sa}[s, a])(r + \gamma \max_{a'} Q[s', a'] - Q[s, a])$ 
     $s, a, r \leftarrow s', \text{argmax}_{a'} f(Q[s', a'], N_{sa}[s', a']), r'$ 
  return  $a$ 

```

Figure 21.8 An exploratory Q-learning agent. It is an active learner that learns the value $Q(s, a)$ 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.

it needs are the Q values. The update equation for TD Q-learning is

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

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.

SARSA

Q-learning has a close relative called **SARSA** (for State-Action-Reward-State-Action). The update rule for SARSA is very similar to Equation (21.8):

$$Q(s, a) \leftarrow Q(s, a) + \alpha(R(s) + \gamma Q(s', a') - Q(s, a)), \quad (21.9)$$

OFF-POLICY
ON-POLICY

where a' is the action *actually taken* in state s' . The rule is applied at the end of each s, a, r, s', a' quintuplet—hence the name. The difference from Q-learning is quite subtle: whereas Q-learning backs up the *best* Q-value from the state reached in the observed transition, SARSA waits until an action is actually taken and backs up the Q-value for that action. Now, for a greedy agent that always takes the action with best Q-value, the two algorithms are identical. When exploration is happening, however, they differ significantly. Because Q-learning uses the best Q-value, it pays no attention to the actual policy being followed—it is an **off-policy** learning algorithm, whereas SARSA is an **on-policy** algorithm. Q-learning is more flexible than SARSA, in the sense that a Q-learning agent can learn how to behave well even when guided by a random or adversarial exploration policy. On the other hand, SARSA is more realistic: for example, if the overall policy is even partly controlled by other agents, it is better to learn a Q-function for what will actually happen rather than what the agent would like to happen.

Both Q-learning and SARSA learn the optimal policy for the 4×3 world, but do so at a much slower rate than the ADP agent. This is because the local updates do not enforce consistency among all the Q-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-utility 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. Backgammon and chess are tiny subsets of the real world, yet their state spaces contain on the order of 10^{20} and 10^{40} states, respectively. It would be absurd to suppose that one must visit all these states many times in order to learn how to play the game!

FUNCTION APPROXIMATION
BASIS FUNCTION

One way to handle such problems is to use **function approximation**, which simply means using any sort of representation for the Q-function other than a lookup 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 5 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^{40} 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.³ 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 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^{12} 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 tradeoff 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

$$\hat{U}_\theta(x, y) = \theta_0 + \theta_1 x + \theta_2 y. \quad (21.10)$$

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 18.)

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

$$\theta_i \leftarrow \theta_i - \alpha \frac{\partial E_j(s)}{\partial \theta_i} = \theta_i + \alpha (u_j(s) - \hat{U}_\theta(s)) \frac{\partial \hat{U}_\theta(s)}{\partial \theta_i}. \quad (21.11)$$

WIDROW-HOFF RULE
DELTA RULE

This is called the **Widrow-Hoff rule**, or the **delta rule**, for online least-squares. For the linear function approximator $\hat{U}_\theta(s)$ in Equation (21.10), we get three simple update rules:

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

³ 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 tradeoffs involved are currently not well understood.



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 parameters θ in response to an observed transition between two states 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 21.5 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 21.8.) If we put the +1 reward at (5,5), the true utility is more like a pyramid and the function approximator in Equation (21.10) 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 $\theta_3 f_3(x, y) = \theta_3 \sqrt{(x - x_g)^2 + (y - y_g)^2}$ 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 on page 836 and 21.8 on page 844) are given by

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

for utilities and

$$\theta_i \leftarrow \theta_i + \alpha [R(s) + \gamma \max_{a'} \hat{Q}_\theta(s', a') - \hat{Q}_\theta(s, a)] \frac{\partial \hat{Q}_\theta(s, a)}{\partial \theta_i} \quad (21.13)$$

for Q-values. For passive TD learning, the update rule can be shown to converge to the closest possible⁴ approximation to the true function when the function approximator is *linear* in the parameters. With active learning and *nonlinear* functions such as neural networks, all bets are off: 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 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

⁴ The definition of distance between utility functions is rather technical; see Tsitsiklis and Van Roy (1997).

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. Some practical examples are described in Section 21.6.

21.5 POLICY SEARCH

POLICY SEARCH

The final approach we will consider for reinforcement learning problems is called **policy search**. In some ways, policy search is the simplest of all the methods in this chapter: the idea is to keep twiddling the policy as long as its performance improves, then stop.

Let us begin with the policies themselves. Remember that a policy π is a function that maps states to actions. We are interested primarily in *parameterized* representations of π that have far fewer parameters than there are states in the state space (just as in the preceding section). For example, we could represent π by a collection of parameterized Q-functions, one for each action, and take the action with the highest predicted value:

$$\pi(s) = \max_a \hat{Q}_\theta(s, a). \quad (21.14)$$



Each Q-function could be a linear function of the parameters θ , as in Equation (21.10), 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 by the two methods may differ very substantially. (For example, the approximate Q-function defined by $\hat{Q}_\theta(s, a) = Q^*(s, a)/10$ gives optimal performance, even though it is not at all close to Q^* .) Another clear instance 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 . A value of θ that gives good results may be a long way from making \hat{U}_θ resemble the true utility function.

STOCHASTIC POLICY
SOFTMAX FUNCTION

One problem with policy representations of the kind given in Equation (21.14) is that the policy is a *discontinuous* function of the parameters when the actions are discrete. (For a continuous action space, the policy can be a smooth function of the parameters.) 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**:

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

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 . Softmax is a generalization of the logistic function (page 725) to multiple variables.

POLICY VALUE

POLICY GRADIENT

Now let us look at methods for improving the policy. We start with the simplest case: a deterministic policy and a deterministic environment. Let $\rho(\theta)$ be the **policy value**, i.e., the expected reward-to-go when π_θ is executed. If we can derive an expression for $\rho(\theta)$ in closed form, then we have 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, if $\rho(\theta)$ is not available in closed form, we can evaluate π_θ simply by executing it and observing the accumulated reward. We can follow the **empirical gradient** by hill climbing—i.e., evaluating the change in policy value for small increments in each parameter. 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 hill climbing, 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 reward $R(a)$ is obtained immediately after doing action a 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 hill climbing using lots of trials at each value of θ . It is still much slower than necessary, however.



Consider the following task: given two blackjack⁵ 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 idea, called **correlated sampling**, underlies a policy-search algorithm called PEGASUS (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.

21.6 APPLICATIONS OF REINFORCEMENT LEARNING

We now turn to examples of large-scale applications of reinforcement learning. We consider applications in game playing, where the transition model is known and the goal is to learn the utility function, and in robotics, where the model is usually unknown.

21.6.1 Applications to game playing

The first significant application of reinforcement learning was also the first significant learning program of any kind—the checkers 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 (21.12) 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 reached in self-play were ignored. This means that it is theoretically 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 checkers play.

Gerry Tesauro’s backgammon program TD-GAMMON (1992) forcefully illustrates the potential of reinforcement learning techniques. In earlier work (Tesauro and Sejnowski, 1989), Tesauro tried learning a neural network representation of $Q(s, a)$ directly from ex-

⁵ Also known as twenty-one or pontoon.

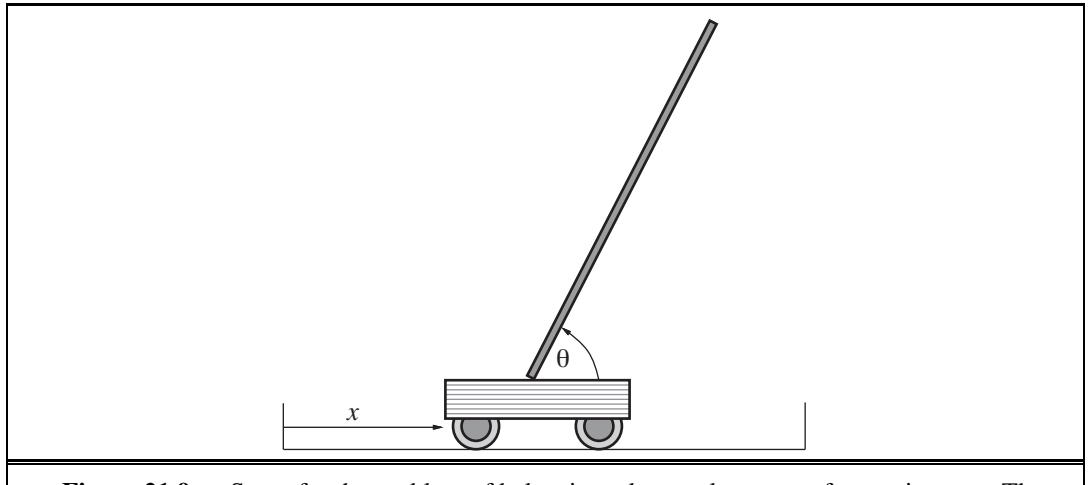


Figure 21.9 Setup for the problem of balancing a long pole on top of a moving cart. The cart can be jerked left or right by a controller that observes x , θ , \dot{x} , and $\ddot{\theta}$.

amples 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 (21.12), 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 nodes 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.”

21.6.2 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. Several 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 $\ddot{\theta}$ 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 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, or by using a continuous-state, nonlinear function approximator such as a neural network. Nowadays, balancing a *triple* inverted pendulum is a common exercise—a feat far beyond the capabilities of most humans.

Still more impressive is the application of reinforcement learning to helicopter flight (Figure 21.10). This work has generally used policy search (Bagnell and Schneider, 2001) as well as the PEGASUS algorithm with simulation based on a learned transition model (Ng *et al.*, 2004). Further details are given in Chapter 25.



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.)

21.7 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 P and a utility function U ; the model-free design, using an action-utility 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 can learn without requiring a transition model. Using a learned model to generate pseudoexperiences can, however, result in faster learning.
- Action-utility 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 variation 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 con-

tinuous, high-dimensional, partially observable environments in which successful behaviors may consist of thousands or even millions of primitive actions.

BIBLIOGRAPHICAL AND HISTORICAL NOTES

Turing (1948, 1950) proposed the reinforcement-learning approach, although he was not convinced of its effectiveness, writing, “the use of punishments and rewards can at best be a part of the teaching process.” Arthur Samuel’s work (1959) was probably the earliest successful machine learning research. Although this work was informal and had a number of flaws, it contained most of the modern ideas in reinforcement learning, including temporal differencing and function approximation. Around the same time, researchers in adaptive control theory (Widrow and Hoff, 1960), building on work by Hebb (1949), were training simple networks using the delta rule. (This early connection between neural networks and reinforcement learning may have led to the persistent misperception that the latter is a subfield of the former.) The cart–pole work of Michie and Chambers (1968) can also be seen as a reinforcement learning method with a function approximator. The psychological literature on reinforcement learning is much older; Hilgard and Bower (1975) provide a good survey. Direct evidence for the operation of reinforcement learning in animals has been provided by investigations into the foraging behavior of bees; there is a clear neural correlate of the reward signal in the form of a large neuron mapping from the nectar intake sensors directly to the motor cortex (Montague *et al.*, 1995). Research using single-cell recording suggests that the dopamine system in primate brains implements something resembling value function learning (Schultz *et al.*, 1997). The neuroscience text by Dayan and Abbott (2001) describes possible neural implementations of temporal-difference learning, while Dayan and Niv (2008) survey the latest evidence from neuroscientific and behavioral experiments.

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 utility 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 Bradtko 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 (least-squares temporal differencing) algorithm is an online algorithm for passive reinforcement learning that gives the same results as offline regression. Least-squares policy iteration, or LSPI (Lagoudakis and Parr, 2003), combines this idea with the policy iteration algorithm, yielding a robust, statistically efficient, model-free algorithm for learning policies.

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), while SARSA appeared in a technical report by Rummery and Niranjan (1994).

Bandit problems, which model the problem of exploration for nonsequential decisions, are analyzed in depth by Berry and Fristedt (1985). Optimal exploration strategies for several settings are obtainable using the technique called **Gittins indices** (Gittins, 1989). A variety of exploration methods for sequential decision problems are discussed by Barto *et al.* (1995). Kearns and Singh (1998) and Brafman and Tennenholtz (2000) describe algorithms that explore unknown environments and are guaranteed to converge on near-optimal policies in polynomial time. Bayesian reinforcement learning (Dearden *et al.*, 1998, 1999) provides another angle on both model uncertainty and exploration.

CMAC

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 approximator, 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 method of correlated sampling for comparing different configurations of a system was described formally by Kahn and Marshall (1953), but seems to have been known long before that. Its use in reinforcement learning is due to Van Roy (1998) and Ng and Jordan (2000); the latter paper also introduced the **PEGASUS** algorithm and proved its formal properties.

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, called **pseudorewards**, for “making progress.” For example, in soccer the real reward is for scoring a goal, but pseudorewards 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 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 the 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 11. 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. The partial-programming language for agent programs extends an ordinary programming language by adding primitives for unspecified choices that must be filled in by learning. 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 has been shown to solve large-scale problems—for example, policies that execute for 10^4 steps in state spaces of 10^{100} states with branching factors of 10^{30} (Marthi *et al.*, 2005). One key result (Dietterich, 2000) is that the hierarchical structure provides a natural *additive decomposition* of the overall utility function into terms that depend on small subsets of the variables defining the state space. This is somewhat analogous to the representation theorems underlying the conciseness of Bayes nets (Chapter 14).

The topic of distributed and multiagent reinforcement learning was not touched upon in the chapter but is of great current interest. In distributed RL, the aim is to devise methods by which multiple, coordinated agents learn to optimize a common utility function. For example,

REWARD SHAPING

PSEUDOREWARD

HIERARCHICAL
REINFORCEMENT
LEARNING

PARTIAL PROGRAM

SUBAGENT

can we devise methods whereby separate **subagents** for robot navigation and robot obstacle avoidance could cooperatively achieve a combined control system that is globally optimal? Some basic results in this direction have been obtained (Guestrin *et al.*, 2002; Russell and Zimdars, 2003). The basic idea is that each subagent learns its own Q-function from its own stream of rewards. For example, a robot-navigation component can receive rewards for making progress towards the goal, while the obstacle-avoidance component receives negative rewards for every collision. Each global decision maximizes the sum of Q-functions and the whole process converges to globally optimal solutions.

Multiagent RL is distinguished from distributed RL by the presence of agents who cannot coordinate their actions (except by explicit communicative acts) and who may not share the same utility function. Thus, multiagent RL deals with sequential game-theoretic problems or **Markov games**, as defined in Chapter 17. The consequent requirement for randomized policies is not a significant complication, as we saw on page 848. What *does* cause problems is the fact that, while an agent is learning to defeat its opponent’s policy, the opponent is changing its policy to defeat the agent. Thus, the environment is **nonstationary** (see page 568). Littman (1994) noted this difficulty when introducing the first RL algorithms for zero-sum Markov games. Hu and Wellman (2003) present a Q-learning algorithm for general-sum games that converges when the Nash equilibrium is unique; when there are multiple equilibria, the notion of convergence is not so easy to define (Shoham *et al.*, 2004).

APPRENTICESHIP
LEARNING

Sometimes the reward function is not easy to define. Consider the task of driving a car. There are extreme states (such as crashing the car) that clearly should have a large penalty. But beyond that, it is difficult to be precise about the reward function. However, it is easy enough for a human to drive for a while and then tell a robot “do it like that.” The robot then has the task of **apprenticeship learning**; learning from an example of the task done right, without explicit rewards. Ng *et al.* (2004) and Coates *et al.* (2009) show how this technique works for learning to fly a helicopter; see Figure 25.25 on page 1002 for an example of the acrobatics the resulting policy is capable of. Russell (1998) describes the task of **inverse reinforcement learning**—figuring out what the reward function must be from an example path through that state space. This is useful as a part of apprenticeship learning, or as a part of doing science—we can understand an animal or robot by working backwards from what it does to what its reward function must be.

INVERSE
REINFORCEMENT
LEARNING

This chapter has dealt only with atomic states—all the agent knows about a state is the set of available actions and the utilities of the resulting states (or of state-action pairs). But it is also possible to apply reinforcement learning to structured representations rather than atomic ones; this is called **relational reinforcement learning** (Tadepalli *et al.*, 2004).

RELATIONAL
REINFORCEMENT
LEARNING

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 POLICY-EVALUATION step may fail if $\gamma = 1$. Show that this problem cannot arise if POLICY-EVALUATION 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:

- 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.
- Experiment with various heuristics for ordering the priority queue, examining their effect on learning rates and computation time.

21.4 Write out the parameter update equations for TD learning with

$$\hat{U}(x, y) = \theta_0 + \theta_1 x + \theta_2 y + \theta_3 \sqrt{(x - x_g)^2 + (y - y_g)^2}.$$



21.5 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 (21.10). Compare their performance in three environments:

- The 4×3 world described in the chapter.
- A 10×10 world with no obstacles and a +1 reward at (10,10).
- A 10×10 world with no obstacles and a +1 reward at (5,5).

21.6 Devise suitable features for reinforcement learning in stochastic grid worlds (generalizations of the 4×3 world) that contain multiple obstacles and multiple terminal states with rewards of +1 or -1.



21.7 Extend the standard game-playing environment (Chapter 5) 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 (21.12)) 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.

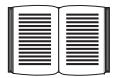
21.8 Compute the true utility function and the best linear approximation in x and y (as in Equation (21.10)) for the following environments:

- 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.



21.9 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.



21.10 Is reinforcement learning an appropriate abstract model for evolution? What connection exists, if any, between hardwired reward signals and evolutionary fitness?

22

NATURAL LANGUAGE PROCESSING

In which we see how to make use of the copious knowledge that is expressed in natural language.

Homo sapiens is set apart from other species by the capacity for language. Somewhere around 100,000 years ago, humans learned how to speak, and about 7,000 years ago learned to write. Although chimpanzees, dolphins, and other animals have shown vocabularies of hundreds of signs, only humans can reliably communicate an unbounded number of qualitatively different messages on any topic using discrete signs.

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 Alan Turing proposed his Test (see Section 1.1.1), he based it on language, not art or TV. There are two main reasons why we want our computer agents to be able to process natural languages: first, to communicate with humans, a topic we take up in Chapter 23, and second, to acquire information from written language, the focus of this chapter.

There are over a trillion pages of information on the Web, almost all of it in natural language. An agent that wants to do **knowledge acquisition** needs to understand (at least partially) the ambiguous, messy languages that humans use. We examine the problem from the point of view of specific information-seeking tasks: text classification, information retrieval, and information extraction. One common factor in addressing these tasks is the use of **language models**: models that predict the probability distribution of language expressions.

KNOWLEDGE
ACQUISITION

LANGUAGE MODEL

22.1 LANGUAGE MODELS

LANGUAGE

GRAMMAR
SEMANTICS

Formal languages, such as the programming languages Java or Python, have precisely defined language models. A **language** can be defined as a set of strings; “`print(2 + 2)`” is a legal program in the language Python, whereas “`2)+(2 print`” is not. Since there are an infinite number of legal programs, they cannot be enumerated; instead they are specified by a set of rules called a **grammar**. Formal languages also have rules that define the meaning or **semantics** of a program; for example, the rules say that the “meaning” of “`2 + 2`” is 4, and the meaning of “`1/0`” is that an error is signaled.

Natural languages, such as English or Spanish, cannot be characterized as a definitive set of sentences. Everyone agrees that “Not to be invited is sad” is a sentence of English, but people disagree on the grammaticality of “To be not invited is sad.” Therefore, it is more fruitful to define a natural language model as a probability distribution over sentences rather than a definitive set. That is, rather than asking if a string of *words* is or is not a member of the set defining the language, we instead ask for $P(S = \text{words})$ —what is the probability that a random sentence would be *words*.

AMBIGUITY

Natural languages are also **ambiguous**. “He saw her duck” can mean either that he saw a waterfowl belonging to her, or that he saw her move to evade something. Thus, again, we cannot speak of a single meaning for a sentence, but rather of a probability distribution over possible meanings.

Finally, natural languages are difficult to deal with because they are very large, and constantly changing. Thus, our language models are, at best, an approximation. We start with the simplest possible approximations and move up from there.

CHARACTERS

22.1.1 N-gram character models

N-GRAM MODEL

Ultimately, a written text is composed of **characters**—letters, digits, punctuation, and spaces in English (and more exotic characters in some other languages). Thus, one of the simplest language models is a probability distribution over sequences of characters. As in Chapter 15, we write $P(c_{1:N})$ for the probability of a sequence of N characters, c_1 through c_N . In one Web collection, $P(\text{"the"}) = 0.027$ and $P(\text{"zgq"}) = 0.000000002$. A sequence of written symbols of length n is called an n -gram (from the Greek root for writing or letters), with special case “unigram” for 1-gram, “bigram” for 2-gram, and “trigram” for 3-gram. A model of the probability distribution of n -letter sequences is thus called an **n -gram model**. (But be careful: we can have n -gram models over sequences of words, syllables, or other units; not just over characters.)

An n -gram model is defined as a **Markov chain** of order $n - 1$. Recall from page 568 that in a Markov chain the probability of character c_i depends only on the immediately preceding characters, not on any other characters. So in a trigram model (Markov chain of order 2) we have

$$P(c_i | c_{1:i-1}) = P(c_i | c_{i-2:i-1}).$$

CORPUS

We can define the probability of a sequence of characters $P(c_{1:N})$ under the trigram model by first factoring with the chain rule and then using the Markov assumption:

$$P(c_{1:N}) = \prod_{i=1}^N P(c_i | c_{i-2:i-1}) = \prod_{i=1}^N P(c_i | c_{i-2:i-1}).$$

For a trigram character model in a language with 100 characters, $\mathbf{P}(C_i | C_{i-2:i-1})$ has a million entries, and can be accurately estimated by counting character sequences in a body of text of 10 million characters or more. We call a body of text a **corpus** (plural *corpora*), from the Latin word for *body*.

What can we do with n -gram character models? One task for which they are well suited is **language identification**: given a text, determine what natural language it is written in. This is a relatively easy task; even with short texts such as “Hello, world” or “Wie geht es dir,” it is easy to identify the first as English and the second as German. Computer systems identify languages with greater than 99% accuracy; occasionally, closely related languages, such as Swedish and Norwegian, are confused.

One approach to language identification is to first build a trigram character model of each candidate language, $P(c_i | c_{i-2:i-1}, \ell)$, where the variable ℓ ranges over languages. For each ℓ the model is built by counting trigrams in a corpus of that language. (About 100,000 characters of each language are needed.) That gives us a model of $\mathbf{P}(\text{Text} | \text{Language})$, but we want to select the most probable language given the text, so we apply Bayes’ rule followed by the Markov assumption to get the most probable language:

$$\begin{aligned}\ell^* &= \underset{\ell}{\operatorname{argmax}} P(\ell | c_{1:N}) \\ &= \underset{\ell}{\operatorname{argmax}} P(\ell) P(c_{1:N} | \ell) \\ &= \underset{\ell}{\operatorname{argmax}} P(\ell) \prod_{i=1}^N P(c_i | c_{i-2:i-1}, \ell)\end{aligned}$$

The trigram model can be learned from a corpus, but what about the prior probability $P(\ell)$? We may have some estimate of these values; for example, if we are selecting a random Web page we know that English is the most likely language and that the probability of Macedonian will be less than 1%. The exact number we select for these priors is not critical because the trigram model usually selects one language that is several orders of magnitude more probable than any other.

Other tasks for character models include spelling correction, genre classification, and named-entity recognition. Genre classification means deciding if a text is a news story, a legal document, a scientific article, etc. While many features help make this classification, counts of punctuation and other character n -gram features go a long way (Kessler *et al.*, 1997). Named-entity recognition is the task of finding names of things in a document and deciding what class they belong to. For example, in the text “Mr. Sopersteen was prescribed aciphex,” we should recognize that “Mr. Sopersteen” is the name of a person and “aciphex” is the name of a drug. Character-level models are good for this task because they can associate the character sequence “ex.” (“ex” followed by a space) with a drug name and “steen.” with a person name, and thereby identify words that they have never seen before.

22.1.2 Smoothing n -gram models

The major complication of n -gram models is that the training corpus provides only an estimate of the true probability distribution. For common character sequences such as “th” any English corpus will give a good estimate: about 1.5% of all trigrams. On the other hand, “ht” is very uncommon—no dictionary words start with ht. It is likely that the sequence would have a count of zero in a training corpus of standard English. Does that mean we should assign $P(“th”) = 0$? If we did, then the text “The program issues an http request” would have

SMOOTHING

BACKOFF MODEL

LINEAR
INTERPOLATION
SMOOTHING

PERPLEXITY

an English probability of zero, which seems wrong. We have a problem in generalization: we want our language models to generalize well to texts they haven't seen yet. Just because we have never seen “`http`” before does not mean that our model should claim that it is impossible. Thus, we will adjust our language model so that sequences that have a count of zero in the training corpus will be assigned a small nonzero probability (and the other counts will be adjusted downward slightly so that the probability still sums to 1). The process of adjusting the probability of low-frequency counts is called **smoothing**.

The simplest type of smoothing was suggested by Pierre-Simon Laplace in the 18th century: he said that, in the lack of further information, if a random Boolean variable X has been false in all n observations so far then the estimate for $P(X = \text{true})$ should be $1/(n+2)$. That is, he assumes that with two more trials, one might be true and one false. Laplace smoothing (also called add-one smoothing) is a step in the right direction, but performs relatively poorly. A better approach is a **backoff model**, in which we start by estimating n -gram counts, but for any particular sequence that has a low (or zero) count, we back off to $(n-1)$ -grams. **Linear interpolation smoothing** is a backoff model that combines trigram, bigram, and unigram models by linear interpolation. It defines the probability estimate as

$$\hat{P}(c_i | c_{i-2:i-1}) = \lambda_3 P(c_i | c_{i-2:i-1}) + \lambda_2 P(c_i | c_{i-1}) + \lambda_1 P(c_i),$$

where $\lambda_3 + \lambda_2 + \lambda_1 = 1$. The parameter values λ_i can be fixed, or they can be trained with an expectation–maximization algorithm. It is also possible to have the values of λ_i depend on the counts: if we have a high count of trigrams, then we weigh them relatively more; if only a low count, then we put more weight on the bigram and unigram models. One camp of researchers has developed ever more sophisticated smoothing models, while the other camp suggests gathering a larger corpus so that even simple smoothing models work well. Both are getting at the same goal: reducing the variance in the language model.

One complication: note that the expression $P(c_i | c_{i-2:i-1})$ asks for $P(c_1 | c_{-1:0})$ when $i = 1$, but there are no characters before c_1 . We can introduce artificial characters, for example, defining c_0 to be a space character or a special “begin text” character. Or we can fall back on lower-order Markov models, in effect defining $c_{-1:0}$ to be the empty sequence and thus $P(c_1 | c_{-1:0}) = P(c_1)$.

22.1.3 Model evaluation

With so many possible n -gram models—unigram, bigram, trigram, interpolated smoothing with different values of λ , etc.—how do we know what model to choose? We can evaluate a model with cross-validation. Split the corpus into a training corpus and a validation corpus. Determine the parameters of the model from the training data. Then evaluate the model on the validation corpus.

The evaluation can be a task-specific metric, such as measuring accuracy on language identification. Alternatively we can have a task-independent model of language quality: calculate the probability assigned to the validation corpus by the model; the higher the probability the better. This metric is inconvenient because the probability of a large corpus will be a very small number, and floating-point underflow becomes an issue. A different way of describing the probability of a sequence is with a measure called **perplexity**, defined as

$$\text{Perplexity}(c_{1:N}) = P(c_{1:N})^{-\frac{1}{N}}.$$

Perplexity can be thought of as the reciprocal of probability, normalized by sequence length. It can also be thought of as the weighted average branching factor of a model. Suppose there are 100 characters in our language, and our model says they are all equally likely. Then for a sequence of any length, the perplexity will be 100. If some characters are more likely than others, and the model reflects that, then the model will have a perplexity less than 100.

22.1.4 *N*-gram word models

VOCABULARY

Now we turn to *n*-gram models over words rather than characters. All the same mechanism applies equally to word and character models. The main difference is that the **vocabulary**—the set of symbols that make up the corpus and the model—is larger. There are only about 100 characters in most languages, and sometimes we build character models that are even more restrictive, for example by treating “A” and “a” as the same symbol or by treating all punctuation as the same symbol. But with word models we have at least tens of thousands of symbols, and sometimes millions. The wide range is because it is not clear what constitutes a word. In English a sequence of letters surrounded by spaces is a word, but in some languages, like Chinese, words are not separated by spaces, and even in English many decisions must be made to have a clear policy on word boundaries: how many words are in “ne’er-do-well”? Or in “(Tel:1-800-960-5660x123)”?

OUT OF VOCABULARY

Word *n*-gram models need to deal with **out of vocabulary** words. With character models, we didn’t have to worry about someone inventing a new letter of the alphabet.¹ But with word models there is always the chance of a new word that was not seen in the training corpus, so we need to model that explicitly in our language model. This can be done by adding just one new word to the vocabulary: <UNK>, standing for the unknown word. We can estimate *n*-gram counts for <UNK> by this trick: go through the training corpus, and the first time any individual word appears it is previously unknown, so replace it with the symbol <UNK>. All subsequent appearances of the word remain unchanged. Then compute *n*-gram counts for the corpus as usual, treating <UNK> just like any other word. Then when an unknown word appears in a test set, we look up its probability under <UNK>. Sometimes multiple unknown-word symbols are used, for different classes. For example, any string of digits might be replaced with <NUM>, or any email address with <EMAIL>.

To get a feeling for what word models can do, we built unigram, bigram, and trigram models over the words in this book and then randomly sampled sequences of words from the models. The results are

Unigram: logical are as are confusion a may right tries agent goal the was . . .

Bigram: systems are very similar computational approach would be represented . . .

Trigram: planning and scheduling are integrated the success of naive bayes model is . . .

Even with this small sample, it should be clear that the unigram model is a poor approximation of either English or the content of an AI textbook, and that the bigram and trigram models are

¹ With the possible exception of the groundbreaking work of T. Geisel (1955).

much better. The models agree with this assessment: the perplexity was 891 for the unigram model, 142 for the bigram model and 91 for the trigram model.

With the basics of n -gram models—both character- and word-based—established, we can turn now to some language tasks.

22.2 TEXT CLASSIFICATION

TEXT
CLASSIFICATION

We now consider in depth the task of **text classification**, also known as **categorization**: given a text of some kind, decide which of a predefined set of classes it belongs to. Language identification and genre classification are examples of text classification, as is sentiment analysis (classifying a movie or product review as positive or negative) and **spam detection** (classifying an email message as spam or not-spam). Since “not-spam” is awkward, researchers have coined the term **ham** for not-spam. We can treat spam detection as a problem in supervised learning. A training set is readily available: the positive (spam) examples are in my spam folder, the negative (ham) examples are in my inbox. Here is an excerpt:

Spam: Wholesale Fashion Watches -57% today. Designer watches for cheap ...
 Spam: You can buy ViagraFr\$1.85 All Medications at unbeatable prices! ...
 Spam: WE CAN TREAT ANYTHING YOU SUFFER FROM JUST TRUST US ...
 Spam: Sta.rt earn*ing the salary yo,u d-eserve by o'btaining the prope,r crede'ntials!
 Ham: The practical significance of hypertree width in identifying more ...
 Ham: Abstract: We will motivate the problem of social identity clustering: ...
 Ham: Good to see you my friend. Hey Peter, It was good to hear from you. ...
 Ham: PDS implies convexity of the resulting optimization problem (Kernel Ridge ...)

From this excerpt we can start to get an idea of what might be good features to include in the supervised learning model. Word n -grams such as “for cheap” and “You can buy” seem to be indicators of spam (although they would have a nonzero probability in ham as well). Character-level features also seem important: spam is more likely to be all uppercase and to have punctuation embedded in words. Apparently the spammers thought that the word bigram “you deserve” would be too indicative of spam, and thus wrote “yo,u d-eserve” instead. A character model should detect this. We could either create a full character n -gram model of spam and ham, or we could handcraft features such as “number of punctuation marks embedded in words.”

Note that we have two complementary ways of talking about classification. In the language-modeling approach, we define one n -gram language model for $\mathbf{P}(\text{Message} \mid \text{spam})$ by training on the spam folder, and one model for $\mathbf{P}(\text{Message} \mid \text{ham})$ by training on the inbox. Then we can classify a new message with an application of Bayes’ rule:

$$\operatorname{argmax}_{c \in \{\text{spam}, \text{ham}\}} P(c \mid \text{message}) = \operatorname{argmax}_{c \in \{\text{spam}, \text{ham}\}} P(\text{message} \mid c) P(c).$$

where $P(c)$ is estimated just by counting the total number of spam and ham messages. This approach works well for spam detection, just as it did for language identification.

BAG OF WORDS

In the machine-learning approach we represent the message as a set of feature/value pairs and apply a classification algorithm h to the feature vector \mathbf{X} . We can make the language-modeling and machine-learning approaches compatible by thinking of the n -grams as features. This is easiest to see with a unigram model. The features are the words in the vocabulary: “a,” “aardvark,” . . . , and the values are the number of times each word appears in the message. That makes the feature vector large and sparse. If there are 100,000 words in the language model, then the feature vector has length 100,000, but for a short email message almost all the features will have count zero. This unigram representation has been called the **bag of words** model. You can think of the model as putting the words of the training corpus in a bag and then selecting words one at a time. The notion of order of the words is lost; a unigram model gives the same probability to any permutation of a text. Higher-order n -gram models maintain some local notion of word order.

With bigrams and trigrams the number of features is squared or cubed, and we can add in other, non- n -gram features: the time the message was sent, whether a URL or an image is part of the message, an ID number for the sender of the message, the sender’s number of previous spam and ham messages, and so on. The choice of features is the most important part of creating a good spam detector—more important than the choice of algorithm for processing the features. In part this is because there is a lot of training data, so if we can propose a feature, the data can accurately determine if it is good or not. It is necessary to constantly update features, because spam detection is an **adversarial task**; the spammers modify their spam in response to the spam detector’s changes.

FEATURE SELECTION

It can be expensive to run algorithms on a very large feature vector, so often a process of **feature selection** is used to keep only the features that best discriminate between spam and ham. For example, the bigram “of the” is frequent in English, and may be equally frequent in spam and ham, so there is no sense in counting it. Often the top hundred or so features do a good job of discriminating between classes.

Once we have chosen a set of features, we can apply any of the supervised learning techniques we have seen; popular ones for text categorization include k -nearest-neighbors, support vector machines, decision trees, naive Bayes, and logistic regression. All of these have been applied to spam detection, usually with accuracy in the 98%–99% range. With a carefully designed feature set, accuracy can exceed 99.9%.

DATA COMPRESSION

22.2.1 Classification by data compression

Another way to think about classification is as a problem in **data compression**. A lossless compression algorithm takes a sequence of symbols, detects repeated patterns in it, and writes a description of the sequence that is more compact than the original. For example, the text “0.142857142857142857” might be compressed to “0.[142857]*3.” Compression algorithms work by building dictionaries of subsequences of the text, and then referring to entries in the dictionary. The example here had only one dictionary entry, “142857.”

In effect, compression algorithms are creating a language model. The LZW algorithm in particular directly models a maximum-entropy probability distribution. To do classification by compression, we first lump together all the spam training messages and compress them as

a unit. We do the same for the ham. Then when given a new message to classify, we append it to the spam messages and compress the result. We also append it to the ham and compress that. Whichever class compresses better—adds the fewer number of additional bytes for the new message—is the predicted class. The idea is that a spam message will tend to share dictionary entries with other spam messages and thus will compress better when appended to a collection that already contains the spam dictionary.

Experiments with compression-based classification on some of the standard corpora for text classification—the 20-Newsgroups data set, the Reuters-10 Corpora, the Industry Sector corpora—indicate that whereas running off-the-shelf compression algorithms like gzip, RAR, and LZW can be quite slow, their accuracy is comparable to traditional classification algorithms. This is interesting in its own right, and also serves to point out that there is promise for algorithms that use character n -grams directly with no preprocessing of the text or feature selection: they seem to be capturing some real patterns.

22.3 INFORMATION RETRIEVAL

INFORMATION
RETRIEVAL

Information retrieval is the task of finding documents that are relevant to a user’s need for information. The best-known examples of information retrieval systems are search engines on the World Wide Web. A Web user can type a query such as [AI book]² 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

QUERY LANGUAGE

1. **A corpus of documents.** Each system must decide what it wants to treat as a document: a paragraph, a page, or a multipage text.
2. **Queries posed in a query language.** A 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 posed 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, rendered as a two-dimensional display.

RESULT SET

RELEVANT

PRESENTATION

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

² We denote a search query as [query]. Square brackets are used rather than quotation marks so that we can distinguish the query [“two words”] from [two words].

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 are unfamiliar to users who are not programmers or logicians. Users find it unintuitive that when they want to know about farming in the states of Kansas *and* Nebraska they need to issue the query [farming (Kansas OR Nebraska)]. 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.

22.3.1 IR scoring functions

BM25 SCORING
FUNCTION

Most IR systems have abandoned the Boolean model and use models based on the statistics of word counts. We describe the **BM25 scoring function**, which comes from the Okapi project of Stephen Robertson and Karen Sparck Jones at London's City College, and has been used in search engines such as the open-source Lucene project.

A scoring function takes a document and a query and returns a numeric score; the most relevant documents have the highest scores. In the BM25 function, the score is a linear weighted combination of scores for each of the words that make up the query. Three factors affect the weight of a query term: First, the frequency with which a query term appears in a document (also known as *TF* for term frequency). For the query [farming in Kansas], documents that mention “farming” frequently will have higher scores. Second, the inverse document frequency of the term, or *IDF*. The word “in” appears in almost every document, so it has a high document frequency, and thus a low inverse document frequency, and thus it is not as important to the query as “farming” or “Kansas.” Third, the length of the document. A million-word document will probably mention all the query words, but may not actually be about the query. A short document that mentions all the words is a much better candidate.

The BM25 function takes all three of these into account. We assume we have created an index of the N documents in the corpus so that we can look up $TF(q_i, d_j)$, the count of the number of times word q_i appears in document d_j . We also assume a table of document frequency counts, $DF(q_i)$, that gives the number of documents that contain the word q_i . Then, given a document d_j and a query consisting of the words $q_{1:N}$, we have

$$BM25(d_j, q_{1:N}) = \sum_{i=1}^N IDF(q_i) \cdot \frac{TF(q_i, d_j) \cdot (k + 1)}{TF(q_i, d_j) + k \cdot (1 - b + b \cdot \frac{|d_j|}{L})},$$

where $|d_j|$ is the length of document d_j in words, and L is the average document length in the corpus: $L = \sum_i |d_i|/N$. We have two parameters, k and b , that can be tuned by cross-validation; typical values are $k = 2.0$ and $b = 0.75$. $IDF(q_i)$ is the inverse document

frequency of word q_i , given by

$$IDF(q_i) = \log \frac{N - DF(q_i) + 0.5}{DF(q_i) + 0.5}.$$

Of course, it would be impractical to apply the BM25 scoring function to every document in the corpus. Instead, systems create an **index** ahead of time that lists, for each vocabulary word, the documents that contain the word. This is called the **hit list** for the word. Then when given a query, we intersect the hit lists of the query words and only score the documents in the intersection.

22.3.2 IR system evaluation

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 relevance judgments. Traditionally, there have been two measures used in the scoring: recall and precision. We 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
Relevant	30	20
Not relevant	10	40

PRECISION

Precision measures the proportion of documents in the result set that are actually relevant. In our example, the precision is $30/(30 + 10) = .75$. The false positive rate is $1 - .75 = .25$.

RECALL

Recall measures the proportion of all the relevant documents in the collection that are in the result set. In our example, recall is $30/(30 + 20) = .60$. The false negative rate is $1 - .60 = .40$. In a very large document collection, such as the World Wide Web, recall is difficult to compute, because there is no easy way to examine every page on the Web for relevance. All we can do is either estimate recall by sampling or ignore recall completely and just judge precision. In the case of a Web search engine, there may be thousands of documents in the result set, so it makes more sense to measure precision for several different sizes, such as “P@10” (precision in the top 10 results) or “P@50,” rather than to estimate precision in the entire result set.

It is possible to trade off precision against recall by varying the size of the result set returned. In the extreme, a system that returns every document in the document collection 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. A summary of both measures is the F_1 score, a single number that is the harmonic mean of precision and recall, $2PR/(P + R)$.

22.3.3 IR refinements

There are many possible refinements to the system described here, and indeed Web search engines are continually updating their algorithms as they discover new approaches and as the Web grows and changes.

INDEX

HIT LIST

One common refinement is a better model of the effect of document length on relevance. Singhal *et al.* (1996) observed that simple document length normalization schemes tend to favor short documents too much and long documents not enough. They propose a *pivoted* document length normalization scheme; the idea is that the pivot is the document length at which the old-style normalization is correct; documents shorter than that get a boost and longer ones get a penalty.

The BM25 scoring function uses a word model that 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 some use a **stemming** algorithm to reduce “couches” to the stem form “couch,” both in the query and the documents. 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 algorithms based on dictionaries (don’t remove “-ing” if the word is already listed in the dictionary) can. While stemming has a small effect in English, it is more important in other languages. In German, for example, it is not uncommon to see words like “Lebensversicherungsgesellschaftsangestellter” (life insurance company employee). Languages such as Finnish, Turkish, Inuit, and Yupik have recursive morphological rules that in principle generate words of unbounded length.

CASE FOLDING
STEMMING

SYNONYM

METADATA
LINKS

PAGERANK

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 can hurt precision. A user who gives the query [Tim Couch] wants to see results about the football player, not sofas. The problem is that “languages abhor absolute synonyms 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 evolve the meanings to remove the confusion. Related words that are not synonyms also play an important role in ranking—terms like “leather”, “wooden,” or “modern” can serve to confirm that the document really is about “couch.” Synonyms and related words can be found in dictionaries or by looking for correlations in documents or in queries—if we find that many users who ask the query [new sofa] follow it up with the query [new couch], we can in the future alter [new sofa] to be [new sofa OR new couch].

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 publication data. On the Web, hypertext **links** between documents are a crucial source of information.

22.3.4 The PageRank algorithm

PageRank³ was one of the two original ideas that set Google’s search apart from other Web search engines when it was introduced in 1997. (The other innovation was the use of anchor

³ The name stands both for Web pages and for coinventor Larry Page (Brin and Page, 1998).

```

function HITS(query) returns pages with hub and authority numbers
  pages  $\leftarrow$  EXPAND-PAGES(RELEVANT-PAGES(query))
  for each p in pages do
    p.AUTHORITY  $\leftarrow$  1
    p.HUB  $\leftarrow$  1
  repeat until convergence do
    for each p in pages do
      p.AUTHORITY  $\leftarrow \sum_i \text{INLINK}_i(p).\text{HUB}
      p.HUB  $\leftarrow \sum_i \text{OUTLINK}_i(p).\text{AUTHORITY}
    NORMALIZE(pages)
  return pages$$ 
```

Figure 22.1 The HITS algorithm for computing hubs and authorities with respect to a query. RELEVANT-PAGES fetches the pages that match the query, and EXPAND-PAGES adds in every page that links to or is linked from one of the relevant pages. NORMALIZE divides each page’s score by the sum of the squares of all pages’ scores (separately for both the authority and hubs scores).

text—the underlined text in a hyperlink—to index a page, even though the anchor text was on a *different* page than the one being indexed.) PageRank was invented to solve the problem of the tyranny of *TF* scores: if the query is [IBM], how do we make sure that IBM’s home page, `ibm.com`, is the first result, even if another page mentions the term “IBM” more frequently? The idea is that `ibm.com` has many in-links (links to the page), so it should be ranked higher: each in-link is a vote for the quality of the linked-to page. But if we only counted in-links, then it would be possible for a Web spammer to create a network of pages and have them all point to a page of his choosing, increasing the score of that page. Therefore, the PageRank algorithm is designed to weight links from high-quality sites more heavily. What is a high-quality site? One that is linked to by other high-quality sites. The definition is recursive, but we will see that the recursion bottoms out properly. The PageRank for a page *p* is defined as:

$$PR(p) = \frac{1-d}{N} + d \sum_i \frac{PR(in_i)}{C(in_i)},$$

where $PR(p)$ is the PageRank of page *p*, N is the total number of pages in the corpus, in_i are the pages that link in to *p*, and $C(in_i)$ is the count of the total number of out-links on page in_i . The constant d is a damping factor. It can be understood through the **random surfer model**: imagine a Web surfer who starts at some random page and begins exploring. With probability d (we’ll assume $d = 0.85$) the surfer clicks on one of the links on the page (choosing uniformly among them), and with probability $1 - d$ she gets bored with the page and restarts on a random page anywhere on the Web. The PageRank of page *p* is then the probability that the random surfer will be at page *p* at any point in time. PageRank can be computed by an iterative procedure: start with all pages having $PR(p) = 1$, and iterate the algorithm, updating ranks until they converge.

22.3.5 The HITS algorithm

The Hyperlink-Induced Topic Search algorithm, also known as “Hubs and Authorities” or HITS, is another influential link-analysis algorithm (see Figure 22.1). HITS differs from PageRank in several ways. First, it is a query-dependent measure: it rates pages with respect to a query. That means that it must be computed anew for each query—a computational burden that most search engines have elected not to take on. Given a query, HITS first finds a set of pages that are relevant to the query. It does that by intersecting hit lists of query words, and then adding pages in the link neighborhood of these pages—pages that link to or are linked from one of the pages in the original relevant set.

AUTHORITY

HUB

Each page in this set is considered an **authority** on the query to the degree that other pages in the relevant set point to it. A page is considered a **hub** to the degree that it points to other authoritative pages in the relevant set. Just as with PageRank, we don’t want to merely count the number of links; we want to give more value to the high-quality hubs and authorities. Thus, as with PageRank, we iterate a process that updates the authority score of a page to be the sum of the hub scores of the pages that point to it, and the hub score to be the sum of the authority scores of the pages it points to. If we then normalize the scores and repeat k times, the process will converge.

Both PageRank and HITS played important roles in developing our understanding of Web information retrieval. These algorithms and their extensions are used in ranking billions of queries daily as search engines steadily develop better ways of extracting yet finer signals of search relevance.

QUESTION
ANSWERING

22.3.6 Question answering

Information retrieval is the task of finding documents that are relevant to a query, where the query may be a question, or just a topic area or concept. **Question answering** is a somewhat different task, in which the query really is a question, and the answer is not a ranked list of documents but rather a short response—a sentence, or even just a phrase. There have been question-answering NLP (natural language processing) systems since the 1960s, but only since 2001 have such systems used Web information retrieval to radically increase their breadth of coverage.

The ASKMSR system (Banko *et al.*, 2002) is a typical Web-based question-answering system. It is based on the intuition that most questions will be answered many times on the Web, so question answering should be thought of as a problem in precision, not recall. We don’t have to deal with all the different ways that an answer might be phrased—we only have to find one of them. For example, consider the query [Who killed Abraham Lincoln?] Suppose a system had to answer that question with access only to a single encyclopedia, whose entry on Lincoln said

John Wilkes Booth altered history with a bullet. He will forever be known as the man who ended Abraham Lincoln’s life.

To use this passage to answer the question, the system would have to know that ending a life can be a killing, that “He” refers to Booth, and several other linguistic and semantic facts.

ASKMSR does not attempt this kind of sophistication—it knows nothing about pronoun reference, or about killing, or any other verb. It does know 15 different kinds of questions, and how they can be rewritten as queries to a search engine. It knows that [Who killed Abraham Lincoln] can be rewritten as the query [* killed Abraham Lincoln] and as [Abraham Lincoln was killed by *]. It issues these rewritten queries and examines the results that come back—not the full Web pages, just the short summaries of text that appear near the query terms. The results are broken into 1-, 2-, and 3-grams and tallied for frequency in the result sets and for weight: an n -gram that came back from a very specific query rewrite (such as the exact phrase match query ["Abraham Lincoln was killed by *"]) would get more weight than one from a general query rewrite, such as [Abraham OR Lincoln OR killed]. We would expect that "John Wilkes Booth" would be among the highly ranked n -grams retrieved, but so would "Abraham Lincoln" and "the assassination of" and "Ford's Theatre."

Once the n -grams are scored, they are filtered by expected type. If the original query starts with "who," then we filter on names of people; for "how many" we filter on numbers, for "when," on a date or time. There is also a filter that says the answer should not be part of the question; together these should allow us to return "John Wilkes Booth" (and not "Abraham Lincoln") as the highest-scoring response.

In some cases the answer will be longer than three words; since the components responses only go up to 3-grams, a longer response would have to be pieced together from shorter pieces. For example, in a system that used only bigrams, the answer "John Wilkes Booth" could be pieced together from high-scoring pieces "John Wilkes" and "Wilkes Booth."

At the Text Retrieval Evaluation Conference (TREC), ASKMSR was rated as one of the top systems, beating out competitors with the ability to do far more complex language understanding. ASKMSR relies upon the breadth of the content on the Web rather than on its own depth of understanding. It won't be able to handle complex inference patterns like associating "who killed" with "ended the life of." But it knows that the Web is so vast that it can afford to ignore passages like that and wait for a simple passage it can handle.

22.4 INFORMATION EXTRACTION

INFORMATION EXTRACTION

Information extraction is the process of acquiring knowledge by skimming a text and looking for occurrences of a particular class of object and for relationships among objects. A typical task is 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. In a limited domain, this can be done with high accuracy. As the domain gets more general, more complex linguistic models and more complex learning techniques are necessary. We will see in Chapter 23 how to define complex language models of the phrase structure (noun phrases and verb phrases) of English. But so far there are no complete models of this kind, so for the limited needs of information extraction, we define limited models that approximate the full English model, and concentrate on just the parts that are needed for the task at hand. The models we describe in this sec-

tion are approximations in the same way that the simple 1-CNF logical model in Figure 7.21 (page 271) is an approximations of the full, wiggly, logical model.

In this section we describe six different approaches to information extraction, in order of increasing complexity on several dimensions: deterministic to stochastic, domain-specific to general, hand-crafted to learned, and small-scale to large-scale.

22.4.1 Finite-state automata for information extraction

ATTRIBUTE-BASED EXTRACTION

The simplest type of information extraction system is an **attribute-based extraction** system that 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 12.7 the problem of extracting from the text “IBM ThinkBook 970. Our price: \$399.00” the set of attributes {Manufacturer=IBM, Model=ThinkBook970, Price=\$399.00}. We can address this problem by defining a **template** (also known as a pattern) for each attribute we would like to extract. The template is defined by a finite state automaton, the simplest example of which is the **regular expression**, or regex. 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 template for prices in dollars:

[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 ...

Templates are often defined with three parts: a prefix regex, a target regex, and a postfix regex. For prices, the target regex is as above, the prefix would look for strings such as “price:” and the postfix could be empty. The idea is that some clues about an attribute come from the attribute value itself and some come from the surrounding text.

If a regular expression for an attribute 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, all we can do is give a default value or leave the attribute missing; but if there are several matches, we need a process to choose among them. One strategy is to have several templates for each attribute, ordered by priority. So, for example, the top-priority template for price might look for the prefix “our price:”; if that is not found, we look for the prefix “price:” and if that is not found, the empty prefix. Another strategy is to take all the matches and find some way to choose among them. For example, we could take the lowest price that is within 50% of the highest price. That will select \$78.00 as the target from the text “List price \$99.00, special sale price \$78.00, shipping \$3.00.”

RELATIONAL EXTRACTION

One step up from attribute-based extraction systems are **relational extraction** systems, which deal with multiple objects and the relations among 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 extract the relations:

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

CASCADED
FINITE-STATE
TRANSDUCERS

A relational extraction system can be built as a series of **cascaded finite-state transducers**. That is, the system consists of a series of small, efficient 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. FASTUS consists of 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. 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 "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")`

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. We will see how to write a complex description of noun and verb phrases in Chapter 23, but here we have simple rules that only approximate the complexity of English, but have the advantage of being representable by finite state automata. The example sentence would emerge from this stage as the following sequence of tagged groups:

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 references to a joint venture, and that they should be merged into one. This is an instance of the **identity uncertainty problem** discussed in Section 14.6.3.

In general, finite-state template-based 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. The cascaded transducer model helps modularize the necessary knowledge, easing construction of the system. These systems work especially well when they are reverse-engineering text that has been generated by a program. For example, a shopping site on the Web is generated by a program that takes database entries and formats them into Web pages; a template-based extractor then recovers the original database. Finite-state information extraction is less successful at recovering information in highly variable format, such as text written by humans on a variety of subjects.

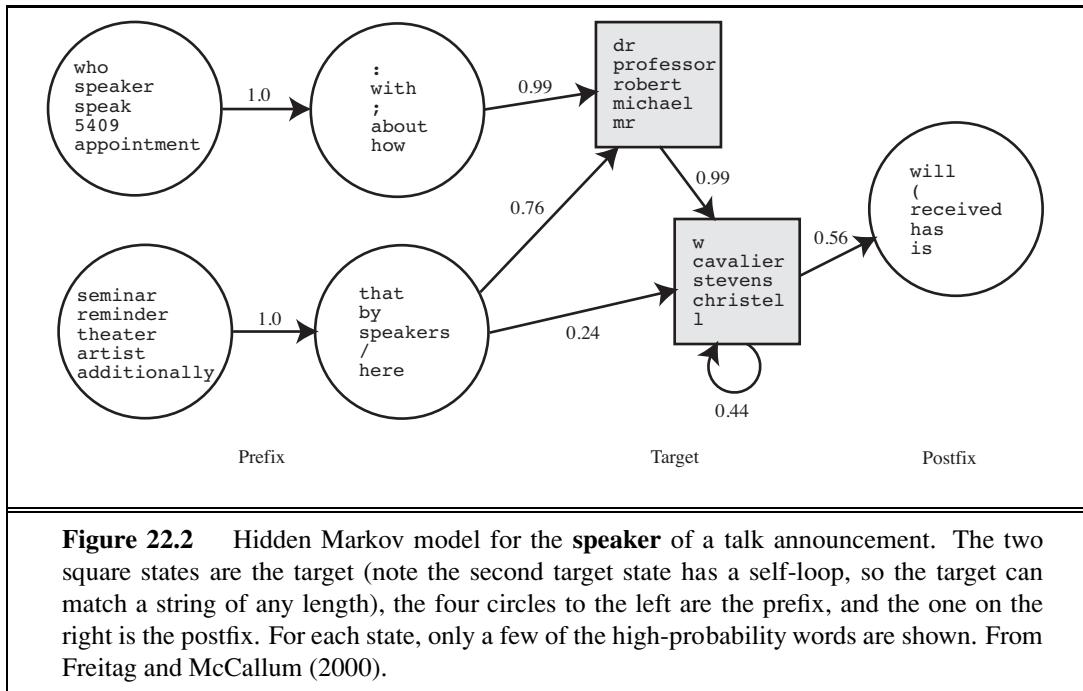
22.4.2 Probabilistic models for information extraction

When information extraction must be attempted from noisy or varied input, simple finite-state approaches fare poorly. It is too hard to get all the rules and their priorities right; it is better to use a probabilistic model rather than a rule-based model. The simplest probabilistic model for sequences with hidden state is the hidden Markov model, or HMM.

Recall from Section 15.3 that an HMM models a progression through a sequence of hidden states, x_t , with an observation e_t at each step. To apply HMMs to information extraction, we can either build one big HMM for all the attributes or build a separate HMM for each attribute. We’ll do the second. The observations are the words of the text, and the hidden states are whether we are in the target, prefix, or postfix part of the attribute template, or in the background (not part of a template). For example, here is a brief text and the most probable (Viterbi) path for that text for two HMMs, one trained to recognize the speaker in a talk announcement, and one trained to recognize dates. The “-” indicates a background state:

Text:	There	will	be	a	seminar	by	Dr.	Andrew	McCallum	on	Friday
Speaker:	-	-	-	-	PRE	PRE	TARGET	TARGET	TARGET	POST	-
Date:	-	-	-	-	-	-	-	-	-	PRE	TARGET

HMMs have two big advantages over FSAs for extraction. First, HMMs are probabilistic, and thus tolerant to noise. In a regular expression, if a single expected character is missing, the regex fails to match; with HMMs there is graceful degradation with missing characters/words, and we get a probability indicating the degree of match, not just a Boolean match/fail. Second,



HMMs can be trained from data; they don't require laborious engineering of templates, and thus they can more easily be kept up to date as text changes over time.

Note that we have assumed a certain level of structure in our HMM templates: they all consist of one or more target states, and any prefix states must precede the targets, postfix states most follow the targets, and other states must be background. This structure makes it easier to learn HMMs from examples. With a partially specified structure, the forward-backward algorithm can be used to learn both the transition probabilities $\mathbf{P}(\mathbf{X}_t | \mathbf{X}_{t-1})$ between states and the observation model, $\mathbf{P}(\mathbf{E}_t | \mathbf{X}_t)$, which says how likely each word is in each state. For example, the word "Friday" would have high probability in one or more of the target states of the date HMM, and lower probability elsewhere.

With sufficient training data, the HMM automatically learns a structure of dates that we find intuitive: the date HMM might have one target state in which the high-probability words are "Monday," "Tuesday," etc., and which has a high-probability transition to a target state with words "Jan", "January", "Feb", etc. Figure 22.2 shows the HMM for the speaker of a talk announcement, as learned from data. The prefix covers expressions such as "Speaker:" and "seminar by," and the target has one state that covers titles and first names and another state that covers initials and last names.

Once the HMMs have been learned, we can apply them to a text, using the Viterbi algorithm to find the most likely path through the HMM states. One approach is to apply each attribute HMM separately; in this case you would expect most of the HMMs to spend most of their time in background states. This is appropriate when the extraction is sparse—when the number of extracted words is small compared to the length of the text.

The other approach is to combine all the individual attributes into one big HMM, which would then find a path that wanders through different target attributes, first finding a speaker target, then a date target, etc. Separate HMMs are better when we expect just one of each attribute in a text and one big HMM is better when the texts are more free-form and dense with attributes. With either approach, in the end we have a collection of target attribute observations, and have to decide what to do with them. If every expected attribute has one target filler then the decision is easy: we have an instance of the desired relation. If there are multiple fillers, we need to decide which to choose, as we discussed with template-based systems. HMMs have the advantage of supplying probability numbers that can help make the choice. If some targets are missing, we need to decide if this is an instance of the desired relation at all, or if the targets found are false positives. A machine learning algorithm can be trained to make this choice.

22.4.3 Conditional random fields for information extraction

One issue with HMMs for the information extraction task is that they model a lot of probabilities that we don't really need. An HMM is a generative model; it models the full joint probability of observations and hidden states, and thus can be used to generate samples. That is, we can use the HMM model not only to parse a text and recover the speaker and date, but also to generate a random instance of a text containing a speaker and a date. Since we're not interested in that task, it is natural to ask whether we might be better off with a model that doesn't bother modeling that possibility. All we need in order to understand a text is a **discriminative model**, one that models the conditional probability of the hidden attributes given the observations (the text). Given a text $\mathbf{e}_{1:N}$, the conditional model finds the hidden state sequence $\mathbf{X}_{1:N}$ that maximizes $P(\mathbf{X}_{1:N} | \mathbf{e}_{1:N})$.

Modeling this directly gives us some freedom. We don't need the independence assumptions of the Markov model—we can have an \mathbf{x}_t that is dependent on \mathbf{x}_1 . A framework for this type of model is the **conditional random field**, or CRF, which models a conditional probability distribution of a set of target variables given a set of observed variables. Like Bayesian networks, CRFs can represent many different structures of dependencies among the variables. One common structure is the **linear-chain conditional random field** for representing Markov dependencies among variables in a temporal sequence. Thus, HMMs are the temporal version of naive Bayes models, and linear-chain CRFs are the temporal version of logistic regression, where the predicted target is an entire state sequence rather than a single binary variable.

Let $\mathbf{e}_{1:N}$ be the observations (e.g., words in a document), and $\mathbf{x}_{1:N}$ be the sequence of hidden states (e.g., the prefix, target, and postfix states). A linear-chain conditional random field defines a conditional probability distribution:

$$\mathbf{P}(\mathbf{x}_{1:N} | \mathbf{e}_{1:N}) = \alpha e^{[\sum_{i=1}^N F(\mathbf{x}_{i-1}, \mathbf{x}_i, \mathbf{e}, i)]},$$

where α is a normalization factor (to make sure the probabilities sum to 1), and F is a feature function defined as the weighted sum of a collection of k component feature functions:

$$F(\mathbf{x}_{i-1}, \mathbf{x}_i, \mathbf{e}, i) = \sum_k \lambda_k f_k(\mathbf{x}_{i-1}, \mathbf{x}_i, \mathbf{e}, i).$$

CONDITIONAL
RANDOM FIELD

LINEAR-CHAIN
CONDITIONAL
RANDOM FIELD

The λ_k parameter values are learned with a MAP (maximum a posteriori) estimation procedure that maximizes the conditional likelihood of the training data. The feature functions are the key components of a CRF. The function f_k has access to a pair of adjacent states, \mathbf{x}_{i-1} and \mathbf{x}_i , but also the entire observation (word) sequence \mathbf{e} , and the current position in the temporal sequence, i . This gives us a lot of flexibility in defining features. We can define a simple feature function, for example one that produces a value of 1 if the current word is ANDREW and the current state is SPEAKER:

$$f_1(\mathbf{x}_{i-1}, \mathbf{x}_i, \mathbf{e}, i) = \begin{cases} 1 & \text{if } \mathbf{x}_i = \text{SPEAKER} \text{ and } \mathbf{e}_i = \text{ANDREW} \\ 0 & \text{otherwise} \end{cases}$$

How are features like these used? It depends on their corresponding weights. If $\lambda_1 > 0$, then whenever f_1 is true, it increases the probability of the hidden state sequence $\mathbf{x}_{1:N}$. This is another way of saying “the CRF model should prefer the target state SPEAKER for the word ANDREW.” If on the other hand $\lambda_1 < 0$, the CRF model will try to avoid this association, and if $\lambda_1 = 0$, this feature is ignored. Parameter values can be set manually or can be learned from data. Now consider a second feature function:

$$f_2(\mathbf{x}_{i-1}, \mathbf{x}_i, \mathbf{e}, i) = \begin{cases} 1 & \text{if } \mathbf{x}_i = \text{SPEAKER} \text{ and } \mathbf{e}_{i+1} = \text{SAID} \\ 0 & \text{otherwise} \end{cases}$$

This feature is true if the current state is SPEAKER and the next word is “said.” One would therefore expect a positive λ_2 value to go with the feature. More interestingly, note that both f_1 and f_2 can hold at the same time for a sentence like “Andrew said . . .” In this case, the two features overlap each other and both boost the belief in $\mathbf{x}_1 = \text{SPEAKER}$. Because of the independence assumption, HMMs cannot use overlapping features; CRFs can. Furthermore, a feature in a CRF can use any part of the sequence $\mathbf{e}_{1:N}$. Features can also be defined over transitions between states. The features we defined here were binary, but in general, a feature function can be any real-valued function. For domains where we have some knowledge about the types of features we would like to include, the CRF formalism gives us a great deal of flexibility in defining them. This flexibility can lead to accuracies that are higher than with less flexible models such as HMMs.

22.4.4 Ontology extraction from large corpora

So far we have thought of information extraction as finding a specific set of relations (e.g., speaker, time, location) in a specific text (e.g., a talk announcement). A different application of extraction technology is building a large knowledge base or ontology of facts from a corpus. This is different in three ways: First it is open-ended—we want to acquire facts about all types of domains, not just one specific domain. Second, with a large corpus, this task is dominated by precision, not recall—just as with question answering on the Web (Section 22.3.6). Third, the results can be statistical aggregates gathered from multiple sources, rather than being extracted from one specific text.

For example, Hearst (1992) looked at the problem of learning an ontology of concept categories and subcategories from a large corpus. (In 1992, a large corpus was a 1000-page encyclopedia; today it would be a 100-million-page Web corpus.) The work concentrated on templates that are very general (not tied to a specific domain) and have high precision (are

almost always correct when they match) but low recall (do not always match). Here is one of the most productive templates:

$$NP \text{ such as } NP \ (, NP)^* (,) ? ((\text{and} \mid \text{or}) \ NP) ? \ .$$

Here the bold words and commas must appear literally in the text, but the parentheses are for grouping, the asterisk means *repetition of zero or more*, and the question mark means *optional*. NP is a variable standing for a noun phrase; Chapter 23 describes how to identify noun phrases; for now just assume that we know some words are nouns and other words (such as verbs) that we can reliably assume are not part of a simple noun phrase. This template matches the texts “diseases such as rabies affect your dog” and “supports network protocols such as DNS,” concluding that rabies is a disease and DNS is a network protocol. Similar templates can be constructed with the key words “including,” “especially,” and “or other.” Of course these templates will fail to match many relevant passages, like “Rabies is a disease.” That is intentional. The “ NP is a NP ” template does indeed sometimes denote a subcategory relation, but it often means something else, as in “There is a God” or “She is a little tired.” With a large corpus we can afford to be picky; to use only the high-precision templates. We’ll miss many statements of a subcategory relationship, but most likely we’ll find a paraphrase of the statement somewhere else in the corpus in a form we can use.

22.4.5 Automated template construction

The *subcategory* relation is so fundamental that is worthwhile to handcraft a few templates to help identify instances of it occurring in natural language text. But what about the thousands of other relations in the world? There aren’t enough AI grad students in the world to create and debug templates for all of them. Fortunately, it is possible to *learn* templates from a few examples, then use the templates to learn more examples, from which more templates can be learned, and so on. In one of the first experiments of this kind, Brin (1999) started with a data set of just five examples:

- (“Isaac Asimov”, “The Robots of Dawn”)
- (“David Brin”, “Startide Rising”)
- (“James Gleick”, “Chaos—Making a New Science”)
- (“Charles Dickens”, “Great Expectations”)
- (“William Shakespeare”, “The Comedy of Errors”)

Clearly these are examples of the author–title relation, but the learning system had no knowledge of authors or titles. The words in these examples were used in a search over a Web corpus, resulting in 199 matches. Each match is defined as a tuple of seven strings,

$$(Author, Title, Order, Prefix, Middle, Postfix, URL) \ ,$$

where *Order* is true if the author came first and false if the title came first, *Middle* is the characters between the author and title, *Prefix* is the 10 characters before the match, *Suffix* is the 10 characters after the match, and *URL* is the Web address where the match was made.

Given a set of matches, a simple template-generation scheme can find templates to explain the matches. The language of templates was designed to have a close mapping to the matches themselves, to be amenable to automated learning, and to emphasize high precision

(possibly at the risk of lower recall). Each template has the same seven components as a match. The *Author* and *Title* are regexes consisting of any characters (but beginning and ending in letters) and constrained to have a length from half the minimum length of the examples to twice the maximum length. The prefix, middle, and postfix are restricted to literal strings, not regexes. The middle is the easiest to learn: each distinct middle string in the set of matches is a distinct candidate template. For each such candidate, the template's *Prefix* is then defined as the longest common suffix of all the prefixes in the matches, and the *Postfix* is defined as the longest common prefix of all the postfixes in the matches. If either of these is of length zero, then the template is rejected. The *URL* of the template is defined as the longest prefix of the URLs in the matches.

In the experiment run by Brin, the first 199 matches generated three templates. The most productive template was

```
<LI><B> Title </B> by Author (URL: www.sff.net/locus/c
```

The three templates were then used to retrieve 4047 more (author, title) examples. The examples were then used to generate more templates, and so on, eventually yielding over 15,000 titles. Given a good set of templates, the system can collect a good set of examples. Given a good set of examples, the system can build a good set of templates.

The biggest weakness in this approach is the sensitivity to noise. If one of the first few templates is incorrect, errors can propagate quickly. One way to limit this problem is to not accept a new example unless it is verified by multiple templates, and not accept a new template unless it discovers multiple examples that are also found by other templates.

22.4.6 Machine reading

Automated template construction is a big step up from handcrafted template construction, but it still requires a handful of labeled examples of each relation to get started. To build a large ontology with many thousands of relations, even that amount of work would be onerous; we would like to have an extraction system with *no* human input of any kind—a system that could read on its own and build up its own database. Such a system would be relation-independent; would work for any relation. In practice, these systems work on *all* relations in parallel, because of the I/O demands of large corpora. They behave less like a traditional information-extraction system that is targeted at a few relations and more like a human reader who learns from the text itself; because of this the field has been called **machine reading**.

A representative machine-reading system is TEXTRUNNER (Banko and Etzioni, 2008). TEXTRUNNER uses cotraining to boost its performance, but it needs something to bootstrap from. In the case of Hearst (1992), specific patterns (e.g., *such as*) provided the bootstrap, and for Brin (1998), it was a set of five author–title pairs. For TEXTRUNNER, the original inspiration was a taxonomy of eight very general syntactic templates, as shown in Figure 22.3. It was felt that a small number of templates like this could cover most of the ways that relationships are expressed in English. The actual bootstrapping starts from a set of labelled examples that are extracted from the Penn Treebank, a corpus of parsed sentences. For example, from the parse of the sentence “Einstein received the Nobel Prize in 1921,” TEXTRUNNER is able

to extract the relation (“Einstein,” “received,” “Nobel Prize”).

Given a set of labeled examples of this type, TEXTRUNNER trains a linear-chain CRF to extract further examples from unlabeled text. The features in the CRF include function words like “to” and “of” and “the,” but not nouns and verbs (and not noun phrases or verb phrases). Because TEXTRUNNER is domain-independent, it cannot rely on predefined lists of nouns and verbs.

Type	Template	Example	Frequency
Verb	$NP_1 \ Verb \ NP_2$	X established Y	38%
Noun–Prep	$NP_1 \ NP \ Prep \ NP_2$	X settlement with Y	23%
Verb–Prep	$NP_1 \ Verb \ Prep \ NP_2$	X moved to Y	16%
Infinitive	$NP_1 \ to \ Verb \ NP_2$	X plans to acquire Y	9%
Modifier	$NP_1 \ Verb \ NP_2 \ Noun$	X is Y winner	5%
Noun–Coordinate	$NP_1 (, \ and - :) \ NP_2 \ NP$	X-Y deal	2%
Verb–Coordinate	$NP_1 (, \ and) \ NP_2 \ Verb$	X, Y merge	1%
Appositive	$NP_1 \ NP \ (: ,)? \ NP_2$	X hometown : Y	1%

Figure 22.3 Eight general templates that cover about 95% of the ways that relations are expressed in English.

TEXTRUNNER achieves a precision of 88% and recall of 45% (F_1 of 60%) on a large Web corpus. TEXTRUNNER has extracted hundreds of millions of facts from a corpus of a half-billion Web pages. For example, even though it has no predefined medical knowledge, it has extracted over 2000 answers to the query [what kills bacteria]; correct answers include antibiotics, ozone, chlorine, Cipro, and broccoli sprouts. Questionable answers include “water,” which came from the sentence “Boiling water for at least 10 minutes will kill bacteria.” It would be better to attribute this to “boiling water” rather than just “water.”

With the techniques outlined in this chapter and continual new inventions, we are starting to get closer to the goal of machine reading.

22.5 SUMMARY

The main points of this chapter are as follows:

- Probabilistic language models based on n -grams recover a surprising amount of information about a language. They can perform well on such diverse tasks as language identification, spelling correction, genre classification, and named-entity recognition.
- These language models can have millions of features, so feature selection and preprocessing of the data to reduce noise is important.
- **Text classification** can be done with naive Bayes n -gram models or with any of the classification algorithms we have previously discussed. Classification can also be seen as a problem in data compression.

- **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. On Web corpora, link-analysis algorithms improve performance.
- **Question answering** can be handled by an approach based on information retrieval, for questions that have multiple answers in the corpus. When more answers are available in the corpus, we can use techniques that emphasize precision rather than recall.
- **Information-extraction** systems use a more complex model that includes limited notions of syntax and semantics in the form of templates. They can be built from finite-state automata, HMMs, or conditional random fields, and can be learned from examples.
- 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 probabilistic models *do* provide insight into some *other* basic problems—problems that context-free models ignore. Chomsky’s remarks had the unfortunate effect of scaring many people away from statistical models for two decades, until these models reemerged for use in speech recognition (Jelinek, 1976).

Kessler *et al.* (1997) show how to apply character n -gram models to genre classification, and Klein *et al.* (2003) describe named-entity recognition with character models. Franz and Brants (2006) describe the Google n -gram corpus of 13 million unique words from a trillion words of Web text; it is now publicly available. The **bag of words** model gets its name from a passage from linguist Zellig Harris (1954), “language is not merely a bag of words but a tool with particular properties.” Norvig (2009) gives some examples of tasks that can be accomplished with n -gram models.

Add-one smoothing, first suggested by Pierre-Simon Laplace (1816), was formalized by Jeffreys (1948), and interpolation smoothing is due to Jelinek and Mercer (1980), who used it for speech recognition. Other techniques include Witten–Bell smoothing (1991), Good–Turing smoothing (Church and Gale, 1991) and Kneser–Ney smoothing (1995). Chen and Goodman (1996) and Goodman (2001) survey smoothing techniques.

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).

Manning and Schütze (1999) and Sebastiani (2002) survey text-classification techniques. 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% with a naive Bayes classifier, and up to 98.6% with 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. Schapire and Singer (2000) show that simple linear classifiers can often achieve accuracy almost as good as more complex models and are more efficient to evaluate. Nigam *et al.* (2000) show how to use the EM algorithm to label unlabeled documents, thus learning a better classification model. Witten *et al.* (1999) describe compression algorithms for classification, and show the deep connection between the LZW compression algorithm and maximum-entropy language models.

Many of the n -gram model techniques are also used in bioinformatics problems. Bio-statistics and probabilistic NLP are coming closer together, as each deals with long, structured sequences chosen from an alphabet of constituents.

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. Croft *et al.* (2009) and Manning *et al.* (2008) are the first textbooks to cover Web-based search as well as traditional IR. Hearst (2009) covers user interfaces for Web search. 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, one due to Ponte and Croft (1998) and one by Maron and Kuhns (1960) and Robertson and Sparck Jones (1976). 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. Craswell *et al.* (2005) describe the BM25 scoring function and Svore and Burges (2009) describe how BM25 can be improved with a machine learning approach that incorporates click data—examples of past search queries and the results that were clicked on.

Brin and Page (1998) describe the PageRank algorithm and the implementation of a Web search engine. Kleinberg (1999) describes the HITS algorithm. Silverstein *et al.* (1998) investigate a log of a billion Web searches. 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). Recent information extraction has been pushed forward by the annual Message Understand Conferences (MUC), sponsored by the U.S. government. The FASTUS finite-state system was done by Hobbs *et al.* (1997). It was based in part on the idea from Pereira and Wright (1991) of using FSAs as approximations to phrase-structure grammars. Surveys of template-based systems are given by Roche and Schabes (1997), Appelt (1999),

and Muslea (1999). Large databases of facts were extracted by Craven *et al.* (2000), Pasca *et al.* (2006), Mitchell (2007), and Durme and Pasca (2008).

Freitag and McCallum (2000) discuss HMMs for Information Extraction. CRFs were introduced by Lafferty *et al.* (2001); an example of their use for information extraction is described in (McCallum, 2003) and a tutorial with practical guidance is given by (Sutton and McCallum, 2007). Sarawagi (2007) gives a comprehensive survey.

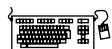
Banko *et al.* (2002) present the ASKMSR question-answering system; a similar system is due to Kwok *et al.* (2001). Pasca and Harabagiu (2001) discuss a contest-winning question-answering system. Two early influential approaches to automated knowledge engineering were by Riloff (1993), who showed that an automatically constructed dictionary performed almost as well as a carefully handcrafted domain-specific dictionary, and by Yarowsky (1995), who showed that the task of word sense classification (see page 756) could be accomplished through unsupervised training on a corpus of unlabeled text with accuracy as good as supervised methods.

The idea of simultaneously extracting templates and examples from a handful of labeled examples was developed independently and simultaneously by Blum and Mitchell (1998), who called it **cotraining** and by Brin (1998), who called it DIPRE (Dual Iterative Pattern Relation Extraction). You can see why the term *cotraining* has stuck. Similar early work, under the name of bootstrapping, was done by Jones *et al.* (1999). The method was advanced by the QXTRACT (Agichtein and Gravano, 2003) and KNOWITALL (Etzioni *et al.*, 2005) systems. Machine reading was introduced by Mitchell (2005) and Etzioni *et al.* (2006) and is the focus of the TEXTRUNNER project (Banko *et al.*, 2007; Banko and Etzioni, 2008).

This chapter has focused on natural language text, but it is also possible to do information extraction based on the physical structure or layout of text rather than on the linguistic structure. HTML lists and tables in both HTML and relational databases are home to data that can be extracted and consolidated (Hurst, 2000; Pinto *et al.*, 2003; Cafarella *et al.*, 2008).

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). The textbook by Manning and Schütze (1999) covers statistical language processing, while Jurafsky and Martin (2008) give a comprehensive introduction to speech and natural language processing.

EXERCISES



22.1 This exercise explores the quality of the n -gram model of language. Find or create a monolingual corpus of 100,000 words or more. Segment it into words, and compute the frequency of each word. How many distinct words are there? 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.



22.2 Write a program to do **segmentation** of words without spaces. Given a string, such as the URL “thelongestlistofthelongeststuffatthelongestdomainnameatlonglast.com,” return a list of component words: [“the,” “longest,” “list,” . . .]. This task is useful for parsing URLs, for spelling correction when words run together, and for languages such as Chinese that do not have spaces between words. It can be solved with a unigram or bigram word model and a dynamic programming algorithm similar to the Viterbi algorithm.



STYLOMETRY

22.3 (Adapted from Jurafsky and Martin (2000).) In this exercise you will develop a classifier for authorship: given a text, the classifier predicts 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 language model on the training set. You can choose what features to use; n -grams of words or letters are the easiest, but you can add additional features that you think may help. Then compute the probability of the text under each language model and chose the most probable model. Assess the accuracy of this technique. How does accuracy change as you alter the set of features? This subfield of linguistics is called **stylometry**; its successes include the identification of the author of the disputed *Federalist Papers* (Mosteller and Wallace, 1964) and some disputed works of Shakespeare (Hope, 1994). Khmelev and Tweedie (2001) produce good results with a simple letter bigram model.



22.4 This exercise concerns the classification of spam email. 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, SVM, logistic regression, or some other algorithm of your choosing) on a training set and report its accuracy on a test set.

22.5 Create a test set of ten queries, and pose them to three major Web search engines. Evaluate each one for precision at 1, 3, and 10 documents. Can you explain the differences between engines?



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

22.7 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.

22.8 Consider the problem of trying to evaluate the quality of an IR system that returns a ranked list of answers (like most Web search engines). The appropriate measure of quality depends on the presumed model of what the searcher is trying to achieve, and what strategy she employs. For each of the following models, propose a corresponding numeric measure.

- a. The searcher will look at the first twenty answers returned, with the objective of getting as much relevant information as possible.
- b. The searcher needs only one relevant document, and will go down the list until she finds the first one.
- c. The searcher has a fairly narrow query and is able to examine all the answers retrieved. She wants to be sure that she has seen everything in the document collection that is

relevant to her query. (E.g., a lawyer wants to be sure that she has found *all* relevant precedents, and is willing to spend considerable resources on that.)

- d. The searcher needs just one document relevant to the query, and can afford to pay a research assistant for an hour's work looking through the results. The assistant can look through 100 retrieved documents in an hour. The assistant will charge the searcher for the full hour regardless of whether he finds it immediately or at the end of the hour.
- e. The searcher will look through all the answers. Examining a document has cost $\$A$; finding a relevant document has value $\$B$; failing to find a relevant document has cost $\$C$ for each relevant document not found.
- f. The searcher wants to collect as many relevant documents as possible, but needs steady encouragement. She looks through the documents in order. If the documents she has looked at so far are mostly good, she will continue; otherwise, she will stop.

23

NATURAL LANGUAGE FOR COMMUNICATION

In which we see how humans communicate with one another in natural language, and how computer agents might join in the conversation.

COMMUNICATION
SIGN

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. Humans are the most chatty of all species, and if computer agents are to be helpful, they'll need to learn to speak the language. In this chapter we look at language models for communication. Models aimed at deep understanding of a conversation necessarily need to be more complex than the simple models aimed at, say, spam classification. We start with grammatical models of the phrase structure of sentences, add semantics to the model, and then apply it to machine translation and speech recognition.

23.1 PHRASE STRUCTURE GRAMMARS

LEXICAL CATEGORY

SYNTACTIC CATEGORIES

PHRASE STRUCTURE

The n -gram language models of Chapter 22 were based on sequences of words. The big issue for these models is **data sparsity**—with a vocabulary of, say, 10^5 words, there are 10^{15} trigram probabilities to estimate, and so a corpus of even a trillion words will not be able to supply reliable estimates for all of them. We can address the problem of sparsity through generalization. From the fact that “black dog” is more frequent than “dog black” and similar observations, we can form the generalization that adjectives tend to come before nouns in English (whereas they tend to follow nouns in French: “chien noir” is more frequent). Of course there are always exceptions; “galore” is an adjective that follows the noun it modifies. Despite the exceptions, the notion of a **lexical category** (also known as a **part of speech**) such as *noun* or *adjective* is a useful generalization—useful in its own right, but more so when we string together lexical categories to form **syntactic categories** such as *noun phrase* or *verb phrase*, and combine these syntactic categories into trees representing the **phrase structure** of sentences: nested phrases, each marked with a category.

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 $A \ B \ C \rightarrow D \ E$. 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 $A \ X \ B \rightarrow A \ Y \ B$ says that an X can be rewritten as a Y 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 1980s, linguists focused on context-free and context-sensitive languages. Since then, there has been renewed interest in regular grammars, brought about by the need to process and learn from gigabytes or terabytes 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) (and note that these three authors all now work on large text corpora at Google).

PROBABILISTIC
CONTEXT-FREE
GRAMMAR

LANGUAGE

NON-TERMINAL
SYMBOLS

TERMINAL SYMBOL

OPEN CLASS
CLOSED CLASS

PARSE TREE

There have been many competing language models based on the idea of phrase structure; we will describe a popular model called the **probabilistic context-free grammar**, or PCFG.¹ A **grammar** is a collection of rules that defines a **language** as a set of allowable strings of words. “Context-free” is described in the sidebar on page 889, and “probabilistic” means that the grammar assigns a probability to every string. Here is a PCFG rule:

$$\begin{array}{l} VP \rightarrow \text{Verb} [0.70] \\ | \quad VP \text{ } NP [0.30]. \end{array}$$

Here *VP* (*verb phrase*) and *NP* (*noun phrase*) are **non-terminal symbols**. The grammar also refers to actual words, which are called **terminal symbols**. This rule is saying that with probability 0.70 a verb phrase consists solely of a verb, and with probability 0.30 it is a *VP* followed by an *NP*. Appendix B describes non-probabilistic context-free grammars.

We now define a grammar for a tiny fragment of English that is suitable for communication between agents exploring the wumpus world. We call this language \mathcal{E}_0 . Later sections improve on \mathcal{E}_0 to make it slightly 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.

23.1.1 The lexicon of \mathcal{E}_0

First we define the **lexicon**, or list of allowable words. The words are grouped into the lexical categories familiar to dictionary users: nouns, pronouns, and names to denote things; verbs to denote events; adjectives to modify nouns; adverbs to modify verbs; and function words: articles (such as *the*), prepositions (*in*), and conjunctions (*and*). Figure 23.1 shows a small lexicon for the language \mathcal{E}_0 .

Each of the categories ends in *...* to indicate that there are other words in the category. For nouns, names, verbs, adjectives, and adverbs, it is infeasible even in principle to list all the words. Not only are there tens of thousands of members in each class, but new ones—like *iPod* or *biodiesel*—are being added constantly. These five categories are called **open classes**. For the categories of pronoun, relative pronoun, article, preposition, and conjunction we could have listed all the words with a little more work. These are called **closed classes**; they have a small number of words (a dozen or so). 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.

23.1.2 The Grammar of \mathcal{E}_0

The next step is to combine the words into phrases. Figure 23.2 shows a grammar for \mathcal{E}_0 , with rules for each of the six syntactic categories and an example for each rewrite rule.² Figure 23.3 shows a **parse tree** for the sentence “Every wumpus smells.” The parse tree

¹ PCFGs are also known as stochastic context-free grammars, or SCFGs.

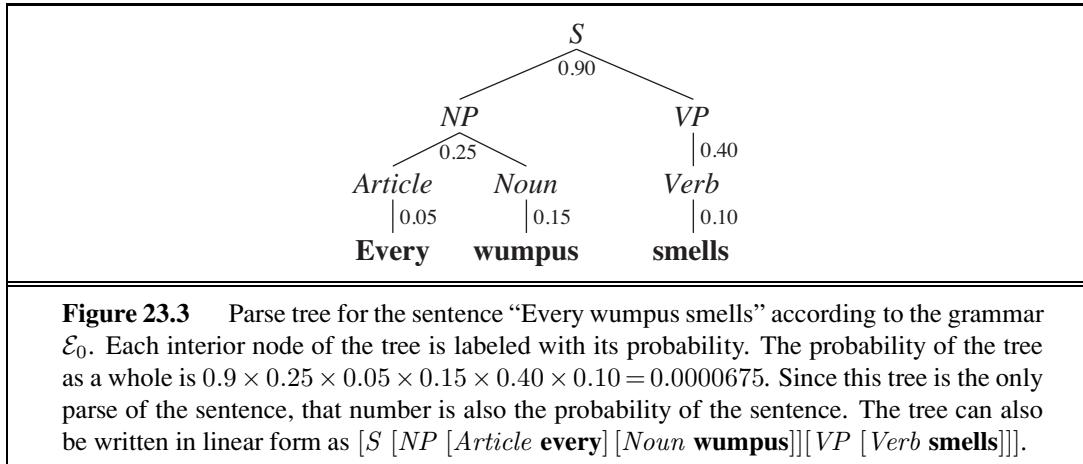
² A relative clause follows and modifies a noun phrase. It consists of a relative pronoun (such as “who” or “that”) followed by a verb phrase. An example of a relative clause is *that stinks* in “The wumpus *that stinks* is in 2 2.” Another kind of relative clause has no relative pronoun, e.g., *I know* in “the man *I know*.”

<i>Noun</i>	\rightarrow	stench [0.05] breeze [0.10] wumpus [0.15] pits [0.05] ...
<i>Verb</i>	\rightarrow	is [0.10] feel [0.10] smells [0.10] stinks [0.05] ...
<i>Adjective</i>	\rightarrow	right [0.10] dead [0.05] smelly [0.02] breezy [0.02] ...
<i>Adverb</i>	\rightarrow	here [0.05] ahead [0.05] nearby [0.02] ...
<i>Pronoun</i>	\rightarrow	me [0.10] you [0.03] I [0.10] it [0.10] ...
<i>RelPro</i>	\rightarrow	that [0.40] which [0.15] who [0.20] whom [0.02] \vee ...
<i>Name</i>	\rightarrow	John [0.01] Mary [0.01] Boston [0.01] ...
<i>Article</i>	\rightarrow	the [0.40] a [0.30] an [0.10] every [0.05] ...
<i>Prep</i>	\rightarrow	to [0.20] in [0.10] on [0.05] near [0.10] ...
<i>Conj</i>	\rightarrow	and [0.50] or [0.10] but [0.20] yet [0.02] \vee ...
<i>Digit</i>	\rightarrow	0 [0.20] 1 [0.20] 2 [0.20] 3 [0.20] 4 [0.20] ...

Figure 23.1 The lexicon for \mathcal{E}_0 . *RelPro* is short for relative pronoun, *Prep* for preposition, and *Conj* for conjunction. The sum of the probabilities for each category is 1.

$\mathcal{E}_0 :$	$S \rightarrow NP VP$	[0.90] I + feel a breeze
	$S Conj S$	[0.10] I feel a breeze + and + It stinks
	$NP \rightarrow Pronoun$	[0.30] I
	$Name$	[0.10] John
	$Noun$	[0.10] pits
	$Article Noun$	[0.25] the + wumpus
	$Article Adjs Noun$	[0.05] the + smelly dead + wumpus
	$Digit Digit$	[0.05] 3 4
	$NP PP$	[0.10] the wumpus + in 1 3
	$NP RelClause$	[0.05] the wumpus + that is smelly
	$VP \rightarrow Verb$	[0.40] stinks
	$VP NP$	[0.35] feel + a breeze
	$VP Adjective$	[0.05] smells + dead
	$VP PP$	[0.10] is + in 1 3
	$VP Adverb$	[0.10] go + ahead
	$Adjs \rightarrow Adjective$	[0.80] smelly
	$Adjective Adjs$	[0.20] smelly + dead
	$PP \rightarrow Prep NP$	[1.00] to + the east
	$RelClause \rightarrow RelPro VP$	[1.00] that + is smelly

Figure 23.2 The grammar for \mathcal{E}_0 , with example phrases for each rule. The syntactic categories are sentence (*S*), noun phrase (*NP*), verb phrase (*VP*), list of adjectives (*Adjs*), prepositional phrase (*PP*), and relative clause (*RelClause*).



gives a constructive proof that the string of words is indeed a sentence according to the rules of \mathcal{E}_0 . The \mathcal{E}_0 grammar generates a wide range of English sentences such as the following:

John is in the pit

The wumpus that stinks is in 2 2

Mary is in Boston and the wumpus is near 3 2

OVERGENERATION
UNDERGENERATION

Unfortunately, the grammar **overgenerates**: that is, it generates sentences that are not grammatical, such as “Me go Boston” and “I smell pits wumpus John.” It also **undergenerates**: there are many sentences of English that it rejects, such as “I think the wumpus is smelly.” We will see how to learn a better grammar later; for now we concentrate on what we can do with the grammar we have.

23.2 SYNTACTIC ANALYSIS (PARSING)

PARSING

Parsing is the process of analyzing a string of words to uncover its phrase structure, according to the rules of a grammar. Figure 23.4 shows that we can start with the S symbol and search top down for a tree that has the words as its leaves, or we can start with the words and search bottom up for a tree that culminates in an S . Both top-down and bottom-up parsing can be inefficient, however, because they can end up repeating effort in areas of the search space that lead to dead ends. 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 and reanalyze the whole sentence under the other interpretation.

<i>List of items</i>	<i>Rule</i>
<i>S</i>	
<i>NP VP</i>	$S \rightarrow NP VP$
<i>NP VP Adjective</i>	$VP \rightarrow VP Adjective$
<i>NP Verb Adjective</i>	$VP \rightarrow Verb$
<i>NP Verb dead</i>	$Adjective \rightarrow dead$
<i>NP is dead</i>	$Verb \rightarrow is$
<i>Article Noun is dead</i>	$NP \rightarrow Article Noun$
<i>Article wumpus is dead</i>	$Noun \rightarrow wumpus$
the wumpus is dead	$Article \rightarrow the$

Figure 23.4 Trace of the process of finding a parse for the string “The wumpus is dead” as a sentence, according to the grammar \mathcal{E}_0 . Viewed as a top-down parse, we start with the list of items being *S* and, on each step, match an item *X* with a rule of the form ($X \rightarrow \dots$) and replace *X* in the list of items with (...). Viewed as a bottom-up parse, we start with the list of items being the words of the sentence, and, on each step, match a string of tokens (...) in the list against a rule of the form ($X \rightarrow \dots$) and replace (...) with *X*.



CHART

CYK ALGORITHM

CHOMSKY NORMAL FORM

To avoid this source of inefficiency we can use dynamic programming: *every time we analyze a substring, store the results so we won’t have to reanalyze 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. There are many types of chart parsers; we describe a bottom-up version called the **CYK algorithm**, after its inventors, John Cocke, Daniel Younger, and Tadeo Kasami.

The CYK algorithm is shown in Figure 23.5. Note that it requires a grammar with all rules in one of two very specific formats: lexical rules of the form $X \rightarrow \text{word}$, and syntactic rules of the form $X \rightarrow Y Z$. This grammar format, called **Chomsky Normal Form**, may seem restrictive, but it is not: any context-free grammar can be automatically transformed into Chomsky Normal Form. Exercise 23.8 leads you through the process.

The CYK algorithm uses space of $O(n^2m)$ for the *P* table, where *n* is the number of words in the sentence, and *m* is the number of nonterminal symbols in the grammar, and takes time $O(n^3m)$. (Since *m* is constant for a particular grammar, this is commonly described as $O(n^3)$.) No algorithm can do better for general context-free grammars, although there are faster algorithms on more restricted grammars. In fact, it is quite a trick for the algorithm to complete in $O(n^3)$ time, given that it is possible for a sentence to have an exponential number of parse trees. Consider the sentence

Fall leaves fall and spring leaves spring.

It is ambiguous because each word (except “and”) can be either a noun or a verb, and “fall” and “spring” can be adjectives as well. (For example, one meaning of “Fall leaves fall” is

```

function CYK-PARSE(words, grammar) returns P, a table of probabilities
  N  $\leftarrow$  LENGTH(words)
  M  $\leftarrow$  the number of nonterminal symbols in grammar
  P  $\leftarrow$  an array of size [M, N, N], initially all 0
  /* Insert lexical rules for each word */
  for i = 1 to N do
    for each rule of form (X  $\rightarrow$  wordsi [p]) do
      P[X, i, 1]  $\leftarrow$  p
    /* Combine first and second parts of right-hand sides of rules, from short to long */
    for length = 2 to N do
      for start = 1 to N - length + 1 do
        for len1 = 1 to N - 1 do
          len2  $\leftarrow$  length - len1
          for each rule of the form (X  $\rightarrow$  Y Z [p]) do
            P[X, start, length]  $\leftarrow$  MAX(P[X, start, length],
                                         P[Y, start, len1]  $\times$  P[Z, start + len1, len2]  $\times$  p)
    return P

```

Figure 23.5 The CYK algorithm for parsing. Given a sequence of words, it finds the most probable derivation for the whole sequence and for each subsequence. It returns the whole table, *P*, in which an entry *P[X, start, len]* is the probability of the most probable *X* of length *len* starting at position *start*. If there is no *X* of that size at that location, the probability is 0.

equivalent to “Autumn abandons autumn.) With \mathcal{E}_0 the sentence has four parses:

[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 c two-ways-ambiguous conjoined subsentences, we would have 2^c ways of choosing parses for the subsentences.³ How does the CYK algorithm process these 2^c parse trees in $O(c^3)$ time? The answer is that it doesn’t examine all the parse trees; all it has to do is compute the probability of the most probable tree. The subtrees are all represented in the *P* table, and with a little work we could enumerate them all (in exponential time), but the beauty of the CYK algorithm is that we don’t have to enumerate them unless we want to.

In practice we are usually not interested in all parses; just the best one or best few. Think of the CYK algorithm as defining the complete state space defined by the “apply grammar rule” operator. It is possible to search just part of this space using A^* search. Each state in this space is a list of items (words or categories), as shown in the bottom-up parse table (Figure 23.4). The start state is a list of words, and a goal state is the single item *S*. The

³ There also would be $O(c!)$ ambiguity in the way the components conjoin—for example, (*X* and (*Y* and *Z*)) versus ((*X* and *Y*) and *Z*). But that is another story, one told well by Church and Patil (1982).

```

[ [S [NP-SBJ-2 Her eyes]
  [VP were
    [VP glazed
      [NP *-2]
      [SBAR-ADV as if
        [S [NP-SBJ she]
        [VP did n't
          [VP [VP hear [NP *-1]]
          or
          [VP [ADVP even] see [NP *-1]]
          [NP-1 him]]]]]]]
  .]

```

Figure 23.6 Annotated tree for the sentence “Her eyes were glazed as if she didn’t hear or even see him.” from the Penn Treebank. Note that in this grammar there is a distinction between an object noun phrase (*NP*) and a subject noun phrase (*NP-SBJ*). Note also a grammatical phenomenon we have not covered yet: the movement of a phrase from one part of the tree to another. This tree analyzes the phrase “hear or even see him” as consisting of two constituent *VPs*, *[VP hear [NP *-1]]* and *[VP [ADVP even] see [NP *-1]]*, both of which have a missing object, denoted **-1*, which refers to the *NP* labeled elsewhere in the tree as *[NP-1 him]*.

cost of a state is the inverse of its probability as defined by the rules applied so far, and there are various heuristics to estimate the remaining distance to the goal; the best heuristics come from machine learning applied to a corpus of sentences. With the *A** algorithm we don’t have to search the entire state space, and we are guaranteed that the first parse found will be the most probable.

23.2.1 Learning probabilities for PCFGs

TREEBANK

A PCFG has many rules, with a probability for each rule. This suggests that **learning** the grammar from data might be better than a knowledge engineering approach. Learning is easiest if we are given a corpus of correctly parsed sentences, commonly called a **treebank**. The Penn Treebank (Marcus *et al.*, 1993) is the best known; it consists of 3 million words which have been annotated with part of speech and parse-tree structure, using human labor assisted by some automated tools. Figure 23.6 shows an annotated tree from the Penn Treebank.

Given a corpus of trees, we can create a PCFG just by counting (and smoothing). In the example above, there are two nodes of the form *[S[NP ...][VP ...]]*. We would count these, and all the other subtrees with root *S* in the corpus. If there are 100,000 *S* nodes of which 60,000 are of this form, then we create the rule:

$$S \rightarrow NP\ VP\ [0.60].$$

What if a treebank is not available, but we have a corpus of raw unlabeled sentences? It is still possible to learn a grammar from such a corpus, but it is more difficult. 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 Bayes nets.) We'll assume that we're given the lexical and syntactic category names. (If not, we can just assume categories X_1, \dots, X_n and use cross-validation to pick the best value of n .) We can then assume that the grammar includes every possible ($X \rightarrow Y Z$) or ($X \rightarrow \text{word}$) rule, although many of these rules will have probability 0 or close to 0.

We can then use an expectation–maximization (EM) approach, just as we did in learning HMMs. The parameters we are trying to learn are the rule probabilities; we start them off at random or uniform values. 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 ($X \rightarrow \dots$). The E step estimates the probability that each subsequence is generated by each rule. The M step then estimates the probability of each rule. The whole computation can be done in a dynamic-programming fashion with an algorithm called the **inside–outside algorithm** in analogy to the forward–backward algorithm for HMMs.

INSIDE-OUTSIDE ALGORITHM

The inside–outside algorithm seems magical in that it induces a grammar from unparsed text. But it has several drawbacks. First, the parses that are assigned by the induced grammars are often difficult to understand and unsatisfying to linguists. This makes it hard to combine handcrafted knowledge with automated induction. Second, it is slow: $O(n^3m^3)$, where n is the number of words in a sentence and m is the number of grammar categories. Third, 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 get closer to the global maximum, at a cost of even more computation. Lari and Young (1990) conclude that inside–outside is “computationally intractable for realistic problems.”

However, progress can be made if we are willing to step outside the bounds of learning solely from unparsed text. One approach is to learn from **prototypes**: to seed the process with a dozen or two rules, similar to the rules in \mathcal{E}_1 . From there, more complex rules can be learned more easily, and the resulting grammar parses English with an overall recall and precision for sentences of about 80% (Haghghi and Klein, 2006). Another approach is to use treebanks, but in addition to learning PCFG rules directly from the bracketings, also learning distinctions that are not in the treebank. For example, note that the tree in Figure 23.6 makes the distinction between NP and $NP - SBJ$. The latter is used for the pronoun “she,” the former for the pronoun “her.” We will explore this issue in Section 23.6; for now let us just say that there are many ways in which it would be useful to **split** a category like NP —grammar induction systems that use treebanks but automatically split categories do better than those that stick with the original category set (Petrov and Klein, 2007c). The error rates for automatically learned grammars are still about 50% higher than for hand-constructed grammar, but the gap is decreasing.

23.2.2 Comparing context-free and Markov models

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 $P(\text{Noun} \rightarrow \text{"banana"})$ versus $P(\text{Noun} \rightarrow \text{"bandanna"})$ and not on the relation between “eat” and the respective objects. A Markov model of order two or more, given a sufficiently large corpus, *will* know that “eat

a banana” is more probable. We can combine a PCFG and Markov model to get the best of both. The simplest approach is to estimate the probability of a sentence with the geometric mean of the probabilities computed by both models. Then we would know that “eat a banana” is probable from both the grammatical and lexical point of view. But it still wouldn’t pick up the relation between “eat” and “banana” in “eat a slightly aging but still palatable banana” because here the relation is more than two words away. Increasing the order of the Markov model won’t get at the relation precisely; to do that we can use a **lexicalized** PCFG, as described in the next section.

Another problem with PCFGs is that they 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 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 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.

23.3 AUGMENTED GRAMMARS AND SEMANTIC INTERPRETATION

In this section we see how to extend context-free grammars—to say that, for example, not every *NP* is independent of context, but rather, certain *NPs* are more likely to appear in one context, and others in another context.

23.3.1 Lexicalized PCFGs

To get at the relationship between the verb “eat” and the nouns “banana” versus “bandanna,” we can use a **lexicalized PCFG**, in which the probabilities for a rule depend on the relationship between words in the parse tree, not just on the adjacency of words in a sentence. Of course, we can’t have the probability depend on every word in the tree, because we won’t have enough training data to estimate all those probabilities. It is useful to introduce the notion of the **head** of a phrase—the most important word. Thus, “eat” is the head of the *VP* “eat a banana” and “banana” is the head of the *NP* “a banana.” We use the notation *VP(v)* to denote a phrase with category *VP* whose head word is *v*. We say that the category *VP* is **augmented** with the head variable *v*. Here is an **augmented grammar** that describes the verb-object relation:

$$\begin{array}{ll}
 VP(v) \rightarrow Verb(v) NP(n) & [P_1(v, n)] \\
 VP(v) \rightarrow Verb(v) & [P_2(v)] \\
 NP(n) \rightarrow Article(a) Adj(j) Noun(n) & [P_3(n, a)] \\
 Noun(banana) \rightarrow banana & [p_n] \\
 \dots & \dots
 \end{array}$$

Here the probability $P_1(v, n)$ depends on the head words *v* and *n*. We would set this probability to be relatively high when *v* is “eat” and *n* is “banana,” and low when *n* is “bandanna.”

LEXICALIZED PCFG

HEAD

AUGMENTED GRAMMAR

Note that since we are considering only heads, the distinction between “eat a banana” and “eat a rancid banana” will not be caught by these probabilities. Another issue with this approach is that, in a vocabulary with, say, 20,000 nouns and 5,000 verbs, P_1 needs 100 million probability estimates. Only a few percent of these can come from a corpus; the rest will have to come from smoothing (see Section 22.1.2). For example, we can estimate $P_1(v, n)$ for a (v, n) pair that we have not seen often (or at all) by backing off to a model that depends only on v . These objectless probabilities are still very useful; they can capture the distinction between a transitive verb like “eat”—which will have a high value for P_1 and a low value for P_2 —and an intransitive verb like “sleep,” which will have the reverse. It is quite feasible to learn these probabilities from a treebank.

23.3.2 Formal definition of augmented grammar rules

DEFINITE CLAUSE
GRAMMAR

Augmented rules are complicated, so we will give them a formal definition by showing how an augmented rule can be translated into a logical sentence. The sentence will have the form of a definite clause (see page 256), so the result is called a **definite clause grammar**, or DCG. We’ll use as an example a version of a rule from the lexicalized grammar for NP with one new piece of notation:

$$NP(n) \rightarrow Article(a) \ Adjs(j) \ Noun(n) \{ Compatible(j, n) \} .$$

The new aspect here is the notation $\{ constraint \}$ to denote a logical constraint on some of the variables; the rule only holds when the constraint is true. Here the predicate $Compatible(j, n)$ is meant to test whether adjective j and noun n are compatible; it would be defined by a series of assertions such as $Compatible(\mathbf{black}, \mathbf{dog})$. We can convert this grammar rule into a definite clause by (1) reversing the order of right- and left-hand sides, (2) making a conjunction of all the constituents and constraints, (3) adding a variable s_i to the list of arguments for each constituent to represent the sequence of words spanned by the constituent, (4) adding a term for the concatenation of words, $Append(s_1, \dots)$, to the list of arguments for the root of the tree. That gives us

$$\begin{aligned} Article(a, s_1) \wedge Adjs(j, s_2) \wedge Noun(n, s_3) \wedge Compatible(j, n) \\ \Rightarrow NP(n, Append(s_1, s_2, s_3)) . \end{aligned}$$

This definite clause says that if the predicate *Article* is true of a head word a and a string s_1 , and *Adjs* is similarly true of a head word j and a string s_2 , and *Noun* is true of a head word n and a string s_3 , and if j and n are compatible, then the predicate *NP* is true of the head word n and the result of appending strings s_1 , s_2 , and s_3 .

The DCG translation left out the probabilities, but we could put them back in: just augment each constituent with one more variable representing the probability of the constituent, and augment the root with a variable that is the product of the constituent probabilities times the rule probability.

The translation from grammar rule to definite clause allows 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. In fact, parsing natural language with DCGs was

one of the first applications of (and motivations for) the Prolog logic programming language. It is sometimes possible to run the process backward and do **language generation** as well as parsing. For example, skipping ahead to Figure 23.10 (page 903), a logic program could be given the semantic form *Loves(John, Mary)* and apply the definite-clause rules to deduce

$$S(\text{Loves}(\text{John}, \text{Mary}), [\text{John}, \text{loves}, \text{Mary}]).$$

This works for toy examples, but serious language-generation systems need more control over the process than is afforded by the DCG rules alone.

$\mathcal{E}_1 :$	$S \rightarrow NP_S VP \mid \dots$ $NP_S \rightarrow \text{Pronoun}_S \mid \text{Name} \mid \text{Noun} \mid \dots$ $NP_O \rightarrow \text{Pronoun}_O \mid \text{Name} \mid \text{Noun} \mid \dots$ $VP \rightarrow VP\ NP_O \mid \dots$ $PP \rightarrow \text{Prep}\ NP_O$ $\text{Pronoun}_S \rightarrow \text{I} \mid \text{you} \mid \text{he} \mid \text{she} \mid \text{it} \mid \dots$ $\text{Pronoun}_O \rightarrow \text{me} \mid \text{you} \mid \text{him} \mid \text{her} \mid \text{it} \mid \dots$ \dots
$\mathcal{E}_2 :$	$S(\text{head}) \rightarrow NP(Sbj, pn, h)\ VP(pn, head) \mid \dots$ $NP(c, pn, head) \rightarrow \text{Pronoun}(c, pn, head) \mid \text{Noun}(c, pn, head) \mid \dots$ $VP(pn, head) \rightarrow VP(pn, head)\ NP(Obj, p, h) \mid \dots$ $PP(head) \rightarrow \text{Prep}(head)\ NP(Obj, pn, h)$ $\text{Pronoun}(Sbj, 1S, \text{I}) \rightarrow \text{I}$ $\text{Pronoun}(Sbj, 1P, \text{we}) \rightarrow \text{we}$ $\text{Pronoun}(Obj, 1S, \text{me}) \rightarrow \text{me}$ $\text{Pronoun}(Obj, 3P, \text{them}) \rightarrow \text{them}$ \dots

Figure 23.7 Top: part of a grammar for the language \mathcal{E}_1 , which handles subjective and objective cases in noun phrases and thus does not overgenerate quite as badly as \mathcal{E}_0 . The portions that are identical to \mathcal{E}_0 have been omitted. Bottom: part of an augmented grammar for \mathcal{E}_2 , with three augmentations: case agreement, subject–verb agreement, and head word. *Sbj*, *Obj*, *1S*, *1P* and *3P* are constants, and lowercase names are variables.

23.3.3 Case agreement and subject–verb agreement

We saw in Section 23.1 that the simple grammar for \mathcal{E}_0 overgenerates, producing nonsentences 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.⁴ We can account for this by

⁴ 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.

CASE AGREEMENT

SUBJECT-VERB
AGREEMENT

splitting NP into two categories, NP_S and NP_O , to stand for noun phrases in the subjective and objective case, respectively. We would also need to split the category *Pronoun* into the two categories *Pronoun_S* (which includes “I”) and *Pronoun_O* (which includes “me”). The top part of Figure 23.7 shows the 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 requires **subject–verb agreement** for person and number of 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: the verb “to be” has three forms, “I am / you are / he is.” So one distinction (case) splits NP two ways, another distinction (person and number) splits NP three ways, and as we uncover other distinctions we would end up with an exponential number of subscripted NP forms if we took the approach of \mathcal{E}_1 . Augmentations are a better approach: they can represent an exponential number of forms as a single rule.

In the bottom of Figure 23.7 we see (part of) an augmented grammar for the language \mathcal{E}_2 , which handles case agreement, subject–verb agreement, and head words. We have just one NP category, but $NP(c, pn, head)$ has three augmentations: c is a parameter for case, pn is a parameter for person and number, and *head* is a parameter for the head word of the phrase. The other categories also are augmented with heads and other arguments. Let’s consider one rule in detail:

$$S(head) \rightarrow NP(Sbj, pn, h) VP(pn, head).$$

This rule is easiest to understand right-to-left: when an NP and a VP are conjoined they form an S , but only if the NP has the subjective (*Sbj*) case and the person and number (*pn*) of the NP and VP are identical. If that holds, then we have an S whose head is the same as the head of the VP . Note the head of the NP , denoted by the dummy variable *h*, is not part of the augmentation of the S . The lexical rules for \mathcal{E}_2 fill in the values of the parameters and are also best read right-to-left. For example, the rule

$$Pronoun(Sbj, 1S, I) \rightarrow I$$

says that “I” can be interpreted as a *Pronoun* in the subjective case, first-person singular, with head “I.” For simplicity we have omitted the probabilities for these rules, but augmentation does work with probabilities. Augmentation can also work with automated learning mechanisms. Petrov and Klein (2007c) show how a learning algorithm can automatically split the NP category into NP_S and NP_O .

23.3.4 Semantic interpretation

To show how to add semantics to a grammar, we start with an example that is simpler than English: the semantics of arithmetic expressions. Figure 23.8 shows a grammar for arithmetic expressions, where each rule is augmented with a variable indicating the semantic interpretation of the phrase. The semantics of a digit such as “3” is the digit itself. The semantics of an expression such as “3 + 4” is the operator “+” applied to the semantics of the phrase “3” and

```

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

```

Figure 23.8 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.

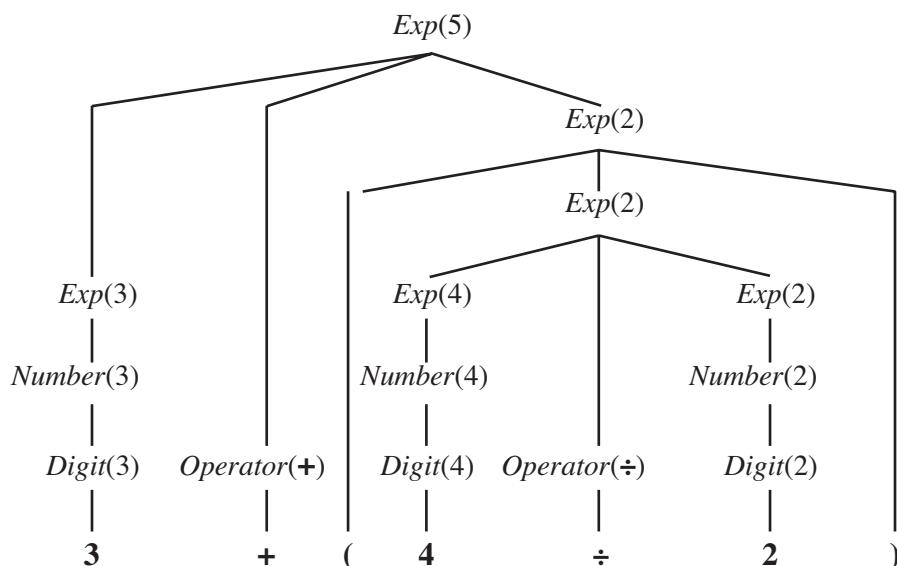


Figure 23.9 Parse tree with semantic interpretations for the string “ $3 + (4 \div 2)$ ”.

the phrase “4.” The rules obey the principle of **compositional semantics**—the semantics of a phrase is a function of the semantics of the subphrases. Figure 23.9 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.

Now let’s move on to the semantics of English, or at least of \mathcal{E}_0 . We start by determining what semantic representations we want to associate with what phrases. We use the simple example sentence “John loves Mary.” The *NP* “John” should have as its semantic interpretation the logical term *John*, and the sentence as a whole should have as its interpretation the logical sentence *Loves(John, Mary)*. That much seems clear. The complicated part is the *VP* “loves Mary.” The semantic interpretation of this phrase is neither a logical term nor a complete logical sentence. Intuitively, “loves Mary” is a description that might or might not

apply to a particular person. (In this case, it applies to John.) This means that “loves Mary” is a **predicate** that, when combined with a term that represents a person (the person doing the loving), yields a complete logical sentence. Using the λ -notation (see page 294), we can represent “loves Mary” as the predicate

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

Now we need a rule that says “an *NP* with semantics *obj* followed by a *VP* with semantics *pred* yields a sentence whose semantics is the result of applying *pred* to *obj*:”

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

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

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

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 $\lambda y \lambda x \text{ Loves}(x, y)$, the predicate that, when given the argument *Mary*, returns the predicate $\lambda x \text{ Loves}(x, \text{Mary})$. We end up with the grammar shown in Figure 23.10 and the parse tree shown in Figure 23.11. We could just as easily have added semantics to \mathcal{E}_2 ; we chose to work with \mathcal{E}_0 so that the reader can focus on one type of augmentation at a time.

Adding semantic augmentations to a grammar by hand is laborious and error prone. Therefore, there have been several projects to learn semantic augmentations from examples. 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 semantic forms—for example;

What is the capital of the state with the largest population?

Answer(c, Capital(s, c) \wedge Largest(p, State(s) \wedge Population(s, p)))

CHILL’s task is to learn a predicate *Parse(words, semantics)* 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 can learn to achieve 70% to 85% accuracy on various database query tasks.

23.3.5 Complications

The grammar of real English is endlessly complex. We will briefly mention some examples.

Time and tense: 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

$$\begin{aligned} S(pred(obj)) &\rightarrow NP(obj) VP(pred) \\ VP(pred(obj)) &\rightarrow Verb(pred) NP(obj) \\ NP(obj) &\rightarrow Name(obj) \end{aligned}$$

$$\begin{aligned} Name(John) &\rightarrow \textbf{John} \\ Name(Mary) &\rightarrow \textbf{Mary} \\ Verb(\lambda y \lambda x Loves(x, y)) &\rightarrow \textbf{loves} \end{aligned}$$

Figure 23.10 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.

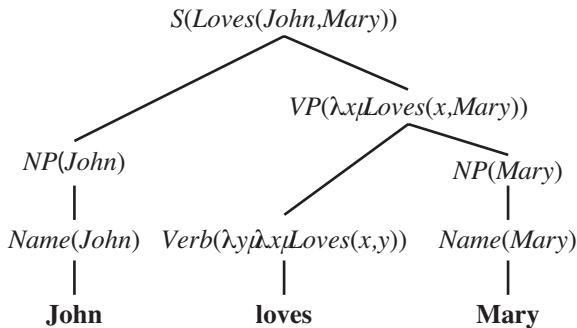


Figure 23.11 A parse tree with semantic interpretations for the string “John loves Mary”.

the relative time of an event. One good choice to represent the time of events is the event calculus notation of Section 12.3. In event calculus we have

John loves mary: $E_1 \in Loves(John, Mary) \wedge During(Now, Extent(E_1))$

John loved mary: $E_2 \in Loves(John, Mary) \wedge After(Now, Extent(E_2))$.

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

$$Verb(\lambda y \lambda x e \in Loves(x, y) \wedge During(Now, e)) \rightarrow \textbf{loves}$$

$$Verb(\lambda y \lambda x e \in Loves(x, y) \wedge After(Now, e)) \rightarrow \textbf{loved}.$$

Other than this change, everything else about the grammar remains the same, which is encouraging news; it suggests we are on the right track if we can so easily add a complication like the tense of verbs (although we have just scratched the surface of a complete grammar for time and tense). It is also encouraging that the distinction between processes and discrete events that we made in our discussion of knowledge representation in Section 12.3.1 is actually reflected in language use. We can say “John slept a lot last night,” where *Sleeping* is a process category, but it is odd to say “John found a unicorn a lot last night,” where *Finding* is a discrete event category. A grammar would reflect that fact by having a low probability for adding the adverbial phrase “a lot” to discrete events.

Quantification: Consider the sentence “Every agent feels a breeze.” The sentence has only one syntactic parse under \mathcal{E}_0 , but it is actually semantically ambiguous; the preferred

meaning is “For every agent there exists a breeze that the agent feels,” but an acceptable alternative meaning is “There exists a breeze that every agent feels.”⁵ The two interpretations can be represented as

$$\begin{aligned} \forall a \ a \in Agents \Rightarrow \\ \exists b \ b \in Breezes \wedge \exists e \ e \in Feel(a, b) \wedge During(Now, e) ; \\ \exists b \ b \in Breezes \forall a \ a \in Agents \Rightarrow \\ \exists e \ e \in Feel(a, b) \wedge During(Now, e) . \end{aligned}$$

QUASI-LOGICAL FORM

PRAGMATICS

INDEXICAL

SPEECH ACT

LONG-DISTANCE DEPENDENCIES

TRACE

AMBIGUITY

The standard approach to quantification is for the grammar to define not an actual logical semantic sentence, but rather a **quasi-logical form** that is then turned into a logical sentence by algorithms outside of the parsing process. Those algorithms can have preference rules for preferring one quantifier scope over another—preferences that need not be reflected directly in the grammar.

Pragmatics: 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. 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,” both “I” and “today” are indexicals. The word “I” would be represented by the fluent *Speaker*, and it would be up to the hearer to resolve the meaning of the fluent—that is not considered part of the grammar but rather an issue of pragmatics; of using the context of the current situation to interpret fluents.

Another part of pragmatics is interpreting the speaker’s intent. The speaker’s action is considered a **speech act**, and it is up to the hearer to decipher what type of action it is—a question, a statement, a promise, a warning, a command, and so on. 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. 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:

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

Long-distance dependencies: Questions introduce a new grammatical complexity. In “Who did the agent tell you to give the gold to?” the final word “to” should be parsed as [*PP* to $_\!$], where the “ $_\!$ ” denotes a gap or **trace** where an *NP* is missing; 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 in just the right way, and prohibit gaps in the wrong places. For example, you can’t have a gap in one branch of an *NP* conjunction: “What did he play [*NP* Dungeons and $_\!$]?” is ungrammatical. But you can have the same gap in both branches of a *VP* conjunction: “What did you [*VP* [*VP* smell $_\!$] and [*VP* shoot an arrow at $_\!$]]?”

Ambiguity: In some cases, hearers are consciously aware of ambiguity in an utterance. Here are some examples taken from newspaper headlines:

⁵ If this interpretation seems unlikely, consider “Every Protestant believes in a just God.”

Squad helps dog bite victim.
 Police begin campaign to run down jaywalkers.
 Helicopter powered by human flies.
 Once-sagging cloth diaper industry saved by full dumps.
 Portable toilet bombed; police have nothing to go on.
 Teacher strikes idle kids.
 Include your children when baking cookies.
 Hospitals are sued by 7 foot doctors.
 Milk drinkers are turning to powder.
 Safety experts say school bus passengers should be belted.



LEXICAL AMBIGUITY



SYNTACTIC AMBIGUITY



SEMANTIC AMBIGUITY



METONYMY

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. **Lexical ambiguity**, in which a word has more than one meaning, 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 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** refers to a phrase that has multiple parses: “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 for the agent.

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

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

We need to change that to

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

This says that there is one entity x that is equal to Chrysler, and another entity m that did the announcing, and that the two are in a metonymy relation. The next step is to define what kinds of metonymy relations can occur. The simplest case is when there is no metonymy at all—the literal object x and the metonymic object m are identical:

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

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 producer for the product (I drive a *Honda*) and the part for the whole (The Red Sox need a strong *arm*). Some examples of metonymy, such as “The *ham sandwich* on Table 4 wants another beer,” are more novel and are interpreted with respect to a situation.

METAPHOR

A **metaphor** is another figure of speech, in which a phrase with one literal meaning is used to suggest a different meaning by way of an analogy. Thus, metaphor can be seen as a kind of metonymy where the relation is one of similarity.

DISAMBIGUATION

Disambiguation is the process of recovering the most probable intended meaning of an utterance. In one sense we already have a framework for solving this problem: each rule has a probability associated with it, so the probability of an interpretation is the product of the probabilities of the rules that led to the interpretation. Unfortunately, the probabilities reflect how common the phrases are in the corpus from which the grammar was learned, and thus reflect general knowledge, not specific knowledge of the current situation. To do disambiguation properly, we need to combine four models:

1. The **world model**: the likelihood that a proposition occurs in the world. Given what we know about the world, it is more likely that a speaker who says “I’m dead” means “I am in big trouble” rather than “My life ended, and yet I can still talk.”
2. The **mental model**: the likelihood that the speaker forms the intention of communicating a certain fact to the hearer. This approach combines models of what the speaker believes, what the speaker believes the hearer believes, and so on. For example, when a politician says, “I am not a crook,” the world model might assign a probability of only 50% to the proposition that the politician is not a criminal, and 99.999% to the proposition that he is not a hooked shepherd’s staff. Nevertheless, we select the former interpretation because it is a more likely thing to say.
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.
4. The **acoustic model**: for spoken communication, the likelihood that a particular sequence of sounds will be generated, given that the speaker has chosen a given string of words. Section 23.5 covers speech recognition.

23.4 MACHINE TRANSLATION

Machine translation is the automatic translation of text from one natural language (the source) to another (the target). It was one of the first application areas envisioned for computers (Weaver, 1949), but it is only in the past decade that the technology has seen widespread usage. Here is a passage from page 1 of this book:

AI is one of the newest fields in science and engineering. Work started in earnest soon after World War II, and the name itself was coined in 1956. Along with molecular biology, AI is regularly cited as the “field I would most like to be in” by scientists in other disciplines.

And here it is translated from English to Danish by an online tool, Google Translate:

AI er en af de nyeste områder inden for videnskab og teknik. Arbejde startede for alvor lige efter Anden Verdenskrig, og navnet i sig selv var opfundet i 1956. Sammen med molekylær biologi, er AI jævnligt nævnt som “feltet Jeg ville de fleste gerne være i” af forskere i andre discipliner.

For those who don’t read Danish, here is the Danish translated back to English. The words that came out different are in *italics*:

AI is one of the newest fields *of* science and engineering. Work *began* in earnest *just* after the *Second* World War, and the name itself was *invented* in 1956. *Together* with molecular biology, AI is *frequently mentioned* as „“field I would most like to be in” by *researchers* in other disciplines.

The differences are all reasonable paraphrases, such as *frequently mentioned* for *regularly cited*. The only real error is the omission of the article *the*, denoted by the „ symbol. This is typical accuracy: of the two sentences, one has an error that would not be made by a native speaker, yet the meaning is clearly conveyed.

Historically, there have been three main applications of machine translation. *Rough translation*, as provided by free online services, gives the “gist” of a foreign sentence or document, but contains errors. *Pre-edited translation* is used by companies to publish their documentation and sales materials in multiple languages. The original source text is written in a constrained language that is easier to translate automatically, and the results are usually edited by a human to correct any errors. *Restricted-source translation* works fully automatically, but only on highly stereotypical language, such as a weather report.

Translation is difficult because, in the fully general case, it requires in-depth understanding of the text. This is true even for very simple texts—even “texts” of one word. Consider the word “Open” on the door of a store.⁶ 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 signs use the identical word to convey different meanings. In German the sign on the door would be “Offen” while the banner would read “Neu Eröffnet.”

⁶ This example is due to Martin Kay.

INTERLINGUA

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). Therefore, representing the meaning of a sentence is more difficult for translation than it is for single-language understanding. An English 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 Eröffnet” sense. A representation language that makes all the distinctions necessary for a set of languages is called an **interlingua**.

A translator (human or machine) often needs to understand the actual situation described in the source, not just the individual words. For example, to translate the English word “him,” into Korean, a choice must be made between the humble and honorific form, a choice that depends on the social relationship between the speaker and the referent of “him.” In Japanese, the honorifics are relative, so the choice depends on the social relationships between the speaker, the referent, and the listener. Translators (both machine and human) sometimes find it difficult to make this choice. As another example, to translate “The baseball hit the window. It broke.” into French, we must choose the feminine “elle” or the masculine “il” for “it,” so we must decide whether “it” refers to the baseball or the window. To get the translation right, one must understand physics as well as language.

Sometimes there is *no choice* that can yield a completely satisfactory translation. For example, an Italian love poem that uses the masculine “il sole” (sun) and feminine “la luna” (moon) to symbolize two lovers will necessarily be altered when translated into German, where the genders are reversed, and further altered when translated into a language where the genders are the same.⁷

23.4.1 Machine translation systems

TRANSFER MODEL

All translation systems must model the source and target languages, but systems vary in the type of models they use. Some systems attempt to analyze the source language text all the way into an interlingua knowledge representation and then generate sentences in the target language from that representation. This is difficult because it involves three unsolved problems: creating a complete knowledge representation of everything; parsing into that representation; and generating sentences from that representation.

Other systems are based on a **transfer model**. They keep a database 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]. Figure 23.12 diagrams the various transfer points.

⁷ Warren Weaver (1949) reports that Max Zeldner points out that the great Hebrew poet H. N. Bialik once said that translation “is like kissing the bride through a veil.”

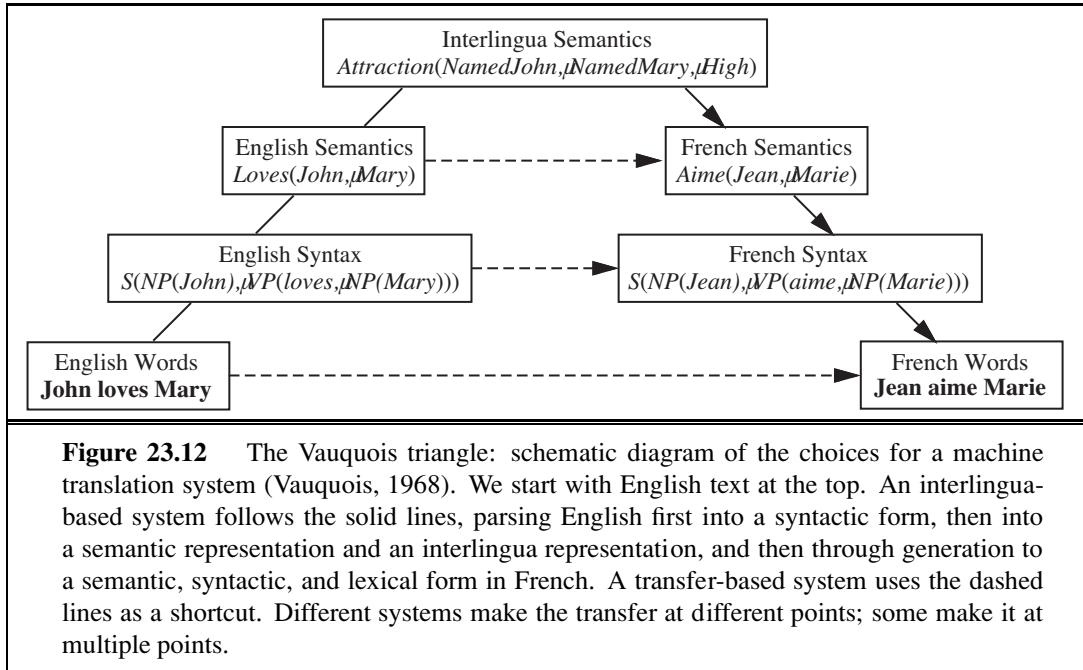


Figure 23.12 The Vauquois triangle: schematic diagram of the choices for a machine translation system (Vauquois, 1968). We start with English text at the top. An interlingua-based system follows the solid lines, parsing English first into a syntactic form, then into a semantic representation and an interlingua 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.

23.4.2 Statistical machine translation

Now that we have seen how complex the translation task can be, it should come as no surprise that the most successful machine translation systems are built by training a probabilistic model using statistics gathered from a large corpus of text. This approach does not need a complex ontology of interlingua concepts, nor does it need handcrafted grammars of the source and target languages, nor a hand-labeled treebank. All it needs is data—sample translations from which a translation model can be learned. To translate a sentence in, say, English (e) into French (f), we find the string of words f^* that maximizes

$$f^* = \underset{f}{\operatorname{argmax}} P(f | e) = \underset{f}{\operatorname{argmax}} P(e | f) P(f).$$

LANGUAGE MODEL
TRANSLATION MODEL

Here the factor $P(f)$ is the target **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 for a given French sentence. Similarly, $P(f|e)$ is a translation model from English to French.

Should we work directly on $P(f | e)$, or apply Bayes' rule and work on $P(e | f) P(f)$? In **diagnostic** applications like medicine, it is easier to model the domain in the causal direction: $P(\text{symptoms} | \text{disease})$ rather than $P(\text{disease} | \text{symptoms})$. But in translation both directions are equally easy. The earliest work in statistical machine translation did apply Bayes' rule—in part because the researchers had a good language model, $P(f)$, and wanted to make use of it, and in part because they came from a background in speech recognition, which *is* a diagnostic problem. We follow their lead in this chapter, but we note that recent work in statistical machine translation often optimizes $P(f | e)$ directly, using a more sophisticated model that takes into account many of the features from the language model.

The language model, $P(f)$, could address any level(s) on the right-hand side of Figure 23.12, but the easiest and most common approach is to build an n -gram model from a French corpus, as we have seen before. This captures only a partial, local idea of French sentences; however, that is often sufficient for rough translation.⁸

BILINGUAL CORPUS

The translation model is learned from a **bilingual corpus**—a collection of parallel texts, each an English/French pair. Now, if we had an infinitely large corpus, then translating a sentence would just be a lookup task: we would have seen the English sentence before in the corpus, so we could just return the paired French sentence. But of course our resources are finite, and most of the sentences we will be asked to translate will be novel. However, they will be composed of **phrases** that we have seen before (even if some phrases are as short as one word). For example, in this book, common phrases include “in this exercise we will,” “size of the state space,” “as a function of the” and “notes at the end of the chapter.” If asked to translate the novel sentence “In this exercise we will compute the size of the state space as a function of the number of actions.” into French, we should be able to break the sentence into phrases, find the phrases in the English corpus (this book), find the corresponding French phrases (from the French translation of the book), and then reassemble the French phrases into an order that makes sense in French. In other words, given a source English sentence, e , finding a French translation f is a matter of three steps:

1. Break the English sentence into phrases e_1, \dots, e_n .
2. For each phrase e_i , choose a corresponding French phrase f_i . We use the notation $P(f_i | e_i)$ for the phrasal probability that f_i is a translation of e_i .
3. Choose a permutation of the phrases f_1, \dots, f_n . We will specify this permutation in a way that seems a little complicated, but is designed to have a simple probability distribution: For each f_i , we choose a **distortion** d_i , which is the number of words that phrase f_i has moved with respect to f_{i-1} ; positive for moving to the right, negative for moving to the left, and zero if f_i immediately follows f_{i-1} .

DISTORTION

Figure 23.13 shows an example of the process. At the top, the sentence “There is a smelly wumpus sleeping in 2 2” is broken into five phrases, e_1, \dots, e_5 . Each of them is translated into a corresponding phrase f_i , and then these are permuted into the order f_1, f_3, f_4, f_2, f_5 . We specify the permutation in terms of the distortions d_i of each French phrase, defined as

$$d_i = \text{START}(f_i) - \text{END}(f_{i-1}) - 1,$$

where $\text{START}(f_i)$ is the ordinal number of the first word of phrase f_i in the French sentence, and $\text{END}(f_{i-1})$ is the ordinal number of the last word of phrase f_{i-1} . In Figure 23.13 we see that f_5 , “à 2 2,” immediately follows f_4 , “qui dort,” and thus $d_5 = 0$. Phrase f_2 , however, has moved one words to the right of f_1 , so $d_2 = 1$. As a special case we have $d_1 = 0$, because f_1 starts at position 1 and $\text{END}(f_0)$ is defined to be 0 (even though f_0 does not exist).

Now that we have defined the distortion, d_i , we can define the probability distribution for distortion, $\mathbf{P}(d_i)$. Note that for sentences bounded by length n we have $|d_i| \leq n$, and

⁸ For the finer points of translation, n -grams are clearly not enough. Marcel Proust’s 4000-page novel *A la recherche du temps perdu* begins and ends with the same word (*longtemps*), so some translators have decided to do the same, thus basing the translation of the final word on one that appeared roughly 2 million words earlier.

so the full probability distribution $\mathbf{P}(d_i)$ has only $2n + 1$ elements, far fewer numbers to learn than the number of permutations, $n!$. That is why we defined the permutation in this circuitous way. Of course, this is a rather impoverished model of distortion. It doesn't say that adjectives are usually distorted to appear after the noun when we are translating from English to French—that fact is represented in the French language model, $P(f)$. The distortion probability is completely independent of the words in the phrases—it depends only on the integer value d_i . The probability distribution provides a summary of the volatility of the permutations; how likely a distortion of $P(d=2)$ is, compared to $P(d=0)$, for example.

We're ready now to put it all together: we can define $P(f, d | e)$, the probability that the sequence of phrases f with distortions d is a translation of the sequence of phrases e . We make the assumption that each phrase translation and each distortion is independent of the others, and thus we can factor the expression as

$$P(f, d | e) = \prod_i P(f_i | e_i) P(d_i)$$

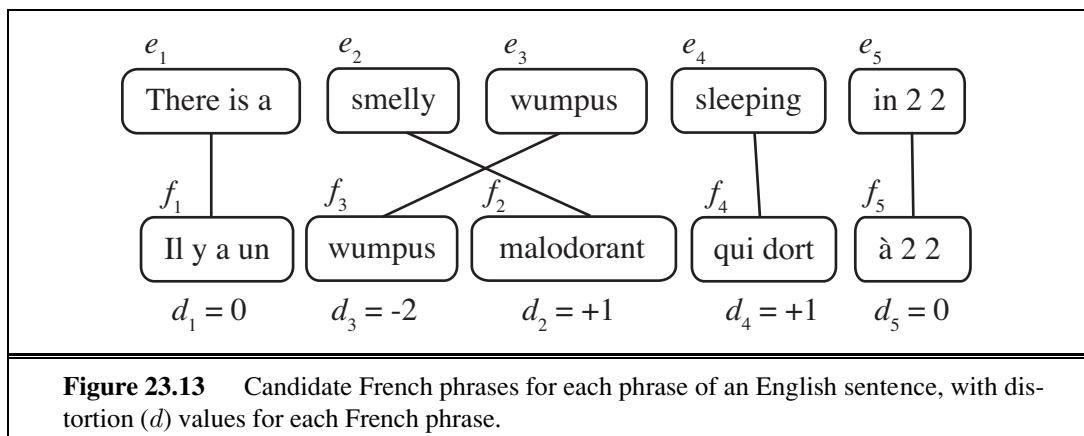


Figure 23.13 Candidate French phrases for each phrase of an English sentence, with distortion (d) values for each French phrase.

That gives us a way to compute the probability $P(f, d | e)$ for a candidate translation f and distortion d . But to find the best f and d we can't just enumerate sentences; with maybe 100 French phrases for each English phrase in the corpus, there are 100^5 different 5-phrase translations, and $5!$ reorderings for each of those. We will have to search for a good solution. A local beam search (see page 125) with a heuristic that estimates probability has proven effective at finding a nearly-most-probable translation.

All that remains is to learn the phrasal and distortion probabilities. We sketch the procedure; see the notes at the end of the chapter for details.

HANSARD

1. **Find parallel texts:** First, gather a parallel bilingual corpus. For example, a **Hansard**⁹ 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. Bilingual text is also available online; some Web sites publish parallel content with parallel URLs, for

⁹ Named after William Hansard, who first published the British parliamentary debates in 1811.

example, /en/ for the English page and /fr/ for the corresponding French page. The leading statistical translation systems train on hundreds of millions of words of parallel text and billions of words of monolingual text.

2. **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. paid \$29.99 on 9.9.09.”; only the final period ends a sentence. One way to decide if a period ends a sentence is to train a model that takes as features the surrounding words and their parts of speech. This approach achieves about 98% accuracy.
3. **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 (i.e. short sentences should align with short sentences), 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 algorithm. Even better alignment can be achieved by using landmarks that are common to both languages, such as numbers, dates, proper names, or words that we know from a bilingual dictionary have an unambiguous translation. For example, if the 3rd English and 4th French sentences contain the string “1989” and neighboring sentences do not, that is good evidence that the sentences should be aligned together.
4. **Align phrases:** Within a sentence, phrases can be aligned by a process that is similar to that used for sentence alignment, but requiring iterative improvement. When we start, we have no way of knowing that “qui dort” aligns with “sleeping,” but we can arrive at that alignment by a process of aggregation of evidence. Over all the example sentences we have seen, we notice that “qui dort” and “sleeping” co-occur with high frequency, and that in the pair of aligned sentences, no phrase other than “qui dort” co-occurs so frequently in other sentences with “sleeping.” A complete phrase alignment over our corpus gives us the phrasal probabilities (after appropriate smoothing).
5. **Extract distortions:** Once we have an alignment of phrases we can define distortion probabilities. Simply count how often distortion occurs in the corpus for each distance $d = 0, \pm 1, \pm 2, \dots$, and apply smoothing.
6. **Improve estimates with EM:** Use expectation–maximization to improve the estimates of $P(f | e)$ and $P(d)$ values. We compute the best alignments with the current values of these parameters in the E step, then update the estimates in the M step and iterate the process until convergence.

23.5 SPEECH RECOGNITION

Speech recognition is the task of identifying a sequence of words uttered by a speaker, given the acoustic signal. It has become one of the mainstream applications of AI—millions of

people interact with speech recognition systems every day to navigate voice mail systems, search the Web from mobile phones, and other applications. Speech is an attractive option when hands-free operation is necessary, as when operating machinery.

SEGMENTATION

Speech recognition is difficult because the sounds made by a speaker are ambiguous and, well, noisy. As a well-known example, the phrase “recognize speech” sounds almost the same as “wreck a nice beach” when spoken quickly. Even this short example shows several of the issues that make speech problematic. First, **segmentation**: written words in English have spaces between them, but in fast speech there are no pauses in “wreck a nice” that would distinguish it as a multiword phrase as opposed to the single word “recognize.” Second, **coarticulation**: when speaking quickly the “s” sound at the end of “nice” merges with the “b” sound at the beginning of “beach,” yielding something that is close to a “sp.” Another problem that does not show up in this example is **homophones**—words like “to,” “too,” and “two” that sound the same but differ in meaning.

COARTICULATION

HOMOPHONES

ACOUSTIC MODEL

LANGUAGE MODEL

NOISY CHANNEL MODEL

We can view speech recognition as a problem in most-likely-sequence explanation. As we saw in Section 15.2, this is the problem of computing the most likely sequence of state variables, $\mathbf{x}_{1:t}$, given a sequence of observations $\mathbf{e}_{1:t}$. In this case the state variables are the words, and the observations are sounds. More precisely, an observation is a vector of features extracted from the audio signal. As usual, the most likely sequence can be computed with the help of Bayes’ rule to be:

$$\underset{\text{word}_{1:t}}{\operatorname{argmax}} P(\text{word}_{1:t} \mid \text{sound}_{1:t}) = \underset{\text{word}_{1:t}}{\operatorname{argmax}} P(\text{sound}_{1:t} \mid \text{word}_{1:t})P(\text{word}_{1:t}).$$

Here $P(\text{sound}_{1:t} \mid \text{word}_{1:t})$ is the **acoustic model**. It describes the sounds of words—that “ceiling” begins with a soft “c” and sounds the same as “sealing.” $P(\text{word}_{1:t})$ is known as the **language model**. It specifies the prior probability of each utterance—for example, that “ceiling fan” is about 500 times more likely as a word sequence than “sealing fan.”

This approach was named the **noisy channel model** by Claude Shannon (1948). He described a situation in which an original message (the *words* in our example) is transmitted over a noisy channel (such as a telephone line) such that a corrupted message (the *sounds* in our example) are received at the other end. Shannon showed that no matter how noisy the channel, it is possible to recover the original message with arbitrarily small error, if we encode the original message in a redundant enough way. The noisy channel approach has been applied to speech recognition, machine translation, spelling correction, and other tasks.

Once we define the acoustic and language models, we can solve for the most likely sequence of words using the Viterbi algorithm (Section 15.2.3 on page 576). Most speech recognition systems use a language model that makes the Markov assumption—that the current state $Word_t$ depends only on a fixed number n of previous states—and represent $Word_t$ as a single random variable taking on a finite set of values, which makes it a Hidden Markov Model (HMM). Thus, speech recognition becomes a simple application of the HMM methodology, as described in Section 15.3—simple that is, once we define the acoustic and language models. We cover them next.

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

Figure 23.14 The ARPA phonetic alphabet, or ARPAbet, listing all the phones used in American English. There are several alternative notations, including an International Phonetic Alphabet (IPA), which contains the phones in all known languages.

23.5.1 Acoustic model

Sound waves are periodic changes in pressure that propagate through the air. When these waves strike the diaphragm of a microphone, the back-and-forth movement generates an electric current. An analog-to-digital converter measures the size of the current—which approximates the amplitude of the sound wave—at discrete intervals called the **sampling rate**. Speech sounds, which are mostly in the range of 100 Hz (100 cycles per second) to 1000 Hz, are typically sampled at a rate of 8 kHz. (CDs and mp3 files are sampled at 44.1 kHz.) The precision of each measurement is determined by the **quantization factor**; speech recognizers typically keep 8 to 12 bits. That means that a low-end system, sampling at 8 kHz with 8-bit quantization, would require nearly half a megabyte per minute of speech.

Since we only want to know what words were spoken, not exactly what they sounded like, we don't need to keep all that information. We only need to distinguish between different speech sounds. Linguists have identified about 100 speech sounds, or **phones**, that can be composed to form all the words in all known human languages. Roughly speaking, a phone is the sound that corresponds to a single vowel or consonant, but there are some complications: combinations of letters, such as “th” and “ng” produce single phones, and some letters produce different phones in different contexts (e.g., the “a” in *rat* and *rate*). Figure 23.14 lists

SAMPLING RATE

QUANTIZATION FACTOR

PHONE

PHONEME

all the phones that are used in English, with an example of each. A **phoneme** is the smallest unit of sound that has a distinct meaning to speakers of a particular language. For example, the “t” in “stick” sounds similar enough to the “t” in “tick” that speakers of English consider them the same phoneme. But the difference is significant in the Thai language, so there they are two phonemes. To represent spoken English we want a representation that can distinguish between different phonemes, but one that need not distinguish the nonphonemic variations in sound: loud or soft, fast or slow, male or female voice, etc.

FRAME

First, we observe that although the sound frequencies in speech may be several kHz, the *changes* in the content of the signal occur much less often, perhaps at no more than 100 Hz. Therefore, speech systems summarize the properties of the signal over time slices called **frames**. A frame length of about 10 milliseconds (i.e., 80 samples at 8 kHz) is short enough to ensure that few short-duration phenomena will be missed. Overlapping frames are used to make sure that we don’t miss a signal because it happens to fall on a frame boundary.

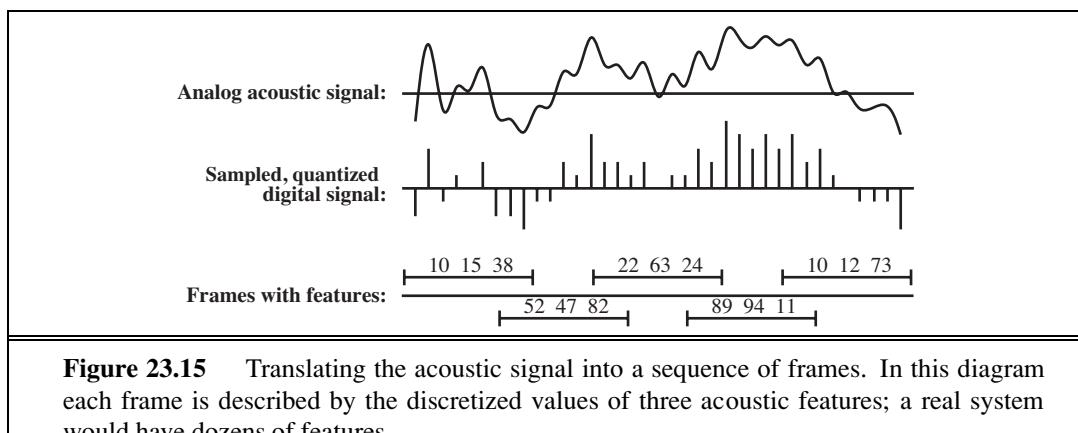
FEATURE

Each frame is summarized by a vector of **features**. Picking out features from a speech signal is like listening to an orchestra and saying “here the French horns are playing loudly and the violins are playing softly.” We’ll give a brief overview of the features in a typical system. First, a Fourier transform is used to determine the amount of acoustic energy at about a dozen frequencies. Then we compute a measure called the **mel frequency cepstral coefficient (MFCC)** or MFCC for each frequency. We also compute the total energy in the frame. That gives thirteen features; for each one we compute the difference between this frame and the previous frame, and the difference between differences, for a total of 39 features. These are continuous-valued; the easiest way to fit them into the HMM framework is to discretize the values. (It is also possible to extend the HMM model to handle continuous mixtures of Gaussians.) Figure 23.15 shows the sequence of transformations from the raw sound to a sequence of frames with discrete features.

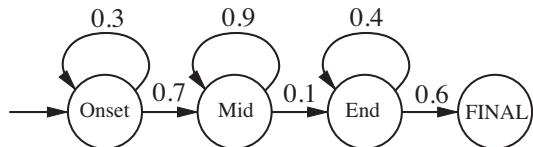
MEL FREQUENCY
CEPSTRAL
COEFFICIENT (MFCC)

PHONE MODEL

We have seen how to go from the raw acoustic signal to a series of observations, \mathbf{e}_t . Now we have to describe the (unobservable) states of the HMM and define the transition model, $\mathbf{P}(\mathbf{X}_t | \mathbf{X}_{t-1})$, and the sensor model, $\mathbf{P}(\mathbf{E}_t | \mathbf{X}_t)$. The transition model can be broken into two levels: word and phone. We’ll start from the bottom: the **phone model** describes



Phone HMM for [m]:

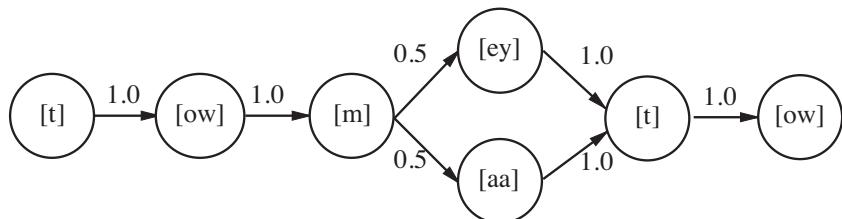


Output probabilities for the phone HMM:

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

Figure 23.16 An HMM for the three-state phone [m]. Each state has several possible outputs, each with its own probability. The MFCC feature labels C_1 through C_7 are arbitrary, standing for some combination of feature values.

(a) Word model with dialect variation:



(b) Word model with coarticulation and dialect variations

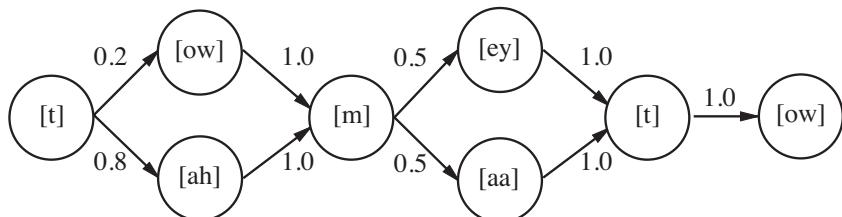


Figure 23.17 Two pronunciation models of the word “tomato.” Each model is shown as a transition diagram with states as circles and arrows showing allowed transitions with their associated probabilities. (a) A model allowing for dialect differences. The 0.5 numbers are estimates based on the two authors’ preferred pronunciations. (b) A model with a coarticulation effect on the first vowel, allowing either the [ow] or the [ah] phone.

a phone as three states, the onset, middle, and end. For example, the [t] phone has a silent beginning, a small explosive burst of sound in the middle, and (usually) a hissing at the end. Figure 23.16 shows an example for the phone [m]. Note that in normal speech, an average phone has a duration of 50–100 milliseconds, or 5–10 frames. The self-loops in each state allows for variation in this duration. By taking many self-loops (especially in the mid state), we can represent a long “mmmmmmmmmmmm” sound. Bypassing the self-loops yields a short “m” sound.

PRONUNCIATION
MODEL

In Figure 23.17 the phone models are strung together to form a **pronunciation model** for a word. According to Gershwin (1937), you say [t oh m ey t oh] and I say [t oh m aa t oh]. Figure 23.17(a) shows a transition model that provides for this dialect variation. Each of the circles in this diagram represents a phone model like the one in Figure 23.16.

In addition to dialect variation, words can have **coarticulation** variation. For example, the [t] phone is produced with the tongue at the top of the mouth, whereas the [ow] has the tongue near the bottom. When speaking quickly, the tongue doesn’t have time to get into position for the [ow], and we end up with [t ah] rather than [t ow]. Figure 23.17(b) gives a model for “tomato” that takes this coarticulation effect into account. More sophisticated phone models take into account the context of the surrounding phones.

There can be substantial variation in pronunciation for a word. The most common pronunciation of “because” is [b iy k ah z], but that only accounts for about a quarter of uses. Another quarter (approximately) substitutes [ix], [ih] or [ax] for the first vowel, and the remainder substitute [ax] or [aa] for the second vowel, [zh] or [s] for the final [z], or drop “be” entirely, leaving “cuz.”

23.5.2 Language model

For general-purpose speech recognition, the language model can be an n -gram model of text learned from a corpus of written sentences. However, spoken language has different characteristics than written language, so it is better to get a corpus of transcripts of spoken language. For task-specific speech recognition, the corpus should be task-specific: to build your airline reservation system, get transcripts of prior calls. It also helps to have task-specific vocabulary, such as a list of all the airports and cities served, and all the flight numbers.

Part of the design of a voice user interface is to coerce the user into saying things from a limited set of options, so that the speech recognizer will have a tighter probability distribution to deal with. For example, asking “What city do you want to go to?” elicits a response with a highly constrained language model, while asking “How can I help you?” does not.

23.5.3 Building a speech recognizer

The quality of a speech recognition system depends on the quality of all of its components—the language model, the word-pronunciation models, the phone models, and the signal-processing algorithms used to extract spectral features from the acoustic signal. We have discussed how the language model can be constructed from a corpus of written text, and we leave the details of signal processing to other textbooks. We are left with the pronunciation and phone models. The *structure* of the pronunciation models—such as the tomato models in

Figure 23.17—is usually developed by hand. Large pronunciation dictionaries are now available for English and other languages, although their accuracy varies greatly. The structure of the three-state phone models is the same for all phones, as shown in Figure 23.16. That leaves the probabilities themselves.

As usual, we will acquire the probabilities from a corpus, this time a corpus of speech. The most common type of corpus to obtain is one that includes the speech signal for each sentence paired with a transcript of the words. Building a model from this corpus is more difficult than building an n -gram model of text, because we have to build a hidden Markov model—the phone sequence for each word and the phone state for each time frame are hidden variables. In the early days of speech recognition, the hidden variables were provided by laborious hand-labeling of spectrograms. Recent systems use expectation–maximization to automatically supply the missing data. The idea is simple: given an HMM and an observation sequence, we can use the smoothing algorithms from Sections 15.2 and 15.3 to compute the probability of each state at each time step and, by a simple extension, the probability of each state–state pair at consecutive time steps. These probabilities can be viewed as *uncertain labels*. From the uncertain labels, we can estimate new transition and sensor probabilities, and the EM procedure repeats. The method is guaranteed to increase the fit between model and data on each iteration, and it generally converges to a much better set of parameter values than those provided by the initial, hand-labeled estimates.

The systems with the highest accuracy work by training a different model for each speaker, thereby capturing differences in dialect as well as male/female and other variations. This training can require several hours of interaction with the speaker, so the systems with the most widespread adoption do not create speaker-specific models.

The accuracy of a system depends on a number of factors. First, the quality of the signal matters: a high-quality directional microphone aimed at a stationary mouth in a padded room will do much better than a cheap microphone transmitting a signal over phone lines from a car in traffic with the radio playing. The vocabulary size matters: when recognizing digit strings with a vocabulary of 11 words (1-9 plus “oh” and “zero”), the word error rate will be below 0.5%, whereas it rises to about 10% on news stories with a 20,000-word vocabulary, and 20% on a corpus with a 64,000-word vocabulary. The task matters too: when the system is trying to accomplish a specific task—book a flight or give directions to a restaurant—the task can often be accomplished perfectly even with a word error rate of 10% or more.

23.6 SUMMARY

Natural language understanding is one of the most important subfields of AI. Unlike most other areas of AI, natural language understanding requires an empirical investigation of actual human behavior—which turns out to be complex and interesting.

- Formal language theory and **phrase structure** grammars (and in particular, **context-free grammar**) are useful tools for dealing with some aspects of natural language. The probabilistic context-free grammar (PCFG) formalism is widely used.

- Sentences in a context-free language can be parsed in $O(n^3)$ time by a **chart parser** such as the **CYK algorithm**, which requires grammar rules to be in **Chomsky Normal Form**.
- A **treebank** can be used to learn a grammar. It is also possible to learn a grammar from an unparsed corpus of sentences, but this is less successful.
- A **lexicalized PCFG** allows us to represent that some relationships between words are more common than others.
- 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.
- **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.
- **Machine translation** systems have been implemented using a range of techniques, from full syntactic and semantic analysis to statistical techniques based on phrase frequencies. Currently the statistical models are most popular and most successful.
- **Speech recognition** systems are also primarily based on statistical principles. Speech systems are popular and useful, albeit imperfect.
- Together, machine translation and speech recognition are two of the big successes of natural language technology. One reason that the models perform well is that large corpora are available—both translation and speech are tasks that are performed “in the wild” by people every day. In contrast, tasks like parsing sentences have been less successful, in part because no large corpora of parsed sentences are available “in the wild” and in part because parsing is not useful in and of itself.

BIBLIOGRAPHICAL AND HISTORICAL NOTES

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, ca. 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. Peter Naur 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 Shieber (1987).

Probabilistic context-free grammars were investigated by Booth (1969) and Salo-maa (1969). Other algorithms for PCFGs are presented in the excellent short monograph by

Charniak (1993) and the excellent long textbooks by Manning and Schütze (1999) and Jurafsky and Martin (2008). Baker (1979) introduces the inside–outside algorithm for learning a PCFG, and Lari and Young (1990) describe its uses and limitations. Stolcke and Omohundro (1994) show how to learn grammar rules with Bayesian model merging; Haghghi and Klein (2006) describe a learning system based on prototypes.

Lexicalized PCFGs (Charniak, 1997; Hwa, 1998) combine the best aspects of PCFGs and n -gram models. Collins (1999) describes PCFG parsing that is lexicalized with head features. Petrov and Klein (2007a) show how to get the advantages of lexicalization without actual lexical augmentations by learning specific syntactic categories from a treebank that has general categories; for example, the treebank has the category NP , from which more specific categories such as NP_O and NP_S can be learned.

There have been many attempts to write formal grammars of natural languages, both in “pure” linguistics and in computational linguistics. There are several comprehensive but informal grammars of English (Quirk *et al.*, 1985; McCawley, 1988; Huddleston and Pullum, 2002). 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** (Clark and Curran, 2004), in which there can be as few as two grammar rules, or with **dependency grammar** (Smith and Eisner, 2008; Kübler *et al.*, 2009) in which there are no syntactic categories, only relations between words. Sleator and Temperley (1993) describe a dependency parser. Paskin (2001) shows that a version of dependency grammar is easier to learn than PCFGs.

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; Earley, 1970; Graham *et al.*, 1980). 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. Klein and Manning (2003) describe A* parsing, and Pauls and Klein (2009) extend that to K -best A* parsing, in which the result is not a single parse but the K best.

Leading parsers today include those by Petrov and Klein (2007b), which achieved 90.6% accuracy on the Wall Street Journal corpus, Charniak and Johnson (2005), which achieved 92.0%, and Koo *et al.* (2008), which achieved 93.2% on the Penn treebank. These numbers are not directly comparable, and there is some criticism of the field that it is focusing too narrowly on a few select corpora, and perhaps overfitting on them.

Formal semantic interpretation of natural languages originates within philosophy and formal logic, particularly Alfred Tarski’s (1935) work on the semantics of formal languages. Bar-Hillel (1954) 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. Richard Montague’s essay “English as a formal language” (1970) is a kind of manifesto for the logical analysis of language, but the books by Dowty *et al.* (1991) and Portner and Partee (2002) are more readable.

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 from the moon by the Apollo program. Roger Schank and his students built a series of programs (Schank and Abelson, 1977; Schank and Riesbeck, 1981) that all had the task of understanding language. Modern approaches to semantic interpretation usually assume that the mapping from syntax to semantics will be learned from examples (Zelle and Mooney, 1996; Zettlemoyer and Collins, 2005).

Hobbs *et al.* (1993) describes a quantitative nonprobabilistic framework for interpretation. More recent work follows an explicitly probabilistic framework (Charniak and Goldman, 1992; Wu, 1993; Franz, 1996). In linguistics, optimality theory (Kager, 1999) is based on the idea of building soft constraints into the grammar, giving a natural ranking to interpretations (similar to a probability distribution), 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. Martin (1990) and Gibbs (2006) offer computational models of metaphor interpretation.

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. Prominent linguists, such as Chomsky (1957) and Pinker (2003), 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 says that children aren't given enough input to learn a CFG, so they must already "know" the grammar and be merely tuning some of its parameters. While this argument continues to hold sway throughout much of Chomskyan linguistics, it has been dismissed by some other linguists (Pullum, 1996; Elman *et al.*, 1997) and most computer scientists. As early as 1969, Horning showed that it is possible to learn, in the sense of PAC learning, a *probabilistic* context-free grammar. Since then, there have been many convincing empirical demonstrations of learning from positive examples alone, such as the ILP work of Mooney (1999) and Muggleton and De Raedt (1994), the sequence learning of Nevill-Manning and Witten (1997), and the remarkable Ph.D. theses of Schütze (1995) and de Marcken (1996). There is an annual International Conference on Grammatical Inference (ICGI). It is possible to learn other grammar formalisms, such as regular languages (Denis, 2001) and finite state automata (Parekh and Honavar, 2001). Abney (2007) is a textbook introduction to semi-supervised learning for language models.

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 Penn Treebank (Marcus *et al.*, 1993) provides parse trees for a 3-million-word corpus of English. Charniak (1996) and Klein and Manning (2001) discuss parsing with treebank grammars. The British National Corpus (Leech *et al.*, 2001) contains 100 million words, and the World Wide Web contains several trillion words; (Brants *et al.*, 2007) describe *n*-gram models over a 2-trillion-word Web corpus.

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 that 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) describes the enthusiasm of this period. However, the U.S. government subsequently reported (ALPAC, 1966) that “there is no immediate or predictable prospect of useful machine translation.” However, limited work continued, and starting in the 1980s, computer power had increased to the point where the ALPAC findings were no longer correct.

The basic statistical approach we describe in the chapter is based on early work by the IBM group (Brown *et al.*, 1988, 1993) and the recent work by the ISI and Google research groups (Och and Ney, 2004; Zollmann *et al.*, 2008). A textbook introduction on statistical machine translation is given by Koehn (2009), and a short tutorial by Kevin Knight (1999) has been influential. Early work on sentence segmentation was done by Palmer and Hearst (1994). Och and Ney (2003) and Moore (2005) cover bilingual sentence alignment.

The prehistory of speech recognition began in the 1920s with Radio Rex, a voice-activated toy dog. Rex jumped out of his doghouse in response to the word “Rex!” (or actually almost any sufficiently loud word). Somewhat more serious work began after World War II. At AT&T Bell Labs, a system was built for recognizing isolated digits (Davis *et al.*, 1952) by means of simple pattern matching of acoustic features. Starting in 1971, the Defense Advanced Research Projects Agency (DARPA) of the United States Department of Defense funded four competing five-year projects to develop high-performance speech recognition systems. The winner, and the only system to meet the goal of 90% accuracy with a 1000-word vocabulary, was the HARPY system at CMU (Lowerre and Reddy, 1980). The final version of HARPY was derived from a system called DRAGON built by CMU graduate student James Baker (1975); DRAGON was the first to use HMMs for speech. Almost simultaneously, Jelinek (1976) at IBM had developed another HMM-based system. Recent years have been characterized by steady incremental progress, larger data sets and models, and more rigorous competitions on more realistic speech tasks. In 1997, Bill Gates predicted, “The PC five years from now—you won’t recognize it, because speech will come into the interface.” That didn’t quite happen, but in 2008 he predicted “In five years, Microsoft expects more Internet searches to be done through speech than through typing on a keyboard.” History will tell if he is right this time around.

Several good textbooks on speech recognition are available (Rabiner and Juang, 1993; Jelinek, 1997; Gold and Morgan, 2000; Huang *et al.*, 2001). The presentation in this chapter drew on the survey by Kay, Gawron, and Norvig (1994) and on the textbook by Jurafsky and Martin (2008). Speech recognition research is published in *Computer Speech and Language*, *Speech Communications*, and the *IEEE Transactions on Acoustics, Speech, and Signal Processing* and at the DARPA Workshops on Speech and Natural Language Processing and the Eurospeech, ICSLP, and ASRU conferences.

Ken Church (2004) shows that natural language research has cycled between concentrating on the data (empiricism) and concentrating on theories (rationalism). The linguist John Firth (1957) proclaimed “You shall know a word by the company it keeps,” and linguistics of the 1940s and early 1950s was based largely on word frequencies, although without the computational power we have available today. Then Noam (Chomsky, 1956) showed the limitations of finite-state models, and sparked an interest in theoretical studies of syntax, disregarding frequency counts. This approach dominated for twenty years, until empiricism made a comeback based on the success of work in statistical speech recognition (Jelinek, 1976). Today, most work accepts the statistical framework, but there is great interest in building statistical models that consider higher-level models, such as syntactic trees and semantic relations, not just sequences of words.

Work on applications of 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*. A broad range of NLP work appears in the journal *Computational Linguistics* and its conference, ACL, and in the Computational Linguistics (COLING) conference.

EXERCISES

23.1 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.

23.2 An *HMM grammar* is essentially a standard HMM whose state variable is N (nonterminal, with values such as *Det*, *Adjective*, *Noun* and so on) and whose evidence variable is W (word, with values such as *is*, *duck*, and so on). The HMM model includes a prior $\mathbf{P}(N_0)$, a transition model $\mathbf{P}(N_{t+1}|N_t)$, and a sensor model $\mathbf{P}(W_t|N_t)$. Show that every HMM grammar can be written as a PCFG. [Hint: start by thinking about how the HMM prior can be represented by PCFG rules for the sentence symbol. You may find it helpful to illustrate for the particular HMM with values A , B for N and values x , y for W .]

23.3 Consider the following PCFG for simple verb phrases:

0.1 : $VP \rightarrow Verb$
 0.2 : $VP \rightarrow Copula\ Adjective$
 0.5 : $VP \rightarrow Verb\ the\ Noun$
 0.2 : $VP \rightarrow VP\ Adverb$
 0.5 : $Verb \rightarrow is$
 0.5 : $Verb \rightarrow shoots$
 0.8 : $Copula \rightarrow is$
 0.2 : $Copula \rightarrow seems$
 0.5 : $Adjective \rightarrow unwell$
 0.5 : $Adjective \rightarrow well$
 0.5 : $Adverb \rightarrow well$
 0.5 : $Adverb \rightarrow badly$
 0.6 : $Noun \rightarrow duck$
 0.4 : $Noun \rightarrow well$

- a. Which of the following have a nonzero probability as a VP? (i) shoots the duck well well well (ii) seems the well well (iii) shoots the unwell well badly
- b. What is the probability of generating “is well well”?
- c. What types of ambiguity are exhibited by the phrase in (b)?
- d. Given any PCFG, is it possible to calculate the probability that the PCFG generates a string of exactly 10 words?

23.4 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.

23.5 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.

23.6 Consider the sentence “Someone walked slowly to the supermarket” and a lexicon consisting of the following words:

<i>Pronoun</i> \rightarrow someone	<i>Verb</i> \rightarrow walked
<i>Adv</i> \rightarrow slowly	<i>Prep</i> \rightarrow to
<i>Article</i> \rightarrow the	<i>Noun</i> \rightarrow supermarket

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

(A):	(B):	(C):
$S \rightarrow NP\ VP$	$S \rightarrow NP\ VP$	$S \rightarrow NP\ VP$
$NP \rightarrow Pronoun$	$NP \rightarrow Pronoun$	$NP \rightarrow Pronoun$
$NP \rightarrow Article\ Noun$	$NP \rightarrow Noun$	$NP \rightarrow Article\ NP$
$VP \rightarrow VP\ PP$	$NP \rightarrow Article\ NP$	$VP \rightarrow Verb\ Adv$
$VP \rightarrow VP\ Adv\ Adv$	$VP \rightarrow Verb\ Vmod$	$Adv \rightarrow Adv\ Adv$
$VP \rightarrow Verb$	$Vmod \rightarrow Adv\ Vmod$	$Adv \rightarrow PP$
$PP \rightarrow Prep\ NP$	$Vmod \rightarrow Adv$	$PP \rightarrow Prep\ NP$
$NP \rightarrow Noun$	$Adv \rightarrow PP$	$NP \rightarrow Noun$
	$PP \rightarrow Prep\ NP$	

For each of the preceding three grammars, write down three sentences of English and three sentences of non-English generated by the grammar. Each sentence should be significantly different, should be at least six words long, and should include some new lexical entries (which you should define). Suggest ways to improve each grammar to avoid generating the non-English sentences.

23.7 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.

23.8 In this exercise you will transform \mathcal{E}_0 into Chomsky Normal Form (CNF). There are five steps: (a) Add a new start symbol, (b) Eliminate ϵ rules, (c) Eliminate multiple words on right-hand sides, (d) Eliminate rules of the form $(X \rightarrow Y)$, (e) Convert long right-hand sides into binary rules.

- The start symbol, S , can occur only on the left-hand side in CNF. Add a new rule of the form $S' \rightarrow S$, using a new symbol S' .
- The empty string, ϵ cannot appear on the right-hand side in CNF. \mathcal{E}_0 does not have any rules with ϵ , so this is not an issue.
- A word can appear on the right-hand side in a rule only of the form $(X \rightarrow word)$. Replace each rule of the form $(X \rightarrow \dots word \dots)$ with $(X \rightarrow \dots W' \dots)$ and $(W' \rightarrow word)$, using a new symbol W' .
- A rule $(X \rightarrow Y)$ is not allowed in CNF; it must be $(X \rightarrow Y Z)$ or $(X \rightarrow word)$. Replace each rule of the form $(X \rightarrow Y)$ with a set of rules of the form $(X \rightarrow \dots)$, one for each rule $(Y \rightarrow \dots)$, where (\dots) indicates one or more symbols.
- Replace each rule of the form $(X \rightarrow Y Z \dots)$ with two rules, $(X \rightarrow Y Z')$ and $(Z' \rightarrow Z \dots)$, where Z' is a new symbol.

Show each step of the process and the final set of rules.

23.9 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 ungrammatical sentences such as “I smells the wumpus.”

23.10 Consider the following PCFG:

$$\begin{aligned} S &\rightarrow NP\ VP\ [1.0] \\ NP &\rightarrow Noun\ [0.6]\mid Pronoun\ [0.4] \\ VP &\rightarrow Verb\ NP\ [0.8]\mid Modal\ Verb\ [0.2] \\ \\ Noun &\rightarrow \mathbf{can}\ [0.1]\mid \mathbf{fish}\ [0.3]\mid \dots \\ Pronoun &\rightarrow \mathbf{I}\ [0.4]\mid \dots \\ Verb &\rightarrow \mathbf{can}\ [0.01]\mid \mathbf{fish}\ [0.1]\mid \dots \\ Modal &\rightarrow \mathbf{can}\ [0.3]\mid \dots \end{aligned}$$

The sentence “I can fish” has two parse trees with this grammar. Show the two trees, their prior probabilities, and their conditional probabilities, given the sentence.

23.11 An augmented context-free grammar can represent languages that a regular context-free grammar cannot. Show an augmented context-free grammar for the language $a^n b^n c^n$. The allowable values for augmentation variables are 1 and $\text{SUCCESSOR}(n)$, where n is a value. The rule for a sentence in this language is

$$S(n) \rightarrow A(n)\ B(n)\ C(n).$$

Show the rule(s) for each of A , B , and C .

23.12 Augment the \mathcal{E}_1 grammar so that it handles article–noun agreement. That is, make sure that “agents” and “an agent” are NPs , but “agent” and “an agents” are not.

23.13 Consider the following sentence (from *The New York Times*, July 28, 2008):

Banks struggling to recover from multibillion-dollar loans on real estate are curtailing loans to American businesses, depriving even healthy companies of money for expansion and hiring.

- a. Which of the words in this sentence are lexically ambiguous?
- b. Find two cases of syntactic ambiguity in this sentence (there are more than two.)
- c. Give an instance of metaphor in this sentence.
- d. Can you find semantic ambiguity?

23.14 Without looking back at Exercise 23.1, 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 things or too many? Why?

23.15 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? If you know a foreign language, look at the translation of one paragraph into that language. Count and describe the errors made, and conjecture why these errors were made.

23.16 The D_i values for the sentence in Figure 23.13 sum to 0. Will that be true of every translation pair? Prove it or give a counterexample.

23.17 (Adapted from Knight (1999).) Our translation model assumes that, after the phrase translation model selects phrases and the distortion model permutes them, the language model can unscramble the permutation. This exercise investigates how sensible that assumption is. Try to unscramble these proposed lists of phrases into the correct order:

- a. have, programming, a, seen, never, I, language, better
- b. loves, john, mary
- c. is the, communication, exchange of, intentional, information brought, by, about, the production, perception of, and signs, from, drawn, a, of, system, signs, conventional, shared
- d. created, that, we hold these, to be, all men, truths, are, equal, self-evident

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.

23.18 Calculate the most probable path through the HMM in Figure 23.16 for the output sequence $[C_1, C_2, C_3, C_4, C_4, C_6, C_7]$. Also give its probability.

23.19 We forgot to mention that the text in Exercise 23.1 is entitled “Washing Clothes.” Reread the text and answer the questions in Exercise 23.14. Did you do better this time? Bransford and Johnson (1973) used this text in a controlled experiment and found that the title helped significantly. What does this tell you about how language and memory works?

24 PERCEPTION

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

PERCEPTION

SENSOR

OBJECT MODEL

RENDERING MODEL

Perception provides agents with information about the world they inhabit by interpreting the response of **sensors**. A sensor measures some aspect of the environment in a form that can be used as input by an agent program. The sensor could be as simple as a switch, which gives one bit telling whether it is on or off, or as complex as the eye. A variety of sensory modalities are available to artificial agents. Those they share with humans include vision, hearing, and touch. Modalities that are not available to the unaided human include 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. Rather than trying to cover all of these, this chapter will cover one modality in depth: vision.

We saw in our description of POMDPs (Section 17.4, page 658) that a model-based decision-theoretic agent in a partially observable environment has a **sensor model**—a probability distribution $\mathbf{P}(E | S)$ over the evidence that its sensors provide, given a state of the world. Bayes’ rule can then be used to update the estimation of the state.

For vision, the sensor model can be broken into two components: An **object model** describes the objects that inhabit the visual world—people, buildings, trees, cars, etc. The object model could include a precise 3D geometric model taken from a computer-aided design (CAD) system, or it could be vague constraints, such as the fact that human eyes are usually 5 to 7 cm apart. A **rendering model** describes the physical, geometric, and statistical processes that produce the stimulus from the world. Rendering models are quite accurate, but they are ambiguous. For example, a white object under low light may appear as the same color as a black object under intense light. A small nearby object may look the same as a large distant object. Without additional evidence, we cannot tell if the image that fills the frame is a toy Godzilla or a real monster.

Ambiguity can be managed with prior knowledge—we know Godzilla is not real, so the image must be a toy—or by selectively choosing to ignore the ambiguity. For example, the vision system for an autonomous car may not be able to interpret objects that are far in the distance, but the agent can choose to ignore the problem, because it is unlikely to crash into an object that is miles away.

A decision-theoretic agent is not the only architecture that can make use of vision sensors. For example, fruit flies (*Drosophila*) are in part reflex agents: they have cervical giant fibers that form a direct pathway from their visual system to the wing muscles that initiate an escape response—an immediate reaction, without deliberation. Flies and many other flying animals make use of a closed-loop control architecture to land on an object. The visual system extracts an estimate of the distance to the object, and the control system adjusts the wing muscles accordingly, allowing very fast changes of direction, with no need for a detailed model of the object.

Compared to the data from other sensors (such as the single bit that tells the vacuum robot that it has bumped into a wall), visual observations are extraordinarily rich, both in the detail they can reveal and in the sheer amount of data they produce. A video camera for robotic applications might produce a million 24-bit pixels at 60 Hz; a rate of 10 GB per minute. The problem for a vision-capable agent then is: *Which aspects of the rich visual stimulus should be considered to help the agent make good action choices, and which aspects should be ignored?* Vision—and all perception—serves to further the agent’s goals, not as an end to itself.



FEATURE EXTRACTION

RECOGNITION

RECONSTRUCTION

We can characterize three broad approaches to the problem. The **feature extraction** approach, as exhibited by *Drosophila*, emphasizes simple computations applied directly to the sensor observations. In the **recognition** approach an agent draws distinctions among the objects it encounters based on visual and other information. Recognition could mean labeling each image with a yes or no as to whether it contains food that we should forage, or contains Grandma’s face. Finally, in the **reconstruction** approach an agent builds a geometric model of the world from an image or a set of images.

The last thirty years of research have produced powerful tools and methods for addressing these approaches. Understanding these methods requires an understanding of the processes by which images are formed. Therefore, we now cover the physical and statistical phenomena that occur in the production of an image.

24.1 IMAGE FORMATION

Imaging distorts the appearance of objects. For example, a picture taken looking down a long straight set of railway tracks will suggest that the rails converge and meet. As another example, if you hold your hand in front of your eye, you can block out the moon, which is not smaller than your hand. As you move your hand back and forth or tilt it, your hand will seem to shrink and grow *in the image*, but it is not doing so in reality (Figure 24.1). Models of these effects are essential for both recognition and reconstruction.

24.1.1 Images without lenses: The pinhole camera

SCENE
IMAGE

Image sensors gather light scattered from objects in a **scene** and create a two-dimensional **image**. In the eye, the image is formed on the retina, which consists of two types of cells: about 100 million rods, which are sensitive to light at a wide range of wavelengths, and 5

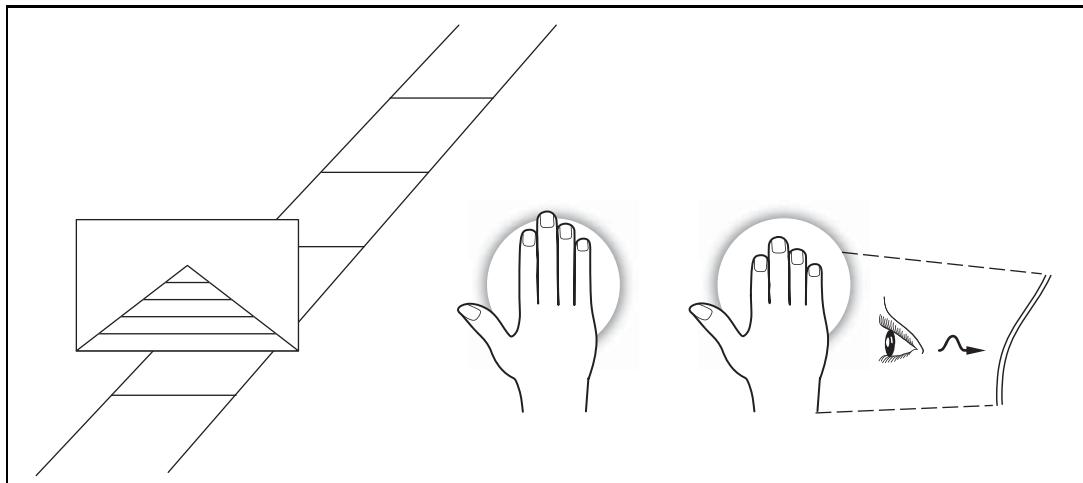


Figure 24.1 Imaging distorts geometry. Parallel lines appear to meet in the distance, as in the image of the railway tracks on the left. In the center, a small hand blocks out most of a large moon. On the right is a foreshortening effect: the hand is tilted away from the eye, making it appear shorter than in the center figure.

PIXEL

PINHOLE CAMERA

million cones. Cones, which are essential for color vision, are of three main types, each of which is sensitive to a different set of wavelengths. In cameras, the image is formed on an image plane, which can be a piece of film coated with silver halides or a rectangular grid of a few million photosensitive **pixels**, each a complementary metal-oxide semiconductor (CMOS) or charge-coupled device (CCD). Each photon arriving at the sensor produces an effect, whose strength depends on the wavelength of the photon. The output of the sensor is the sum of all effects due to photons observed in some time window, meaning that image sensors report a weighted average of the intensity of light arriving at the sensor.

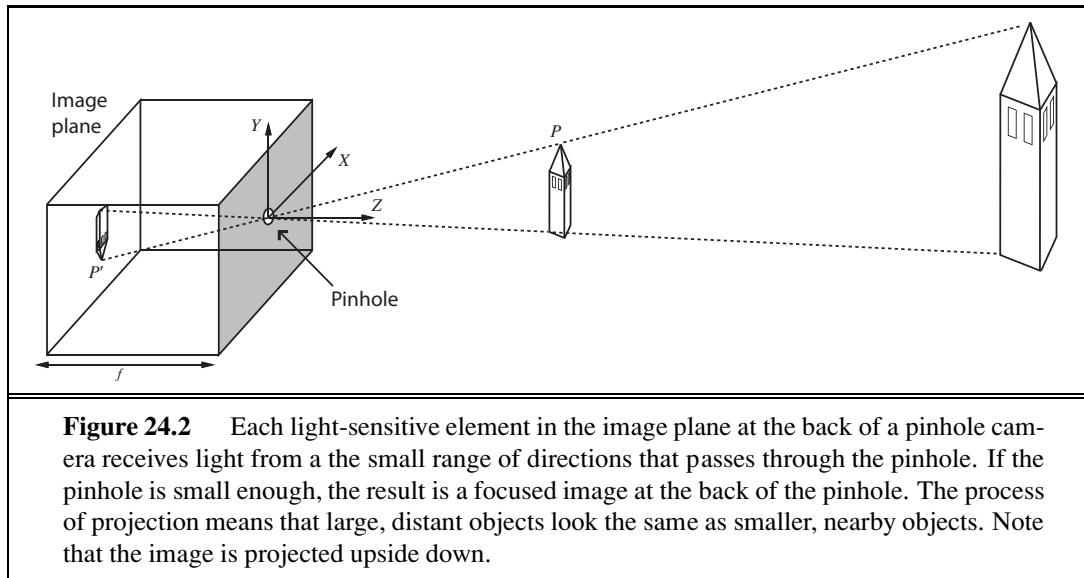
To see a focused image, we must ensure that all the photons from approximately the same spot in the scene arrive at approximately the same point in the image plane. The simplest way to form a focused image is to view stationary objects with a **pinhole camera**, which consists of a pinhole opening, O , at the front of a box, and an image plane at the back of the box (Figure 24.2). Photons from the scene must pass through the pinhole, so if it is small enough then nearby photons in the scene will be nearby in the image plane, and the image will be in focus.

The geometry of scene and image is easiest to understand with the pinhole camera. We use a three-dimensional coordinate system with the origin at the pinhole, and 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:

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

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, distant objects look small. This is what allows you to cover the moon with your hand (Figure 24.1). An important result of this effect is that parallel lines converge to a point on the horizon. (Think of railway tracks, Figure 24.1.) A line in the scene in the direction (U, V, W) and passing through the point (X_0, Y_0, Z_0) 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 $+\infty$. Different choices of (X_0, Y_0, Z_0) yield different lines parallel to one another. The projection of a point P_λ from this line onto the image plane is given by

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

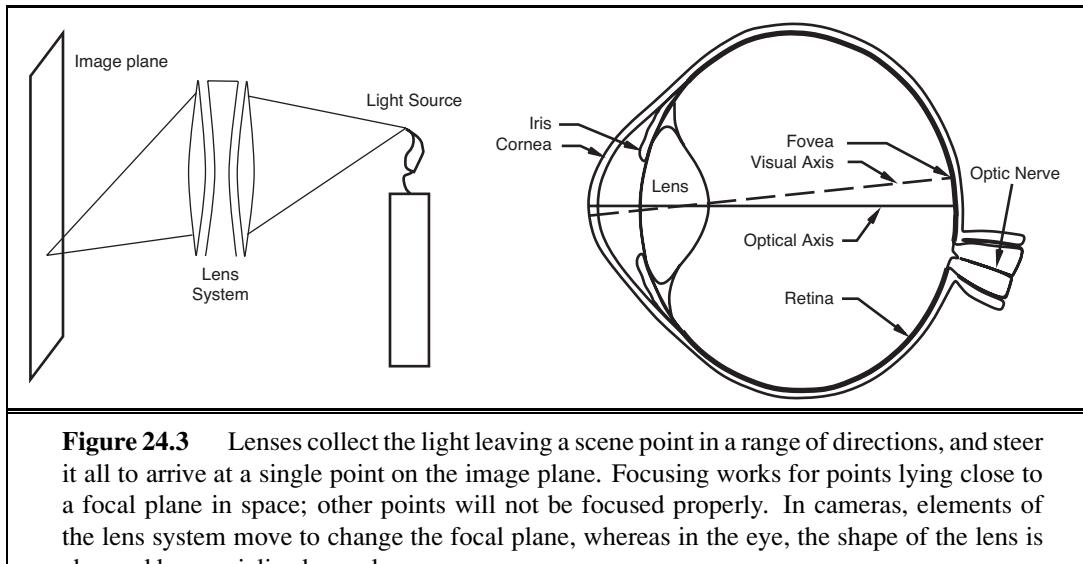
As $\lambda \rightarrow \infty$ or $\lambda \rightarrow -\infty$, this becomes $p_\infty = (fU/W, fV/W)$ if $W \neq 0$. This means that two parallel lines leaving different points in space will converge in the image—for large λ , the image points are nearly the same, whatever the value of (X_0, Y_0, Z_0) (again, think railway tracks, Figure 24.1). 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.

VANISHING POINT

MOTION BLUR

24.1.2 Lens systems

The drawback of the pinhole camera is that we need a small pinhole to keep the image in focus. But the smaller the pinhole, the fewer photons get through, meaning the image will be dark. We can gather more photons by keeping the pinhole open longer, but then we will get **motion blur**—objects in the scene that move will appear blurred because they send photons to multiple locations on the image plane. If we can't keep the pinhole open longer, we can try to make it bigger. More light will enter, but light from a small patch of object in the scene will now be spread over a patch on the image plane, causing a blurred image.



LENS Vertebrate eyes and modern cameras use a **lens** system to gather sufficient light while keeping the image in focus. A large opening is covered with a lens that focuses light from nearby object locations down to nearby locations in the image plane. However, lens systems have a limited **depth of field**: they can focus light only from points that lie within a range of depths (centered around a **focal plane**). Objects outside this range will be out of focus in the image. To move the focal plane, the lens in the eye can change shape (Figure 24.3); in a camera, the lenses move back and forth.

DEPTH OF FIELD

FOCAL PLANE

**SCALED
ORTHOGRAPHIC
PROJECTION**

24.1.3 Scaled orthographic projection

Perspective effects aren't always pronounced. For example, spots on a distant leopard may look small because the leopard is far away, but two spots that are next to each other will have about the same size. This is because the difference in distance to the spots is small compared to the distance to them, and so we can simplify the projection model. The appropriate model is **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 $y = sY$. Scaled orthographic projection is an approximation that is valid only for those parts of the scene with not much internal depth variation. For example, scaled orthographic projection can be a good model for the features on the front of a distant building.

24.1.4 Light and shading

The brightness of a pixel in the image is a function of the brightness of the surface patch in the scene that projects to the pixel. We will assume a linear model (current cameras have non-linearities at the extremes of light and dark, but are linear in the middle). Image brightness is

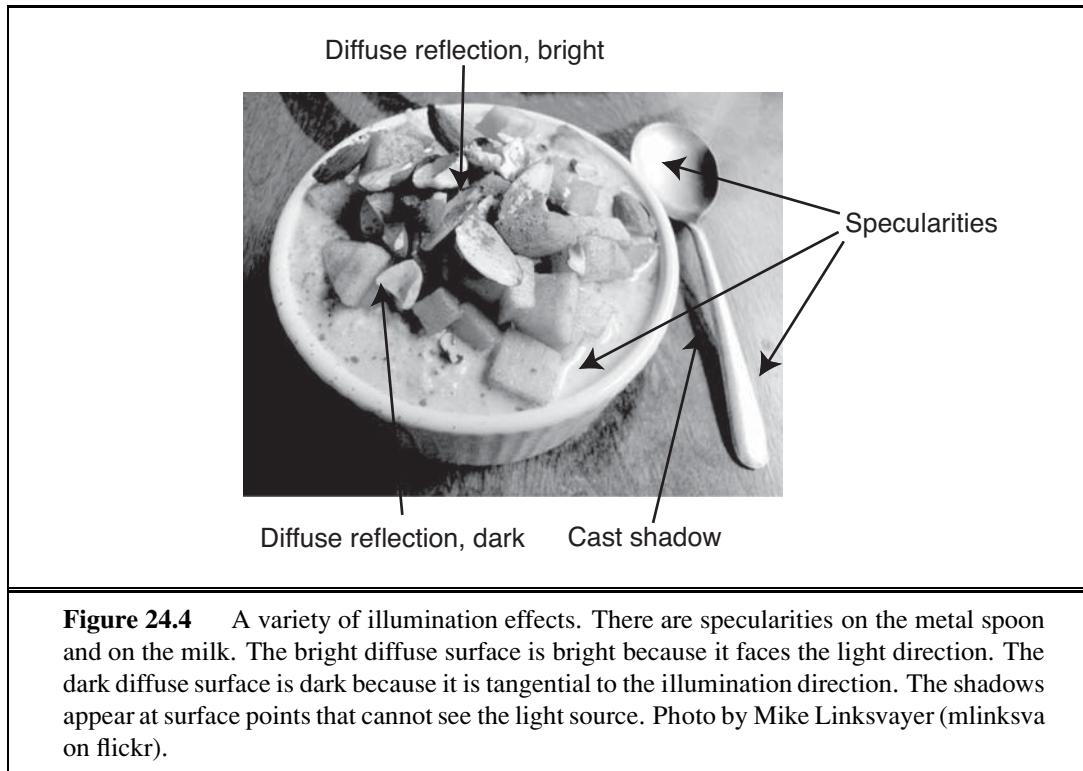


Figure 24.4 A variety of illumination effects. There are specularities on the metal spoon and on the milk. The bright diffuse surface is bright because it faces the light direction. The dark diffuse surface is dark because it is tangential to the illumination direction. The shadows appear at surface points that cannot see the light source. Photo by Mike Linksvayer (mlinksva on flickr).

OVERALL INTENSITY

REFLECT

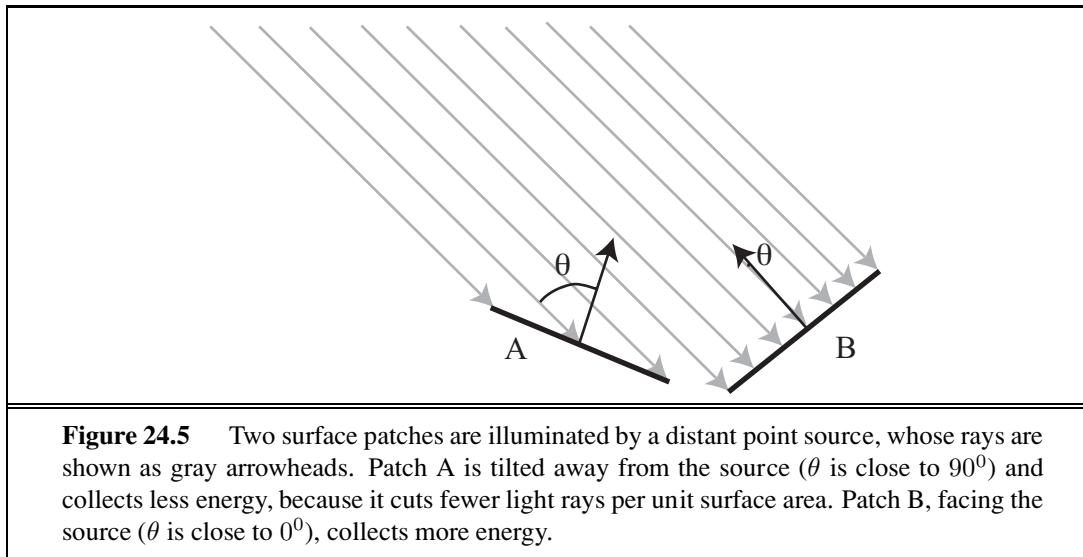
SHADING

DIFFUSE REFLECTION

SPECULAR REFLECTION
SPECULARITIES

a strong, if ambiguous, cue to the shape of an object, and from there to its identity. People are usually able to distinguish the three main causes of varying brightness and reverse-engineer the object's properties. The first cause is **overall intensity** of the light. Even though a white object in shadow may be less bright than a black object in direct sunlight, the eye can distinguish relative brightness well, and perceive the white object as white. Second, different points in the scene may **reflect** more or less of the light. Usually, the result is that people perceive these points as lighter or darker, and so see texture or markings on the object. Third, surface patches facing the light are brighter than surface patches tilted away from the light, an effect known as **shading**. Typically, people can tell that this shading comes from the geometry of the object, but sometimes get shading and markings mixed up. For example, a streak of dark makeup under a cheekbone will often look like a shading effect, making the face look thinner.

Most surfaces reflect light by a process of **diffuse reflection**. Diffuse reflection scatters light evenly across the directions leaving a surface, so the brightness of a diffuse surface doesn't depend on the viewing direction. Most cloth, paints, rough wooden surfaces, vegetation, and rough stone are diffuse. Mirrors are not diffuse, because what you see depends on the direction in which you look at the mirror. The behavior of a perfect mirror is known as **specular reflection**. Some surfaces—such as brushed metal, plastic, or a wet floor—display small patches where specular reflection has occurred, called **specularities**. These are easy to identify, because they are small and bright (Figure 24.4). For almost all purposes, it is enough to model all surfaces as being diffuse with specularities.

DISTANT POINT
LIGHT SOURCE

DIFFUSE ALBEDO

LAMBERT'S COSINE
LAW

SHADOW

INTERREFLECTIONS

AMBIENT
ILLUMINATION

The main source of illumination outside is the sun, whose rays all travel parallel to one another. We model this behavior as a **distant point light source**. This is the most important model of lighting, and is quite effective for indoor scenes as well as outdoor scenes. The amount of light collected by a surface patch in this model depends on the angle θ between the illumination direction and the normal to the surface.

A diffuse surface patch illuminated by a distant point light source will reflect some fraction of the light it collects; this fraction is called the **diffuse albedo**. White paper and snow have a high albedo, about 0.90, whereas flat black velvet and charcoal have a low albedo of about 0.05 (which means that 95% of the incoming light is absorbed within the fibers of the velvet or the pores of the charcoal). **Lambert's cosine law** states that the brightness of a diffuse patch is given by

$$I = \rho I_0 \cos \theta ,$$

where ρ is the diffuse albedo, I_0 is the intensity of the light source and θ is the angle between the light source direction and the surface normal (see Figure 24.5). Lambert's law predicts bright image pixels come from surface patches that face the light directly and dark pixels come from patches that see the light only tangentially, so that the shading on a surface provides some shape information. We explore this cue in Section 24.4.5. If the surface is not reached by the light source, then it is in **shadow**. Shadows are very seldom a uniform black, because the shadowed surface receives some light from other sources. Outdoors, the most important such source is the sky, which is quite bright. Indoors, light reflected from other surfaces illuminates shadowed patches. These **interreflections** can have a significant effect on the brightness of other surfaces, too. These effects are sometimes modeled by adding a constant **ambient illumination** term to the predicted intensity.

24.1.5 Color

Fruit is a bribe that a tree offers to animals to carry its seeds around. Trees have evolved to have fruit that turns red or yellow when ripe, and animals have evolved to detect these color changes. Light arriving at the eye has different amounts of energy at different wavelengths; this can be represented by a spectral energy density function. Human eyes respond to light in the 380–750nm wavelength region, with three different types of color receptor cells, which have peak receptiveness at 420nm (blue), 540nm (green), and 570nm (red). The human eye can capture only a small fraction of the full spectral energy density function—but it is enough to tell when the fruit is ripe.

PRINCIPLE OF TRICHRMOMACY

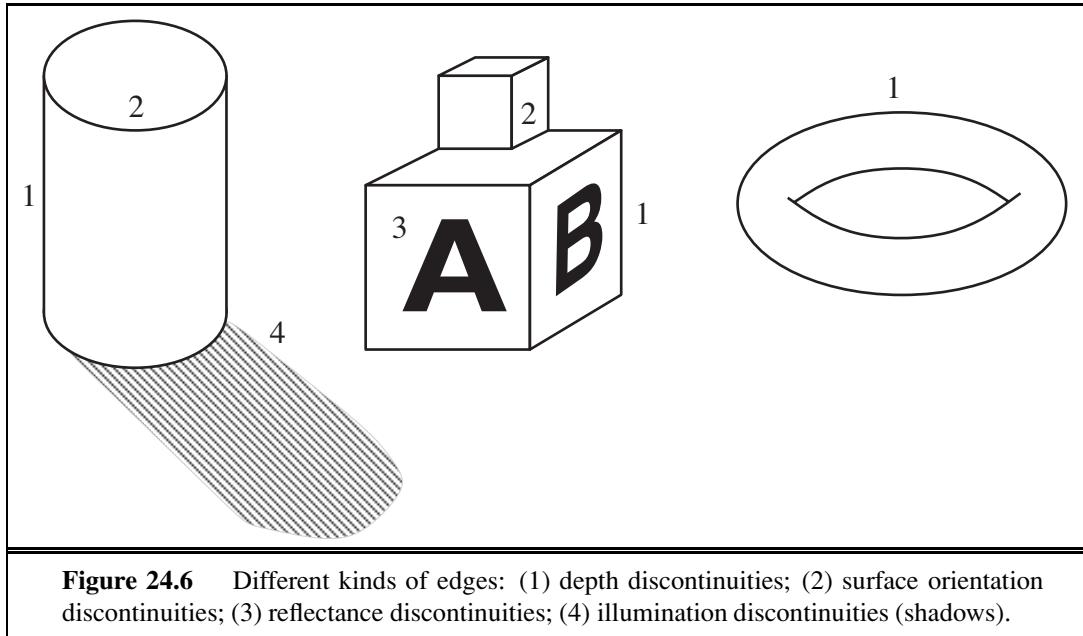
The **principle of trichromacy** states that for any spectral energy density, no matter how complicated, it is possible to construct another spectral energy density consisting of a mixture of just three colors—usually red, green, and blue—such that a human can't tell the difference between the two. That means that our TVs and computer displays can get by with just the three red/green/blue (or R/G/B) color elements. It makes our computer vision algorithms easier, too. Each surface can be modeled with three different albedos for R/G/B. Similarly, each light source can be modeled with three R/G/B intensities. We then apply Lambert's cosine law to each to get three R/G/B pixel values. This model predicts, correctly, that the same surface will produce different colored image patches under different-colored lights. In fact, human observers are quite good at ignoring the effects of different colored lights and are able to estimate the color of the surface under white light, an effect known as **color constancy**. Quite accurate color constancy algorithms are now available; simple versions show up in the “auto white balance” function of your camera. Note that if we wanted to build a camera for mantis shrimp, we would need 12 different pixel colors, corresponding to the 12 types of color receptors of the crustacean.

COLOR CONSTANCY

24.2 EARLY IMAGE-PROCESSING OPERATIONS

We have seen how light reflects off objects in the scene to form an image consisting of, say, five million 3-byte pixels. With all sensors there will be noise in the image, and in any case there is a lot of data to deal with. So how do we get started on analyzing this data?

In this section we will study three useful image-processing operations: edge detection, texture analysis, and computation of optical flow. 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 perform these operations without consideration of the objects that might be present in the scene. This makes the low-level operations good candidates for implementation in parallel hardware—either in a graphics processor unit (GPU) or an eye. We will then look at one mid-level operation: segmenting the image into regions.



24.2.1 Edge detection

EDGE

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, multimegabyte image and toward a more compact, abstract representation, as in Figure 24.6. The motivation is that edge contours in the image correspond to important scene contours. In the figure we have three examples of depth discontinuity, labeled 1; two surface-normal discontinuities, labeled 2; a reflectance discontinuity, labeled 3; and an illumination discontinuity (shadow), labeled 4. Edge detection is concerned only with the image, and thus does not distinguish between these different types of scene discontinuities; later processing will.

Figure 24.7(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. There are gaps where no edge appears, and there are “noise” edges 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 24.8 (top).

Edges correspond to locations in images where the brightness undergoes a sharp change, so a naive idea would be to differentiate the image and look for places where the magnitude of the derivative $I'(x)$ is large. That almost works. In Figure 24.8 (middle), we see that there is indeed a peak at $x = 50$, but there are also subsidiary peaks at other locations (e.g., $x = 75$). 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 the bottom of the figure.

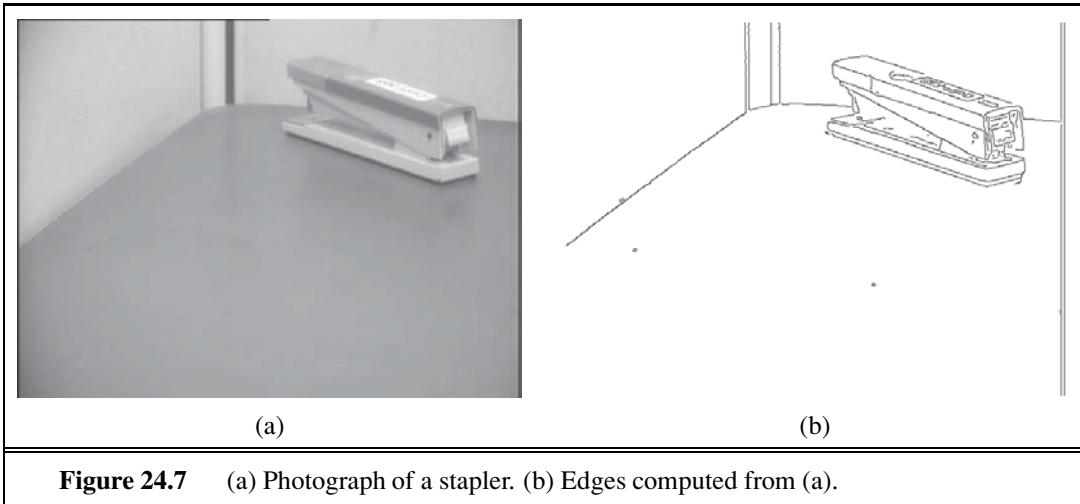


Figure 24.7 (a) Photograph of a stapler. (b) Edges computed from (a).

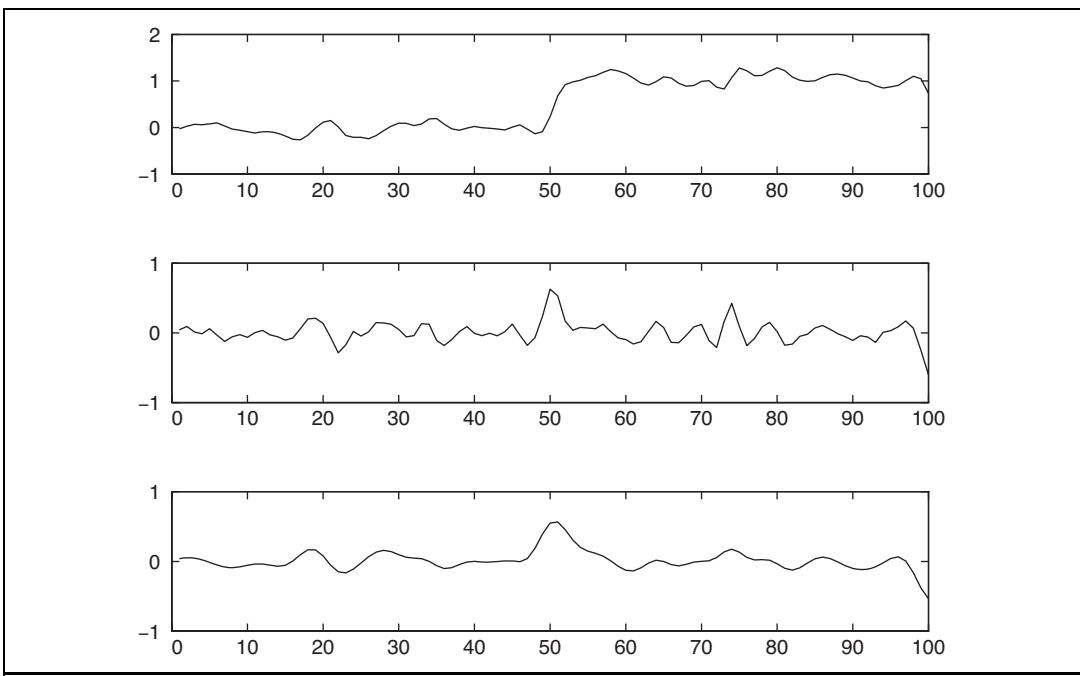


Figure 24.8 Top: Intensity profile $I(x)$ along a one-dimensional section across an edge at $x = 50$. Middle: The derivative of intensity, $I'(x)$. Large values of this function correspond to edges, but the function is noisy. Bottom: The derivative of a smoothed version of the intensity, $(I * G_\sigma)'$, which can be computed in one step as the convolution $I * G'_\sigma$. The noisy candidate edge at $x = 75$ has disappeared.

The measurement of brightness at a pixel in a CCD camera is based on a physical process involving the absorption of photons and the release of electrons; inevitably there will be statistical fluctuations of the measurement—noise. The noise can be modeled with

GAUSSIAN FILTER

a Gaussian probability distribution, with each pixel independent of the others. One way to smooth an image is to assign to each pixel the average of its neighbors. This tends to cancel out extreme values. But how many neighbors should we consider—one pixel away, or two, or more? One good answer is a weighted average that weights the nearest pixels the most, then gradually decreases the weight for more distant pixels. The **Gaussian filter** does just that. (Users of Photoshop recognize this as the *Gaussian blur* operation.) Recall that the Gaussian function with standard deviation σ and mean 0 is

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

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

CONVOLUTION

The application of the Gaussian filter replaces the intensity $I(x_0, y_0)$ with the sum, over all (x, y) pixels, of $I(x, y) N_\sigma(d)$, 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 $f * g$) if we have

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

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

So the smoothing function is achieved by convolving the image with the Gaussian, $I * N_\sigma$. 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 quickly at a distance, we can replace the $\pm\infty$ in the sums with $\pm 3\sigma$.

We can optimize the computation by combining smoothing and 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 smoothing function, N'_σ . We then mark as edges those peaks in the response that are above some threshold.

There is a natural generalization of this algorithm from one-dimensional cross sections to general two-dimensional images. In two dimensions edges may be at any angle θ . Considering the image brightness as a scalar function of the variables x, y , its gradient is a vector

$$\nabla I = \begin{pmatrix} \frac{\partial I}{\partial x} \\ \frac{\partial I}{\partial y} \end{pmatrix} = \begin{pmatrix} I_x \\ I_y \end{pmatrix}.$$

Edges correspond to locations in images where the brightness undergoes a sharp change, and so the magnitude of the gradient, $\|\nabla I\|$, should be large at an edge point. Of independent interest is the direction of the gradient

$$\frac{\nabla I}{\|\nabla I\|} = \begin{pmatrix} \cos \theta \\ \sin \theta \end{pmatrix}.$$

ORIENTATION

This gives us a $\theta = \theta(x, y)$ at every pixel, which defines the edge **orientation** at that pixel.

As in one dimension, to form the gradient we don't compute ∇I , but rather $\nabla(I * N_\sigma)$, the gradient after smoothing the image by convolving it with a Gaussian. And again, the shortcut is that this is equivalent to convolving the image with the partial derivatives of a Gaussian. Once we have computed the gradient, we can obtain edges by finding edge points and linking them together. To tell whether a point is an edge point, we must look at other points a small distance forward and back along the direction of the gradient. If the gradient magnitude at one of these points is larger, then we could get a better edge point by shifting the edge curve very slightly. Furthermore, if the gradient magnitude is too small, the point cannot be an edge point. So at an edge point, the gradient magnitude is a local maximum along the direction of the gradient, and the gradient magnitude is above a suitable threshold.

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 edge pixels with consistent orientations must belong to the same edge curve.

24.2.2 Texture

TEXTURE

In everyday language, **texture** is the visual feel of a surface—what you see evokes what the surface might feel like if you touched it (“texture” has the same root as “textile”). In computational vision, texture refers to a spatially repeating pattern on a surface that can be sensed visually. Examples include the pattern of windows on a building, stitches on a sweater, spots on a leopard, blades of grass on a lawn, pebbles on a beach, and people in a stadium. Sometimes the arrangement is quite periodic, as in the stitches on a sweater; in other cases, such as pebbles on a beach, the regularity is only statistical.

Whereas brightness is a property of individual pixels, the concept of texture makes sense only for a multipixel patch. Given such a patch, we could compute the orientation at each pixel, and then characterize the patch by a histogram of orientations. The texture of bricks in a wall would have two peaks in the histogram (one vertical and one horizontal), whereas the texture of spots on a leopard's skin would have a more uniform distribution of orientations.

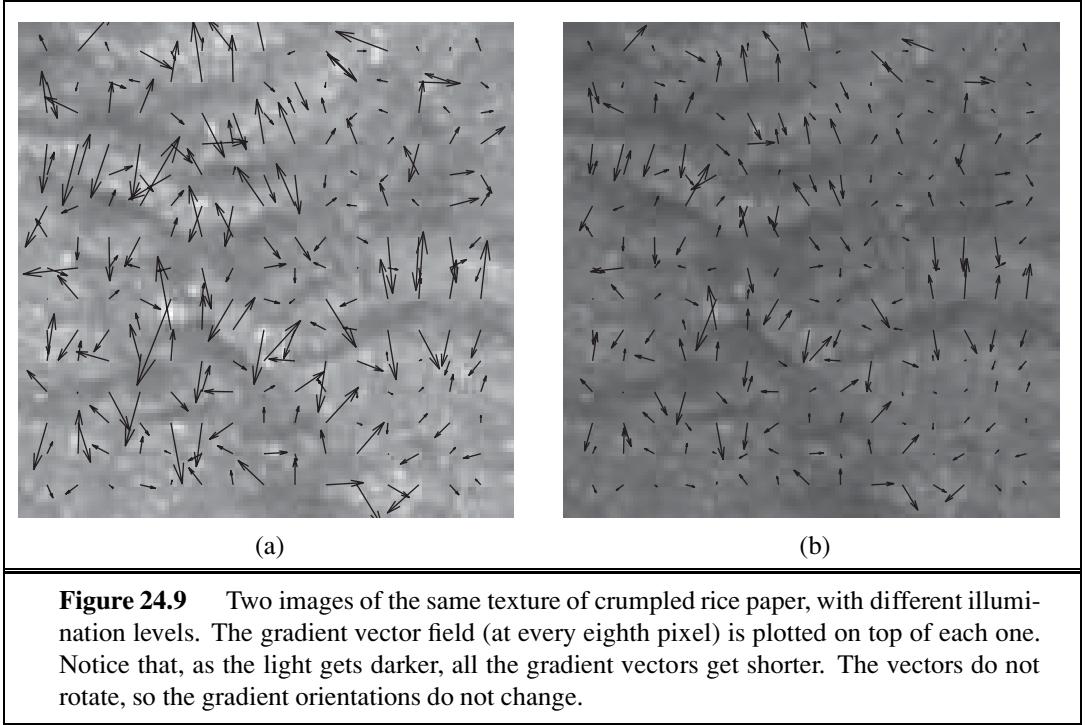
Figure 24.9 shows that orientations are largely invariant to changes in illumination. This makes texture an important clue for object recognition, because other clues, such as edges, can yield different results in different lighting conditions.

In images of textured objects, edge detection does not work as well as it does for smooth objects. This is because the most important edges can be lost among the texture elements. Quite literally, we may miss the tiger for the stripes. The solution is to look for differences in texture properties, just the way we look for differences in brightness. A patch on a tiger and a patch on the grassy background will have very different orientation histograms, allowing us to find the boundary curve between them.

24.2.3 Optical flow

OPTICAL FLOW

Next, let us consider what happens when we have a video sequence, instead of just a single static image. When an object in the video is moving, or when the camera is moving relative to an object, the resulting apparent motion in the image is called **optical flow**. Optical flow describes the direction and speed of motion of features *in the image*—the optical flow of a



video of a race car would be measured in pixels per second, not miles per hour. The optical flow encodes useful information about scene structure. For example, in a video of scenery taken from a moving train, distant objects have slower apparent motion than close objects; thus, the rate of apparent motion can tell us something about distance. Optical flow also enables us to recognize actions. In Figure 24.10(a) and (b), we show two frames from a video of a tennis player. In (c) we display the optical flow vectors computed from these images, showing that the racket and front leg are moving fastest.

The optical flow vector field can be represented at any point (x, y) 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. A simple-minded technique is based on 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 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):

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

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 $(v_x, v_y) = (D_x/D_t, D_y/D_t)$. Note that for this to work, there needs to be some texture or variation in the scene. If one is looking at a uniform white wall, then the SSD is going to be nearly the same for the different can-

SUM OF SQUARED
DIFFERENCES



Figure 24.10 Two frames of a video sequence. On the right is the optical flow field corresponding to the displacement from one frame to the other. Note how the movement of the tennis racket and the front leg is captured by the directions of the arrows. (Courtesy of Thomas Brox.)

dicate matches, and the algorithm is reduced to making a blind guess. The best-performing algorithms for measuring optical flow rely on a variety of additional constraints when the scene is only partially textured.

24.2.4 Segmentation of images

SEGMENTATION
REGIONS

Segmentation is the process of breaking an image into **regions** of similar pixels. Each image pixel can be associated with certain visual properties, such as brightness, color, and texture. 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 more of these attributes. There are two approaches to segmentation, one focusing on detecting the boundaries of these regions, and the other on detecting the regions themselves (Figure 24.11).

A boundary curve passing through a pixel (x, y) will have an orientation θ , so one way to formalize the problem of detecting boundary curves is as a machine learning classification problem. Based on features from a local neighborhood, we want to compute the probability $P_b(x, y, \theta)$ that indeed there is a boundary curve at that pixel along that orientation. Consider a circular disk centered at (x, y) , subdivided into two half disks by a diameter oriented at θ . If there is a boundary at (x, y, θ) the two half disks might be expected to differ significantly in their brightness, color, and texture. Martin, Fowlkes, and Malik (2004) used features based on differences in histograms of brightness, color, and texture values measured in these two half disks, and then trained a classifier. For this they used a data set of natural images where humans had marked the “ground truth” boundaries, and the goal of the classifier was to mark exactly those boundaries marked by humans and no others.

Boundaries detected by this technique turn out to be significantly better than those found using the simple edge-detection technique described previously. But still there are two limitations. (1) The boundary pixels formed by thresholding $P_b(x, y, \theta)$ are not guaranteed to form closed curves, so this approach doesn’t deliver regions, and (2) the decision making exploits only local context and does not use global consistency constraints.

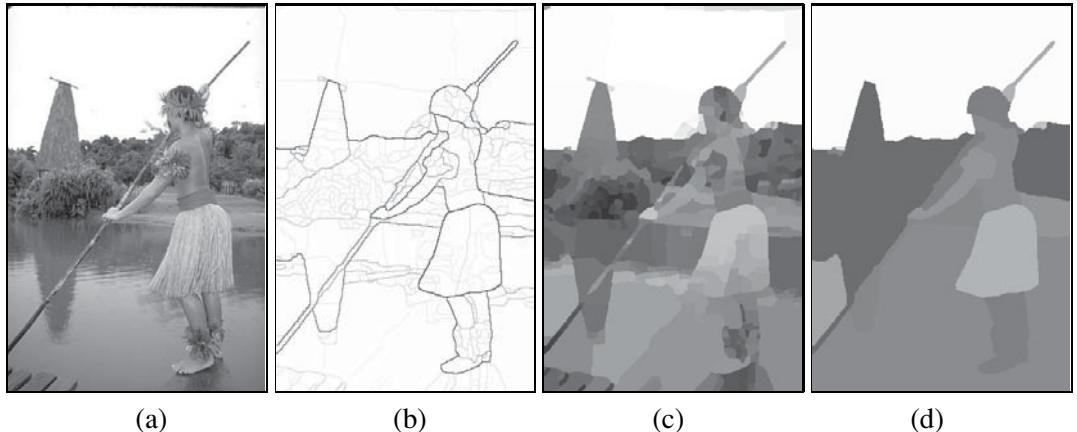


Figure 24.11 (a) Original image. (b) Boundary contours, where the higher the P_b value, the darker the contour. (c) Segmentation into regions, corresponding to a fine partition of the image. Regions are rendered in their mean colors. (d) Segmentation into regions, corresponding to a coarser partition of the image, resulting in fewer regions. (Courtesy of Pablo Arbelaez, Michael Maire, Charles Fowlkes, and Jitendra Malik)

The alternative approach is based on trying to “cluster” the pixels into regions based on their brightness, color, and texture. 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. Partitions that minimize a *normalized cut* criterion are then found. Roughly speaking, the criterion for partitioning the graph is to minimize the sum of weights of connections across the groups of pixels and maximize the sum of weights of connections within the groups.

Segmentation based purely on low-level, local attributes such as brightness and color cannot be expected to deliver the final correct boundaries of all the objects in the scene. To reliably find object boundaries we need high-level knowledge of the likely kinds of objects in the scene. Representing this knowledge is a topic of active research. A popular strategy is to produce an over-segmentation of an image, containing hundreds of homogeneous regions known as **superpixels**. From there, knowledge-based algorithms can take over; they will find it easier to deal with hundreds of superpixels rather than millions of raw pixels. How to exploit high-level knowledge of objects is the subject of the next section.

SUPERPIXELS

24.3 OBJECT RECOGNITION BY APPEARANCE

APPEARANCE

Appearance is shorthand for what an object tends to look like. Some object categories—for example, baseballs—vary rather little in appearance; all of the objects in the category look about the same under most circumstances. In this case, we can compute a set of features describing each class of images likely to contain the object, then test it with a classifier.

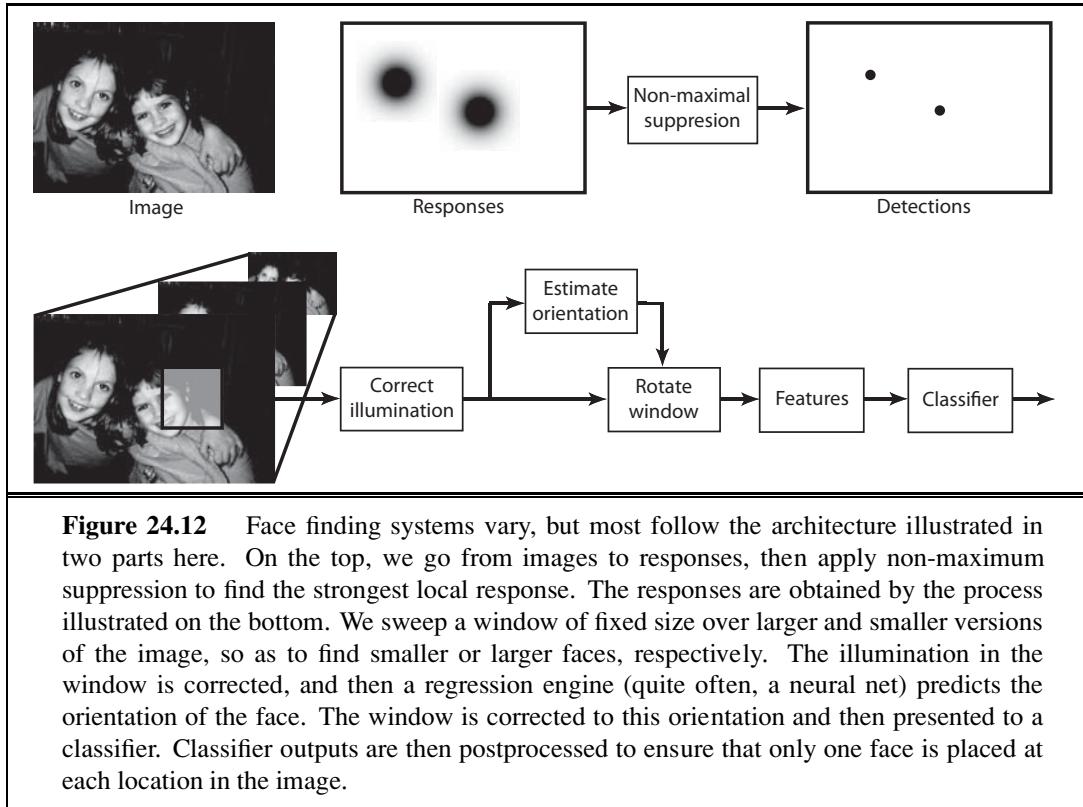
Other object categories—for example, houses or ballet dancers—vary greatly. A house can have different size, color, and shape and can look different from different angles. A dancer looks different in each pose, or when the stage lights change colors. A useful abstraction is to say that some objects are made up of local patterns which tend to move around with respect to one another. We can then find the object by looking at local histograms of detector responses, which expose whether some part is present but suppress the details of where it is.

Testing each class of images with a learned classifier is an important general recipe. It works extremely well for faces looking directly at the camera, because at low resolution and under reasonable lighting, all such faces look quite similar. The face is round, and quite bright compared to the eye sockets; these are dark, because they are sunken, and the mouth is a dark slash, as are the eyebrows. Major changes of illumination can cause some variations in this pattern, but the range of variation is quite manageable. That makes it possible to detect face positions in an image that contains faces. Once a computational challenge, this feature is now commonplace in even inexpensive digital cameras.

For the moment, we will consider only faces where the nose is oriented vertically; we will deal with rotated faces below. We sweep a round window of fixed size over the image, compute features for it, and present the features to a classifier. This strategy is sometimes called the **sliding window**. Features need to be robust to shadows and to changes in brightness caused by illumination changes. One strategy is to build features out of gradient orientations. Another is to estimate and correct the illumination in each image window. To find faces of different sizes, repeat the sweep over larger or smaller versions of the image. Finally, we postprocess the responses across scales and locations to produce the final set of detections.

Postprocessing is important, because it is unlikely that we have chosen a window size that is exactly the right size for a face (even if we use multiple sizes). Thus, we will likely have several overlapping windows that each report a match for a face. However, if we use a classifier that can report strength of response (for example, logistic regression or a support vector machine) we can combine these partial overlapping matches at nearby locations to yield a single high-quality match. That gives us a face detector that can search over locations and scales. To search rotations as well, we use two steps. We train a regression procedure to estimate the best orientation of any face present in a window. Now, for each window, we estimate the orientation, reorient the window, then test whether a vertical face is present with our classifier. All this yields a system whose architecture is sketched in Figure 24.12.

Training data is quite easily obtained. There are several data sets of marked-up face images, and rotated face windows are easy to build (just rotate a window from a training data set). One trick that is widely used is to take each example window, then produce new examples by changing the orientation of the window, the center of the window, or the scale very slightly. This is an easy way of getting a bigger data set that reflects real images fairly well; the trick usually improves performance significantly. Face detectors built along these lines now perform very well for frontal faces (side views are harder).

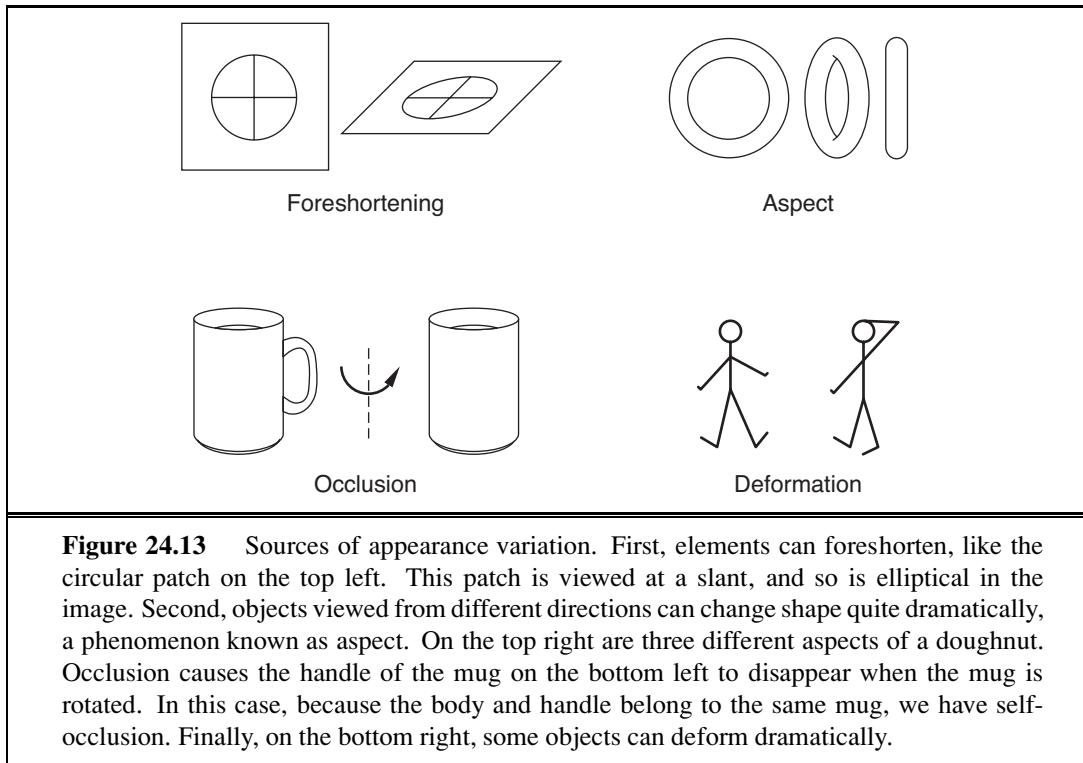


24.3.1 Complex appearance and pattern elements

Many objects produce much more complex patterns than faces do. This is because several effects can move features around in an image of the object. Effects include (Figure 24.13)

- **Foreshortening**, which causes a pattern viewed at a slant to be significantly distorted.
- **Aspect**, which causes objects to look different when seen from different directions. Even as simple an object as a doughnut has several aspects; seen from the side, it looks like a flattened oval, but from above it is an annulus.
- **Occlusion**, where some parts are hidden from some viewing directions. Objects can occlude one another, or parts of an object can occlude other parts, an effect known as self-occlusion.
- **Deformation**, where internal degrees of freedom of the object change its appearance. For example, people can move their arms and legs around, generating a very wide range of different body configurations.

However, our recipe of searching across location and scale can still work. This is because some structure will be present in the images produced by the object. For example, a picture of a car is likely to show some of headlights, doors, wheels, windows, and hubcaps, though they may be in somewhat different arrangements in different pictures. This suggests modeling objects with pattern elements—collections of parts. These pattern elements may move around

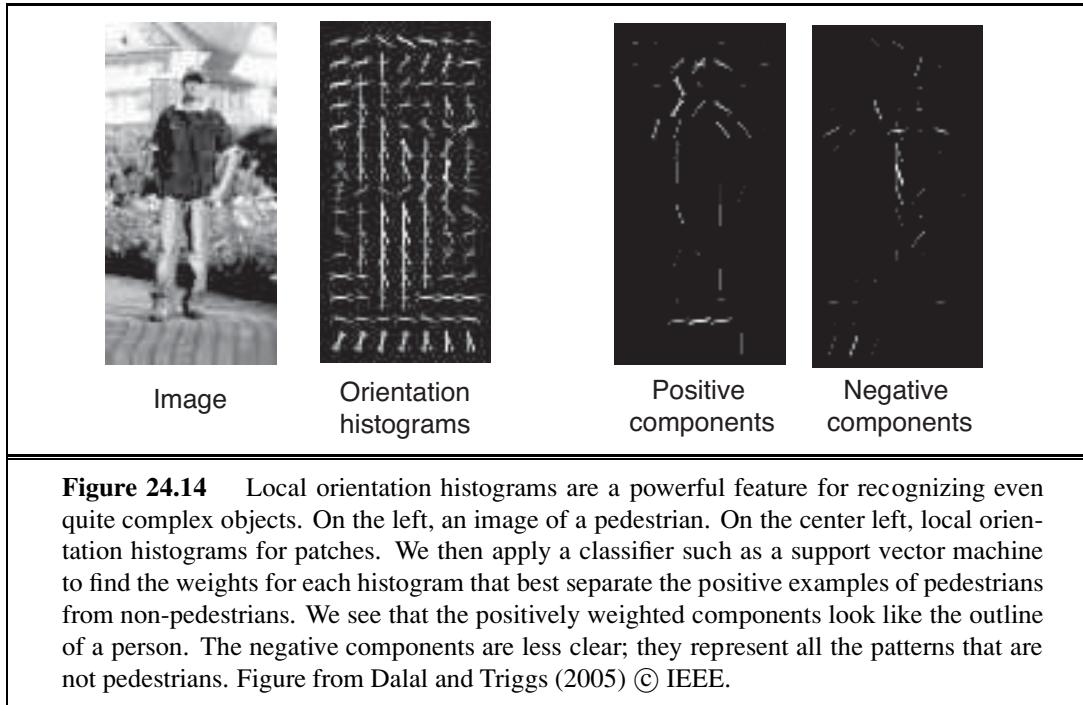


with respect to one another, but if most of the pattern elements are present in about the right place, then the object is present. An object recognizer is then a collection of features that can tell whether the pattern elements are present, and whether they are in about the right place.

The most obvious approach is to represent the image window with a histogram of the pattern elements that appear there. This approach does not work particularly well, because too many patterns get confused with one another. For example, if the pattern elements are color pixels, the French, UK, and Netherlands flags will get confused because they have approximately the same color histograms, though the colors are arranged in very different ways. Quite simple modifications of histograms yield very useful features. The trick is to preserve some spatial detail in the representation; for example, headlights tend to be at the front of a car and wheels tend to be at the bottom. Histogram-based features have been successful in a wide variety of recognition applications; we will survey pedestrian detection.

24.3.2 Pedestrian detection with HOG features

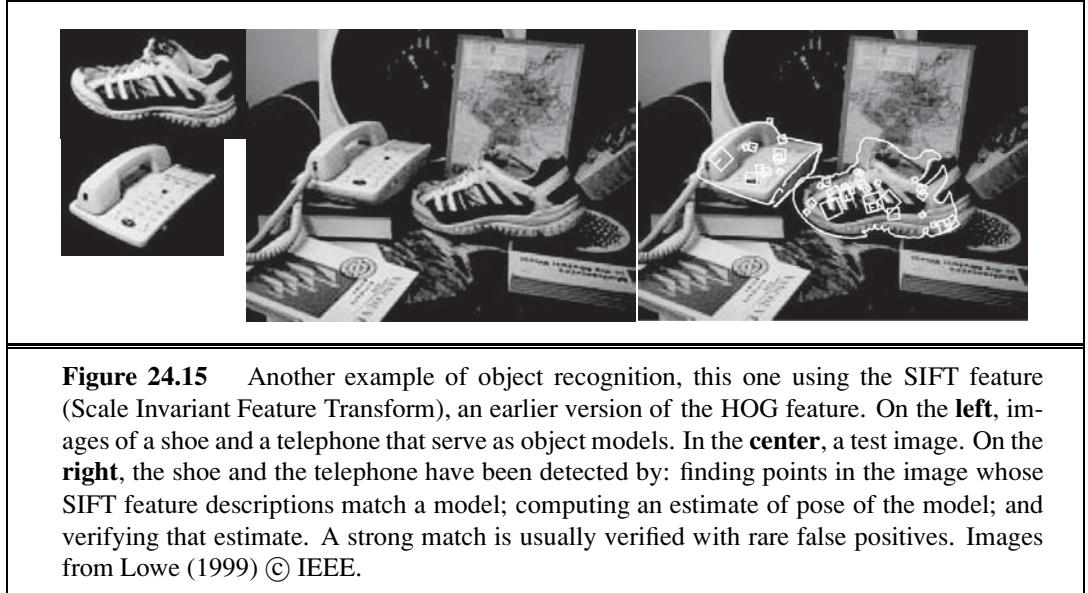
The World Bank estimates that each year car accidents kill about 1.2 million people, of whom about two thirds are pedestrians. This means that detecting pedestrians is an important application problem, because cars that can automatically detect and avoid pedestrians might save many lives. Pedestrians wear many different kinds of clothing and appear in many different configurations, but, at relatively low resolution, pedestrians can have a fairly characteristic appearance. The most usual cases are lateral or frontal views of a walk. In these cases,



we see either a “lollipop” shape — the torso is wider than the legs, which are together in the stance phase of the walk — or a “scissor” shape — where the legs are swinging in the walk. We expect to see some evidence of arms and legs, and the curve around the shoulders and head also tends to visible and quite distinctive. This means that, with a careful feature construction, we can build a useful moving-window pedestrian detector.

There isn’t always a strong contrast between the pedestrian and the background, so it is better to use orientations than edges to represent the image window. Pedestrians can move their arms and legs around, so we should use a histogram to suppress some spatial detail in the feature. We break up the window into cells, which could overlap, and build an orientation histogram in each cell. Doing so will produce a feature that can tell whether the head-and-shoulders curve is at the top of the window or at the bottom, but will not change if the head moves slightly.

One further trick is required to make a good feature. Because orientation features are not affected by illumination brightness, we cannot treat high-contrast edges specially. This means that the distinctive curves on the boundary of a pedestrian are treated in the same way as fine texture detail in clothing or in the background, and so the signal may be submerged in noise. We can recover contrast information by counting gradient orientations with weights that reflect how significant a gradient is compared to other gradients in the same cell. We will write $\|\nabla I_x\|$ for the gradient magnitude at point x in the image, write C for the cell whose histogram we wish to compute, and write $w_{x,C}$ for the weight that we will use for the



orientation at \mathbf{x} for this cell. A natural choice of weight is

$$w_{\mathbf{x}, \mathcal{C}} = \frac{\|\nabla I_{\mathbf{x}}\|}{\sum_{\mathbf{u} \in \mathcal{C}} \|\nabla I_{\mathbf{u}}\|}.$$

HOG FEATURE

This compares the gradient magnitude to others in the cell, so gradients that are large compared to their neighbors get a large weight. The resulting feature is usually called a **HOG feature** (for Histogram Of Gradient orientations).

This feature construction is the main way in which pedestrian detection differs from face detection. Otherwise, building a pedestrian detector is very like building a face detector. The detector sweeps a window across the image, computes features for that window, then presents it to a classifier. Non-maximum suppression needs to be applied to the output. In most applications, the scale and orientation of typical pedestrians is known. For example, in driving applications in which a camera is fixed to the car, we expect to view mainly vertical pedestrians, and we are interested only in nearby pedestrians. Several pedestrian data sets have been published, and these can be used for training the classifier.

Pedestrians are not the only type of object we can detect. In Figure 24.15 we see that similar techniques can be used to find a variety of objects in different contexts.

24.4 RECONSTRUCTING THE 3D WORLD

In this section we show how to go from the two-dimensional image to a three-dimensional representation of the scene. The fundamental question is this: Given that all points in the scene that fall along a ray to the pinhole are projected to the same point in the image, how do we recover three-dimensional information? Two ideas come to our rescue:

- If we have two (or more) images from different camera positions, then we can triangulate to find the position of a point in the scene.
- We can exploit background knowledge about the physical scene that gave rise to the image. Given an object model $\mathbf{P}(\text{Scene})$ and a rendering model $\mathbf{P}(\text{Image} \mid \text{Scene})$, we can compute a posterior distribution $\mathbf{P}(\text{Scene} \mid \text{Image})$.

There is as yet no single unified theory for scene reconstruction. We survey eight commonly used visual cues: **motion**, **binocular stereopsis**, **multiple views**, **texture**, **shading**, **contour**, and **familiar objects**.

24.4.1 Motion parallax

If the camera moves relative to the three-dimensional scene, the resulting apparent motion in the image, optical flow, can be a source of information for both the movement of the camera and depth in the scene. To understand this, we state (without proof) an equation that relates the optical flow to the viewer's translational velocity \mathbf{T} and the depth in the scene.

The components of the optical flow field are

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

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

FOCUS OF EXPANSION

Note that 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

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

Note that there is a scale-factor ambiguity here. If the camera was moving twice as fast, and every object in the scene was twice as big and at twice the distance to the camera, the optical flow field would be exactly the same. But we can still extract quite useful information.

1. 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. There is considerable experimental evidence that many different animal species exploit this cue.
2. Consider two points at depths Z_1, Z_2 , respectively. We may not know the absolute value of either of these, but by considering the inverse of the ratio of the optical flow magnitudes at these points, we can determine the depth ratio Z_1/Z_2 . This is the cue of motion parallax, one we use when we look out of the side window of a moving car or train and infer that the slower moving parts of the landscape are farther away.

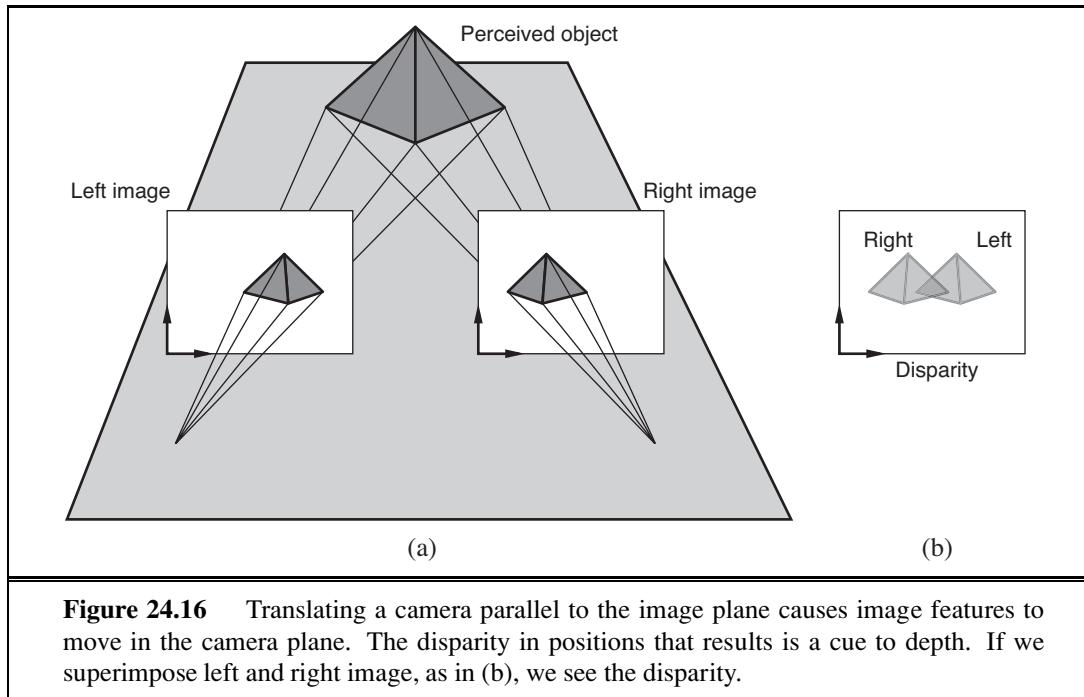


Figure 24.16 Translating a camera parallel to the image plane causes image features to move in the camera plane. The disparity in positions that results is a cue to depth. If we superimpose left and right image, as in (b), we see the disparity.

24.4.2 Binocular stereopsis

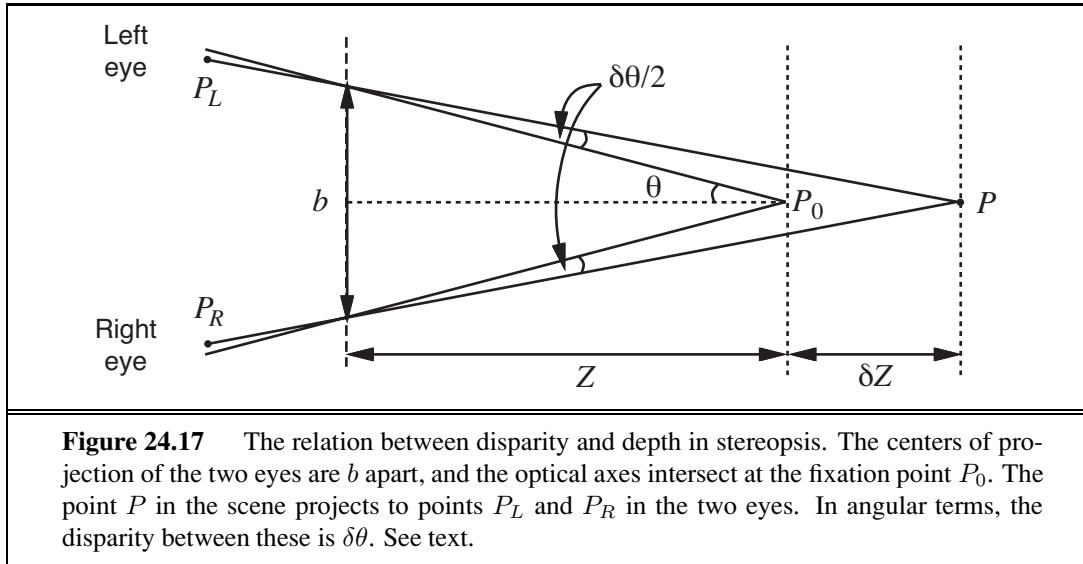
Most vertebrates have *two* eyes. This is useful for redundancy in case of a lost eye, but it helps in other ways too. Most prey have eyes on the side of the head to enable a wider field of vision. Predators have the eyes in the front, enabling them to use **binocular stereopsis**. The idea is similar to motion parallax, except that instead of using images over time, we use two (or more) images separated in space. 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 24.16, where the nearest point of the pyramid is shifted to the left in the right image and to the right in the left image.

Note that to measure disparity we need to solve the correspondence problem, that is, determine for a point in the left image, the point in the right image that results from the projection of the same scene point. This is analogous to what one has to do in measuring optical flow, and the most simple-minded approaches are somewhat similar and based on comparing blocks of pixels around corresponding points using the sum of squared differences. In practice, we use much more sophisticated algorithms, which exploit additional constraints.

Assuming that we can measure disparity, how does this yield information about depth in the scene? We will need to 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 a displacement along the x -axis by an amount b , the baseline. We can use the optical flow equations from the previous section, if we think of this as resulting from a translation

BINOCULAR
STEREOPSIS

DISPARITY



vector \mathbf{T} acting for time δt , with $T_x = b/\delta t$ and $T_y = T_z = 0$. The horizontal and vertical disparity are given by the optical flow components, multiplied by the time step δt , $H = v_x \delta t$, $V = v_y \delta t$. Carrying out the substitutions, we get the result that $H = b/Z$, $V = 0$. In words, the horizontal disparity is equal to the ratio of the baseline to the depth, and the vertical disparity is zero. Given that we know b , we can measure H and recover the depth Z .

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 24.17 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 farther 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 Figure 24.17, $\tan \theta = \frac{b/2}{Z}$ and $\tan(\theta - \delta\theta/2) = \frac{b/2}{Z + \delta Z}$, but for small angles, $\tan \theta \approx \theta$, so

$$\delta\theta/2 = \frac{b/2}{Z} - \frac{b/2}{Z + \delta Z} \approx \frac{b\delta Z}{2Z^2}$$

and, since the actual disparity is $\delta\theta$, we have

$$\text{disparity} = \frac{b\delta Z}{Z^2}.$$

BASELINE

In humans, b (the **baseline** distance between the eyes) is about 6 cm. Suppose that Z is about 100 cm. If the smallest detectable $\delta\theta$ (corresponding to the pixel size) is about 5 seconds of arc, this gives 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.

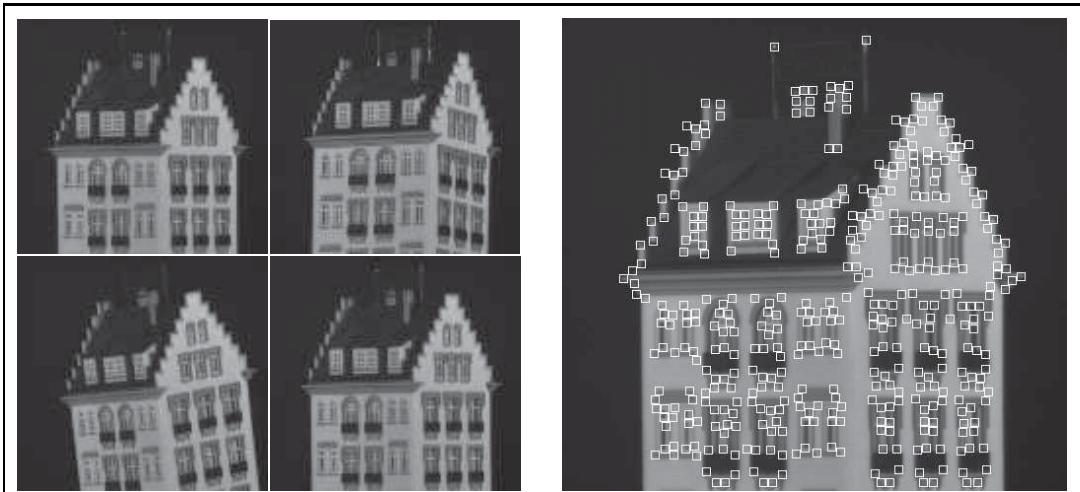


Figure 24.18 (a) Four frames from a video sequence in which the camera is moved and rotated relative to the object. (b) The first frame of the sequence, annotated with small boxes highlighting the features found by the feature detector. (Courtesy of Carlo Tomasi.)

24.4.3 Multiple views

Shape from optical flow or binocular disparity are two instances of a more general framework, that of exploiting multiple views for recovering depth. In computer vision, there is no reason for us to be restricted to differential motion or to only use two cameras converging at a fixation point. Therefore, techniques have been developed that exploit the information available in multiple views, even from hundreds or thousands of cameras. Algorithmically, there are three subproblems that need to be solved:

- The correspondence problem, i.e., identifying features in the different images that are projections of the same feature in the three-dimensional world.
- The relative orientation problem, i.e., determining the transformation (rotation and translation) between the coordinate systems fixed to the different cameras.
- The depth estimation problem, i.e., determining the depths of various points in the world for which image plane projections were available in at least two views

The development of robust matching procedures for the correspondence problem, accompanied by numerically stable algorithms for solving for relative orientations and scene depth, is one of the success stories of computer vision. Results from one such approach due to Tomasi and Kanade (1992) are shown in Figures 24.18 and 24.19.

24.4.4 Texture

Earlier we saw how texture was used for segmenting objects. It can also be used to estimate distances. In Figure 24.20 we see that a homogeneous texture in the scene results in varying texture elements, or **texels**, in the image. All the paving tiles in (a) are identical in the scene. They appear different in the image for two reasons:

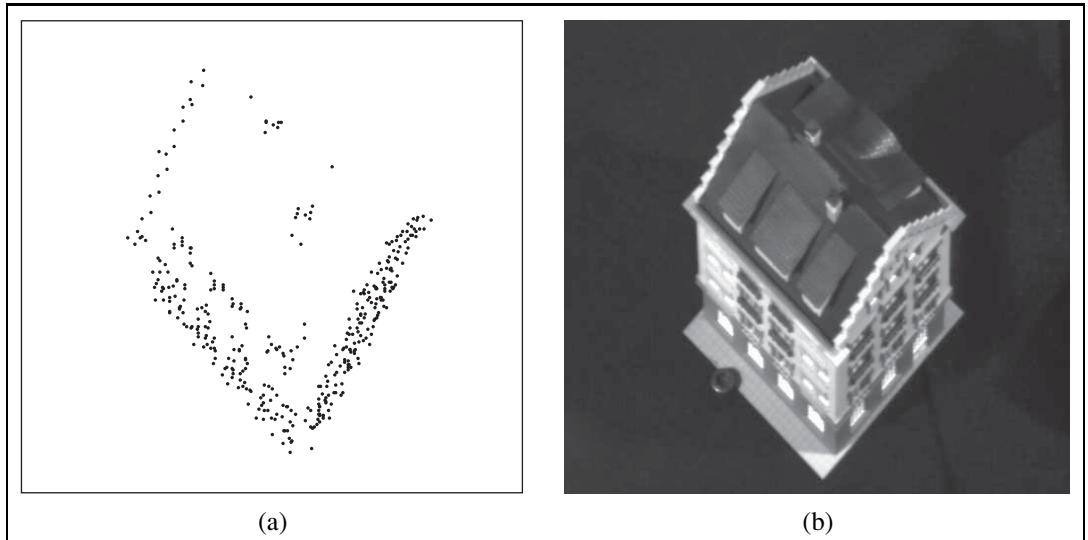


Figure 24.19 (a) Three-dimensional reconstruction of the locations of the image features in Figure 24.18, shown from above. (b) The real house, taken from the same position.

1. *Differences in the distances of the texels from the camera.* Distant objects appear smaller by a scaling factor of $1/Z$.
2. *Differences in the foreshortening of the texels.* If all the texels are in the ground plane then distance ones are viewed at an angle that is farther off the perpendicular, and so are more foreshortened. The magnitude of the foreshortening effect is proportional to $\cos \sigma$, where σ is the slant, the angle between the Z -axis and \mathbf{n} , the surface normal to the texel.

Researchers have developed various algorithms that try to exploit the variation in the appearance of the projected texels as a basis for determining surface normals. However, the accuracy and applicability of these algorithms is not anywhere as general as those based on using multiple views.

24.4.5 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.

From the physical model of section 24.1.4, we know that if a surface normal points toward the light source, the surface is brighter, and if it points away, the surface is darker. We cannot conclude that a dark patch has its normal pointing away from the light; instead, it could have low albedo. Generally, albedo changes quite quickly in images, and shading

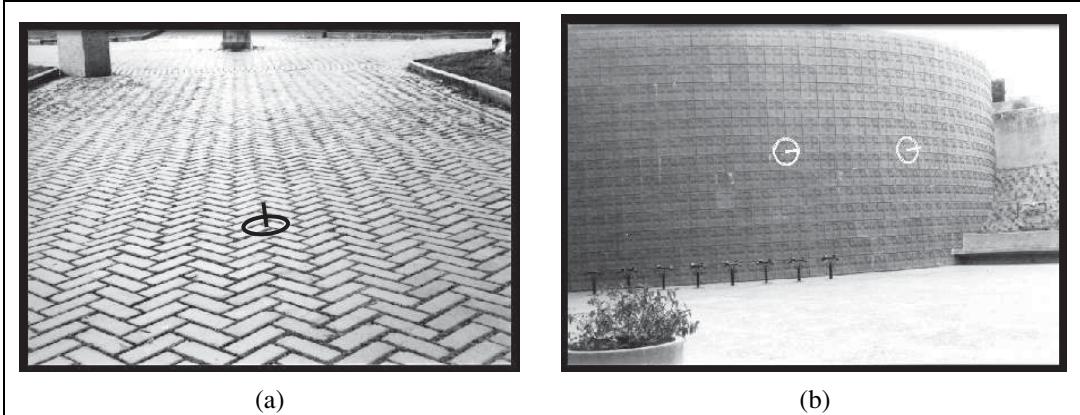


Figure 24.20 (a) A textured scene. Assuming that the real texture is uniform allows recovery of the surface orientation. The computed surface orientation is indicated by overlaying a black circle and pointer, transformed as if the circle were painted on the surface at that point. (b) Recovery of shape from texture for a curved surface (white circle and pointer this time). Images courtesy of Jitendra Malik and Ruth Rosenholtz (1994).

changes rather slowly, and humans seem to be quite good at using this observation to tell whether low illumination, surface orientation, or albedo caused a surface patch to be dark. To simplify the problem, let us assume that the albedo is known at every surface point. It is still difficult to recover the normal, because the image brightness is one measurement but the normal has two unknown parameters, so we cannot simply solve for the normal. The key to this situation seems to be that nearby normals will be similar, because most surfaces are smooth—they do not have sharp changes.

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 and make it quite difficult to predict the relationship between the normal and the image brightness. Two surface patches with the same normal might have quite different brightnesses, because one receives light reflected from a large white wall and the other faces only a dark bookcase. Despite these difficulties, the problem is important. Humans seem to be able to ignore the effects of interreflections and get a useful perception of shape from shading, but we know frustratingly little about algorithms to do this.

24.4.6 Contour

When we look at a line drawing, such as Figure 24.21, we get a vivid perception of three-dimensional shape and layout. How? It is a combination of recognition of familiar objects in the scene and the application of generic constraints such as the following:

- Occluding contours, such as the outlines of the hills. One side of the contour is nearer to the viewer, the other side is farther away. Features such as local convexity and sym-

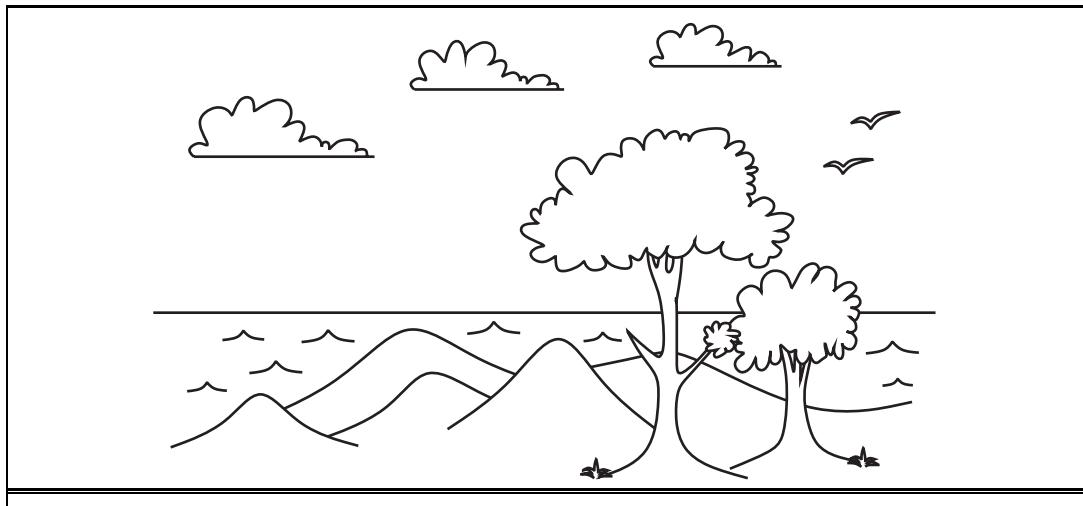


Figure 24.21 An evocative line drawing. (Courtesy of Isha Malik.)

FIGURE-GROUND

metry provide cues to solving the **figure-ground** problem—assigning which side of the contour is figure (nearer), and which is ground (farther). At an occluding contour, the line of sight is tangential to the surface in the scene.

- T-junctions. When one object occludes another, the contour of the farther object is interrupted, assuming that the nearer object is opaque. A T-junction results in the image.
- Position on the ground plane. Humans, like many other terrestrial animals, are very often in a scene that contains a **ground plane**, with various objects at different locations on this plane. Because of gravity, typical objects don't float in air but are supported by this ground plane, and we can exploit the very special geometry of this viewing scenario.

GROUND PLANE

Let us work out the projection of objects of different heights and at different locations on the ground plane. Suppose that the eye, or camera, is at a height h_c above the ground plane. Consider an object of height δY resting on the ground plane, whose bottom is at $(X, -h_c, Z)$ and top is at $(X, \delta Y - h_c, Z)$. The bottom projects to the image point $(fX/Z, -fh_c/Z)$ and the top to $(fX/Z, f(\delta Y - h_c)/Z)$. The bottoms of nearer objects (small Z) project to points lower in the image plane; farther objects have bottoms closer to the horizon.

24.4.7 Objects and the geometric structure of scenes

A typical adult human head is about 9 inches long. This means that for someone who is 43 feet away, the angle subtended by the head at the camera is 1 degree. If we see a person whose head appears to subtend just half a degree, Bayesian inference suggests we are looking at a normal person who is 86 feet away, rather than someone with a half-size head. This line of reasoning supplies us with a method to check the results of a pedestrian detector, as well as a method to estimate the distance to an object. For example, all pedestrians are about the same height, and they tend to stand on a ground plane. If we know where the horizon is in an image, we can rank pedestrians by distance to the camera. This works because we know where their

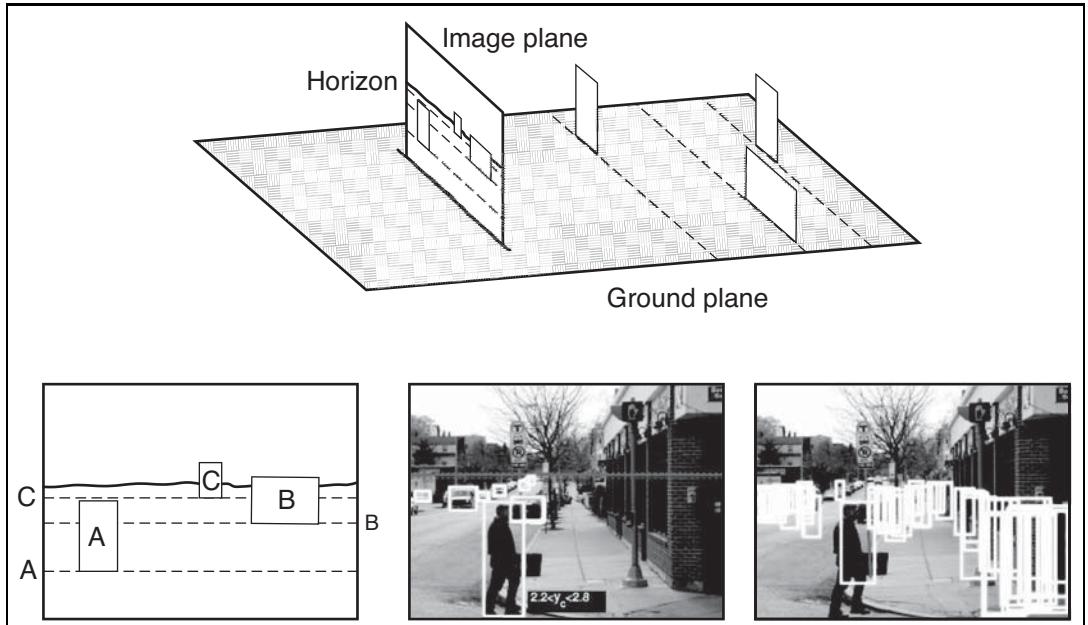


Figure 24.22 In an image of people standing on a ground plane, the people whose feet are closer to the horizon in the image must be farther away (top drawing). This means they must look smaller in the image (left lower drawing). This means that the size and location of real pedestrians in an image depend upon one another and on the location of the horizon. To exploit this, we need to identify the ground plane, which is done using shape-from-texture methods. From this information, and from some likely pedestrians, we can recover a horizon as shown in the center image. On the right, acceptable pedestrian boxes given this geometric context. Notice that pedestrians who are higher in the scene must be smaller. If they are not, then they are false positives. Images from Hoiem *et al.* (2008) © IEEE.

feet are, and pedestrians whose feet are closer to the horizon in the image are farther away from the camera (Figure 24.22). Pedestrians who are farther away from the camera must also be smaller in the image. This means we can rule out some detector responses — if a detector finds a pedestrian who is large in the image and whose feet are close to the horizon, it has found an enormous pedestrian; these don't exist, so the detector is wrong. In fact, many or most image windows are not acceptable pedestrian windows, and need not even be presented to the detector.

There are several strategies for finding the horizon, including searching for a roughly horizontal line with a lot of blue above it, and using surface orientation estimates obtained from texture deformation. A more elegant strategy exploits the reverse of our geometric constraints. A reasonably reliable pedestrian detector is capable of producing estimates of the horizon, if there are several pedestrians in the scene at different distances from the camera. This is because the relative scaling of the pedestrians is a cue to where the horizon is. So we can extract a horizon estimate from the detector, then use this estimate to prune the pedestrian detector's mistakes.

ALIGNMENT METHOD

If the object is familiar, we can estimate more than just the distance to it, because what it looks like in the image depends very strongly on its pose, i.e., its position and orientation with respect to the viewer. This has many applications. 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 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 \mathbf{R} , followed by translation by an unknown amount \mathbf{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

$$p_i = \Pi(\mathbf{R}m_i + \mathbf{t}) = Q(m_i)$$

for a three-dimensional model point m_i and the corresponding image point p_i . Here, \mathbf{R} is a rotation matrix, \mathbf{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 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.

We can solve for Q , given the three-dimensional coordinates of three model points and their two-dimensional projections. The intuition is as follows: we 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 \mathbf{R} and the translation vector \mathbf{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.

Let us specify position and orientation in mathematical terms. The position of a point P in the scene is characterized by three numbers, the (X, Y, Z) coordinates of P in a coordinate frame with its origin at the pinhole and the Z -axis along the optical axis (Figure 24.2 on page 931). What we have available is the perspective projection (x, y) of the point in the image. This specifies the ray from the pinhole along which 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.

SLANT
TILT

SHAPE

2. **The orientation of the surface of the object at P .** This can be specified by a normal vector, \mathbf{n} —which 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.

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.

24.5 OBJECT RECOGNITION FROM STRUCTURAL INFORMATION

DEFORMABLE
TEMPLATE

Putting a box around pedestrians in an image may well be enough to avoid driving into them. We have seen that we can find a box by pooling the evidence provided by orientations, using histogram methods to suppress potentially confusing spatial detail. If we want to know more about what someone is doing, we will need to know where their arms, legs, body, and head lie in the picture. Individual body parts are quite difficult to detect on their own using a moving window method, because their color and texture can vary widely and because they are usually small in images. Often, forearms and shins are as small as two to three pixels wide. Body parts do not usually appear on their own, and representing what is connected to what could be quite powerful, because parts that are easy to find might tell us where to look for parts that are small and hard to detect.

Inferring the layout of human bodies in pictures is an important task in vision, because the layout of the body often reveals what people are doing. A model called a **deformable template** can tell us which configurations are acceptable: the elbow can bend but the head is never joined to the foot. The simplest deformable template model of a person connects lower arms to upper arms, upper arms to the torso, and so on. There are richer models: for example,

we could represent the fact that left and right upper arms tend to have the same color and texture, as do left and right legs. These richer models remain difficult to work with, however.

24.5.1 The geometry of bodies: Finding arms and legs

For the moment, we assume that we know what the person's body parts look like (e.g., we know the color and texture of the person's clothing). We can model the geometry of the body as a tree of eleven segments (upper and lower left and right arms and legs respectively, a torso, a face, and hair on top of the face) each of which is rectangular. We assume that the position and orientation (**pose**) of the left lower arm is independent of all other segments given the pose of the left upper arm; that the pose of the left upper arm is independent of all segments given the pose of the torso; and extend these assumptions in the obvious way to include the right arm and the legs, the face, and the hair. Such models are often called "cardboard people" models. The model forms a tree, which is usually rooted at the torso. We will search the image for the best match to this cardboard person using inference methods for a tree-structured Bayes net (see Chapter 14).

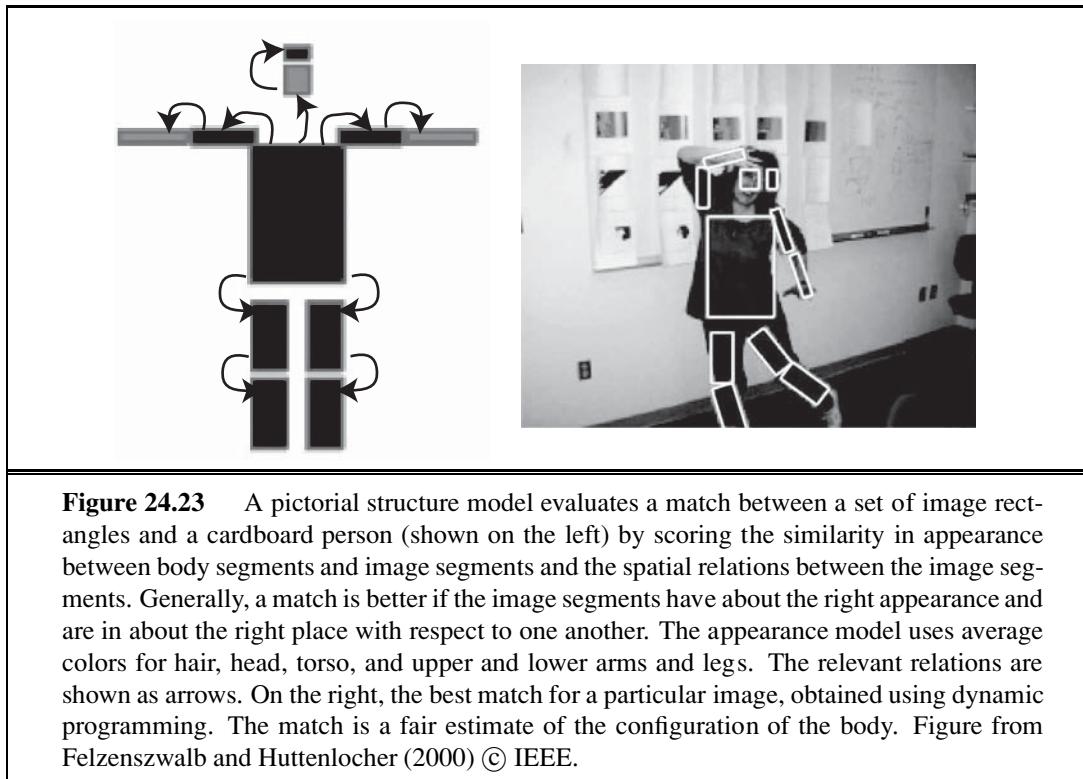
There are two criteria for evaluating a configuration. First, an image rectangle should look like its segment. For the moment, we will remain vague about precisely what that means, but we assume we have a function ϕ_i that scores how well an image rectangle matches a body segment. For each pair of related segments, we have another function ψ that scores how well relations between a pair of image rectangles match those to be expected from the body segments. The dependencies between segments form a tree, so each segment has only one parent, and we could write $\psi_{i,\text{pa}(i)}$. All the functions will be larger if the match is better, so we can think of them as being like a log probability. The cost of a particular match that allocates image rectangle m_i to body segment i is then

$$\sum_{i \in \text{segments}} \phi_i(m_i) + \sum_{i \in \text{segments}} \psi_{i,\text{pa}(i)}(m_i, m_{\text{pa}(i)}) .$$

Dynamic programming can find the best match, because the relational model is a tree.

It is inconvenient to search a continuous space, and we will discretize the space of image rectangles. We do so by discretizing the location and orientation of rectangles of fixed size (the sizes may be different for different segments). Because ankles and knees are different, we need to distinguish between a rectangle and the same rectangle rotated by 180° . One could visualize the result as a set of very large stacks of small rectangles of image, cut out at different locations and orientations. There is one stack per segment. We must now find the best allocation of rectangles to segments. This will be slow, because there are many image rectangles and, for the model we have given, choosing the right torso will be $O(M^6)$ if there are M image rectangles. However, various speedups are available for an appropriate choice of ψ , and the method is practical (Figure 24.23). The model is usually known as a **pictorial structure model**.

Recall our assumption that we know what we need to know about what the person looks like. If we are matching a person in a single image, the most useful feature for scoring segment matches turns out to be color. Texture features don't work well in most cases, because folds on loose clothing produce strong shading patterns that overlay the image texture. These

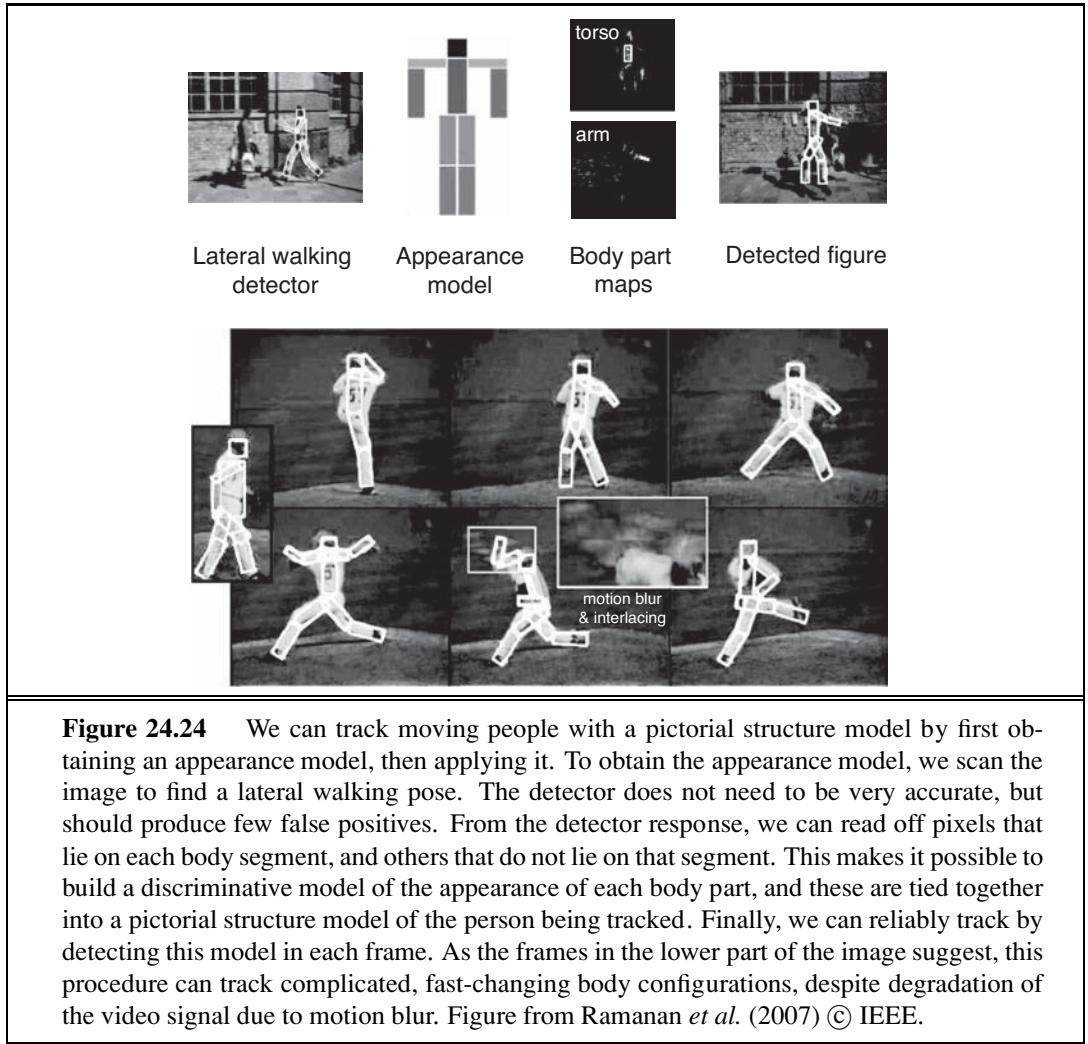


APPEARANCE MODEL

patterns are strong enough to disrupt the true texture of the cloth. In current work, ψ typically reflects the need for the ends of the segments to be reasonably close together, but there are usually no constraints on the angles. Generally, we don't know what a person looks like, and must build a model of segment appearances. We call the description of what a person looks like the **appearance model**. If we must report the configuration of a person in a single image, we can start with a poorly tuned appearance model, estimate configuration with this, then re-estimate appearance, and so on. In video, we have many frames of the same person, and this will reveal their appearance.

24.5.2 Coherent appearance: Tracking people in video

Tracking people in video is an important practical problem. If we could reliably report the location of arms, legs, torso, and head in video sequences, we could build much improved game interfaces and surveillance systems. Filtering methods have not had much success with this problem, because people can produce large accelerations and move quite fast. This means that for 30 Hz video, the configuration of the body in frame i doesn't constrain the configuration of the body in frame $i+1$ all that strongly. Currently, the most effective methods exploit the fact that appearance changes very slowly from frame to frame. If we can infer an appearance model of an individual from the video, then we can use this information in a pictorial structure model to detect that person in each frame of the video. We can then link these locations across time to make a track.



There are several ways to infer a good appearance model. We regard the video as a large stack of pictures of the person we wish to track. We can exploit this stack by looking for appearance models that explain many of the pictures. This would work by detecting body segments in each frame, using the fact that segments have roughly parallel edges. Such detectors are not particularly reliable, but the segments we want to find are special. They will appear at least once in most of the frames of video; such segments can be found by clustering the detector responses. It is best to start with the torso, because it is big and because torso detectors tend to be reliable. Once we have a torso appearance model, upper leg segments should appear near the torso, and so on. This reasoning yields an appearance model, but it can be unreliable if people appear against a near-fixed background where the segment detector generates lots of false positives. An alternative is to estimate appearance for many of the frames of video by repeatedly reestimating configuration and appearance; we then see if one appearance model explains many frames. Another alternative, which is quite



Figure 24.25 Some complex human actions produce consistent patterns of appearance and motion. For example, drinking involves movements of the hand in front of the face. The first three images are correct detections of drinking; the fourth is a false-positive (the cook is looking into the coffee pot, but not drinking from it). Figure from Laptev and Perez (2007) © IEEE.

reliable in practice, is to apply a detector for a fixed body configuration to all of the frames. A good choice of configuration is one that is easy to detect reliably, and where there is a strong chance the person will appear in that configuration even in a short sequence (lateral walking is a good choice). We tune the detector to have a low false positive rate, so we know when it responds that we have found a real person; and because we have localized their torso, arms, legs, and head, we know what these segments look like.

24.6 USING VISION

If vision systems could analyze video and understood what people are doing, we would be able to: design buildings and public places better by collecting and using data about what people do in public; build more accurate, more secure, and less intrusive surveillance systems; build computer sports commentators; and build human-computer interfaces that watch people and react to their behavior. Applications for reactive interfaces range from computer games that make a player get up and move around to systems that save energy by managing heat and light in a building to match where the occupants are and what they are doing.

Some problems are well understood. If people are relatively small in the video frame, and the background is stable, it is easy to detect the people by subtracting a background image from the current frame. If the absolute value of the difference is large, this **background subtraction** declares the pixel to be a foreground pixel; by linking foreground blobs over time, we obtain a track.

Structured behaviors like ballet, gymnastics, or tai chi have specific vocabularies of actions. When performed against a simple background, videos of these actions are easy to deal with. Background subtraction identifies the major moving regions, and we can build HOG features (keeping track of flow rather than orientation) to present to a classifier. We can detect consistent patterns of action with a variant of our pedestrian detector, where the orientation features are collected into histogram buckets over time as well as space (Figure 24.25).

More general problems remain open. The big research question is to link observations of the body and the objects nearby to the goals and intentions of the moving people. One source of difficulty is that we lack a simple vocabulary of human behavior. Behavior is a lot

like color, in that people tend to think they know a lot of behavior names but can't produce long lists of such words on demand. There is quite a lot of evidence that behaviors combine—you can, for example, drink a milkshake while visiting an ATM—but we don't yet know what the pieces are, how the composition works, or how many composites there might be. A second source of difficulty is that we don't know what features expose what is happening. For example, knowing someone is close to an ATM may be enough to tell that they're visiting the ATM. A third difficulty is that the usual reasoning about the relationship between training and test data is untrustworthy. For example, we cannot argue that a pedestrian detector is safe simply because it performs well on a large data set, because that data set may well omit important, but rare, phenomena (for example, people mounting bicycles). We wouldn't want our automated driver to run over a pedestrian who happened to do something unusual.

24.6.1 Words and pictures

Many Web sites offer collections of images for viewing. How can we find the images we want? Let's suppose the user enters a text query, such as "bicycle race." Some of the images will have keywords or captions attached, or will come from Web pages that contain text near the image. For these, image retrieval can be like text retrieval: ignore the images and match the image's text against the query (see Section 22.3 on page 867).

However, keywords are usually incomplete. For example, a picture of a cat playing in the street might be tagged with words like "cat" and "street," but it is easy to forget to mention the "garbage can" or the "fish bones." Thus an interesting task is to annotate an image (which may already have a few keywords) with additional appropriate keywords.

In the most straightforward version of this task, we have a set of correctly tagged example images, and we wish to tag some test images. This problem is sometimes known as auto-annotation. The most accurate solutions are obtained using nearest-neighbors methods. One finds the training images that are closest to the test image in a feature space metric that is trained using examples, then reports their tags.

Another version of the problem involves predicting which tags to attach to which regions in a test image. Here we do not know which regions produced which tags for the training data. We can use a version of expectation maximization to guess an initial correspondence between text and regions, and from that estimate a better decomposition into regions, and so on.

24.6.2 Reconstruction from many views

Binocular stereopsis works because for each point we have four measurements constraining three unknown degrees of freedom. The four measurements are the (x, y) positions of the point in each view, and the unknown degrees of freedom are the (x, y, z) coordinate values of the point in the scene. This rather crude argument suggests, correctly, that there are geometric constraints that prevent most pairs of points from being acceptable matches. Many images of a set of points should reveal their positions unambiguously.

We don't always need a second picture to get a second view of a set of points. If we believe the original set of points comes from a familiar rigid 3D object, then we might have

an object model available as a source of information. If this object model consists of a set of 3D points or of a set of pictures of the object, and if we can establish point correspondences, we can determine the parameters of the camera that produced the points in the original image. This is very powerful information. We could use it to evaluate our original hypothesis that the points come from an object model. We do this by using some points to determine the parameters of the camera, then projecting model points in this camera and checking to see whether there are image points nearby.

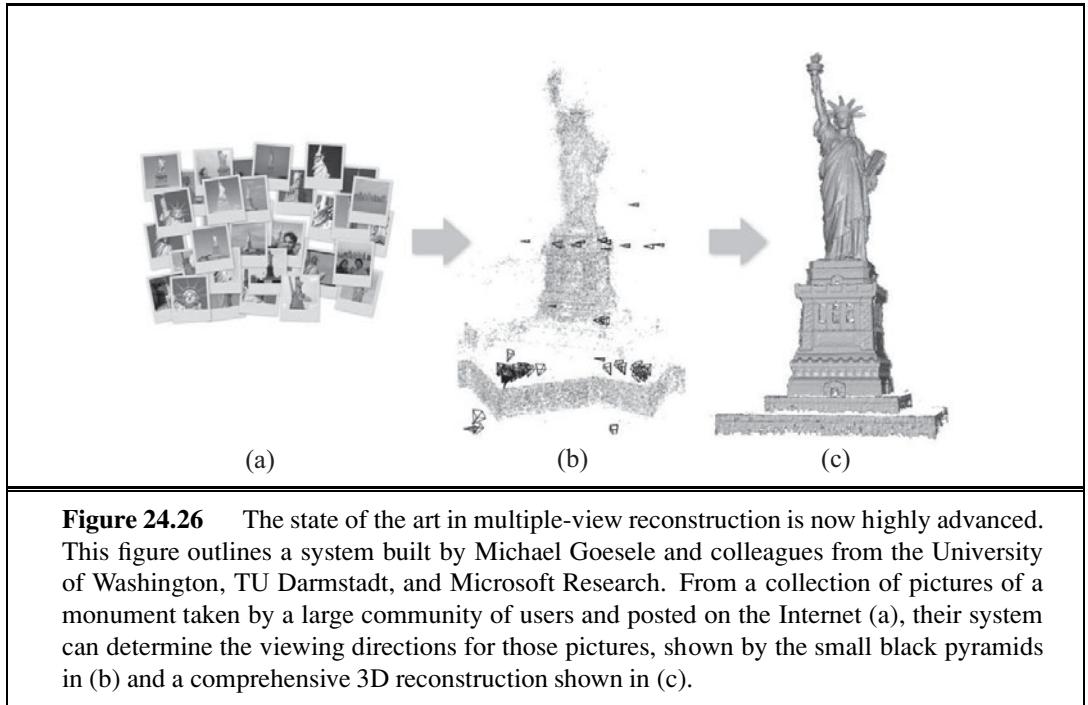
We have sketched here a technology that is now very highly developed. The technology can be generalized to deal with views that are not orthographic; to deal with points that are observed in only some views; to deal with unknown camera properties like focal length; to exploit various sophisticated searches for appropriate correspondences; and to do reconstruction from very large numbers of points and of views. If the locations of points in the images are known with some accuracy and the viewing directions are reasonable, very high accuracy camera and point information can be obtained. Some applications are

- **Model-building:** For example, one might build a modeling system that takes a video sequence depicting an object and produces a very detailed three-dimensional mesh of textured polygons for use in computer graphics and virtual reality applications. Models like this can now be built from apparently quite unpromising sets of pictures. For example, Figure 24.26 shows a model of the Statue of Liberty built from pictures found on the Internet.
- **Matching moves:** To place computer graphics characters into real video, we need to know how the camera moved for the real video, so that we can render the character correctly.
- **Path reconstruction:** Mobile robots need to know where they have been. If they are moving in a world of rigid objects, then performing a reconstruction and keeping the camera information is one way to obtain a path.

24.6.3 Using vision for controlling movement

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 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.”

Let us consider a vision system for an automated vehicle driving on a freeway. The tasks faced by the driver include the following:



1. Lateral control—ensure that the vehicle remains securely within its lane or changes lanes 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. 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. If we have good detailed maps of the road, then the vision system serves to confirm our position (and to watch for obstacles that are not on the map).

For longitudinal control, one needs to know distances to the vehicles in front. This can be accomplished with binocular stereopsis or optical flow. Using these techniques, vision-controlled cars can now drive reliably at highway speeds.

The more general case of mobile robots navigating in various indoor and outdoor environments has been studied, too. One particular problem, localizing the robot in its environment, now has pretty good solutions. A group at Sarnoff has developed a system based on two cameras looking forward that track feature points in 3D and use that to reconstruct the

position of the robot relative to the environment. In fact, they have two stereoscopic camera systems, one looking front and one looking back—this gives greater robustness in case the robot has to go through a featureless patch due to dark shadows, blank walls, and the like. It is unlikely that there are no features either in the front or in the back. Now of course, that could happen, so a backup is provided by using an inertial motion unit (IMU) somewhat akin to the mechanisms for sensing acceleration that we humans have in our inner ears. By integrating the sensed acceleration twice, one can keep track of the change in position. Combining the data from vision and the IMU is a problem of probabilistic evidence fusion and can be tackled using techniques, such as Kalman filtering, we have studied elsewhere in the book.

In the use of visual odometry (estimation of change in position), as in other problems of odometry, there is the problem of “drift,” positional errors accumulating over time. The solution for this is to use landmarks to provide absolute position fixes: as soon as the robot passes a location in its internal map, it can adjust its estimate of its position appropriately. Accuracies on the order of centimeters have been demonstrated with the these techniques.



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. Instead, a vision system should compute just what is needed to accomplish the task.

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 various 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 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

The eye developed in the Cambrian explosion (530 million years ago), apparently in a common ancestor. Since then, endless variations have developed in different creatures, but the same gene, Pax-6, regulates the development of the eye in animals as diverse as humans, mice, and *Drosophila*.

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 use of perspective in art was developed in ancient Roman culture, as evidenced by art found in the ruins of Pompeii (A.D. 79), but was then largely lost for 1300 years. 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). Stork (2004) analyzes the creation of various pieces of Renaissance art using computer vision techniques.

Although perspective was known to the ancient 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 Abu Ali Alhazen, in the 10th century. Alhazen also developed the *camera obscura*, a room (*camera* is Latin for “room” or “chamber”) with a pinhole that casts an image on the opposite wall. Of course 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). Kepler first proposed that the lens of the eye focuses an image on the retina, and Descartes surgically removed an ox eye and demonstrated that Kepler was right. There was still puzzlement as to why we do not see everything upside down; today we realize it is just a question of accessing the retinal data structure in the right way.

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. They pointed out the importance of perceptual organization: for a human observer, the image is not a collection of pointillist photoreceptor outputs (pixels in computer vision terminology); rather it is organized into coherent groups. One could trace the motivation in computer vision of finding regions and curves back to this insight. The Gestaltists also drew attention to the “figure-ground” phenomenon—a contour separating two image regions that, in the world, are at different depths, appears to belong only to the nearer region, the “figure,” and not the farther

region, the “ground.” The computer vision problem of classifying image curves according to their significance in the scene can be thought of as a generalization of this insight.

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 emphasized the role of the active observer whose self-directed movement facilitates the pickup of information about the external environment.

Computer vision was founded in the 1960s. Roberts's (1963) thesis at MIT was one of the earliest publications in the field, introducing key ideas such as edge detection and model-based matching. There is an urban legend that Marvin Minsky assigned the problem of “solving” computer vision to a graduate student as a summer project. According to Minsky the legend is untrue—it was actually an undergraduate student. But it was an exceptional undergraduate, Gerald Jay Sussman (who is now a professor at MIT) and the task was not to “solve” vision, but to investigate some aspects of it.

In the 1960s and 1970s, progress was slow, hampered considerably by the lack of computational and storage resources. Low-level visual processing received a lot of attention. The widely used Canny edge-detection technique was introduced in Canny (1986). Techniques for finding texture boundaries based on multiscale, multiorientation filtering of images date to work such as Malik and Perona (1990). Combining multiple clues—brightness, texture and color—for finding boundary curves in a learning framework was shown by Martin, Fowlkes and Malik (2004) to considerably improve performance.

The closely related problem of finding regions of coherent brightness, color, and texture, naturally lends itself to formulations in which finding the best partition becomes an optimization problem. Three leading examples are the Markov Random Fields approach of Geman and Geman (1984), the variational formulation of Mumford and Shah (1989), and normalized cuts by Shi and Malik (2000).

Through much of the 1960s, 1970s and 1980s, there were two distinct paradigms in which visual recognition was pursued, dictated by different perspectives on what was perceived to be the primary problem. Computer vision research on object recognition largely focused on issues arising from the projection of three-dimensional objects onto two-dimensional images. The idea of alignment, also first introduced by Roberts, resurfaced in the 1980s in the work of Lowe (1987) and Huttenlocher and Ullman (1990). Also popular was an approach based on describing shapes in terms of volumetric primitives, with **generalized cylinders**, introduced by Tom Binford (1971), proving particularly popular.

In contrast, the pattern recognition community viewed the 3D-to-2D aspects of the problem as not significant. Their 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.

In the late 1990s, these two paradigms started to converge, as both sides adopted the probabilistic modeling and learning techniques that were becoming popular throughout AI. Two lines of work contributed significantly. One was research on face detection, such as that

of Rowley, Baluja and Kanade (1996), and of Viola and Jones (2002b) which demonstrated the power of pattern recognition techniques on clearly important and useful tasks. The other was the development of point descriptors, which enable one to construct feature vectors from parts of objects. This was pioneered by Schmid and Mohr (1996). Lowe's (2004) SIFT descriptor is widely used. The HOG descriptor is due to Dalal and Triggs (2005).

Ullman (1979) and Longuet-Higgins (1981) are influential early works in reconstruction from multiple images. Concerns about the stability of structure from motion were significantly allayed by the work of Tomasi and Kanade (1992) who showed that with the use of multiple frames shape could be recovered quite accurately. In the 1990s, with great increase in computer speed and storage, 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.

For single images, inferring shape from shading was first studied by Horn (1970), and Horn and Brooks (1989) present an extensive survey of the main papers from a period when this was a much-studied problem. Gibson (1950) was the first to propose texture gradients as a cue to shape, though a comprehensive analysis for curved surfaces first appears in Garding (1992) and Malik and Rosenholtz (1997). The mathematics of occluding contours, and more generally understanding the visual events in the projection of smooth curved objects, owes much to the work of Koenderink and van Doorn, which finds an extensive treatment in Koenderink's (1990) *Solid Shape*. In recent years, attention has turned to treating the problem of shape and surface recovery from a single image as a probabilistic inference problem, where geometrical cues are not modeled explicitly, but used implicitly in a learning framework. A good representative is the work of Hoiem, Efros, and Hebert (2008).

For the reader interested in human vision, Palmer (1999) provides the best comprehensive treatment; Bruce *et al.* (2003) is a shorter textbook. The books by Hubel (1988) and Rock (1984) are friendly introductions centered on neurophysiology and perception respectively. David Marr's book *Vision* (Marr, 1982) played a historical role in connecting computer vision to psychophysics and neurobiology. While many of his specific models haven't stood the test of time, the theoretical perspective from which each task is analyzed at an informational, computational, and implementation level is still illuminating.

For computer vision, the most comprehensive textbook is Forsyth and Ponce (2002). Trucco and Verri (1998) is a shorter account. Horn (1986) and Faugeras (1993) are two older and still useful textbooks.

The main journals for computer vision are *IEEE Transactions on Pattern Analysis and Machine Intelligence* and *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). Research with a machine learning component is also published in the NIPS (Neural Information Processing Systems) conference, and work on the interface with computer graphics often appears at the ACM SIGGRAPH (Special Interest Group in Graphics) conference.

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 Consider a picture of a white sphere floating in front of a black backdrop. The image curve separating white pixels from black pixels is sometimes called the “outline” of the sphere. Show that the outline of a sphere, viewed in a perspective camera, can be an ellipse. Why do spheres not look like ellipses to you?

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 z -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? Draw the contours of constant brightness in the projected image. Are the contours of equal brightness uniformly spaced?

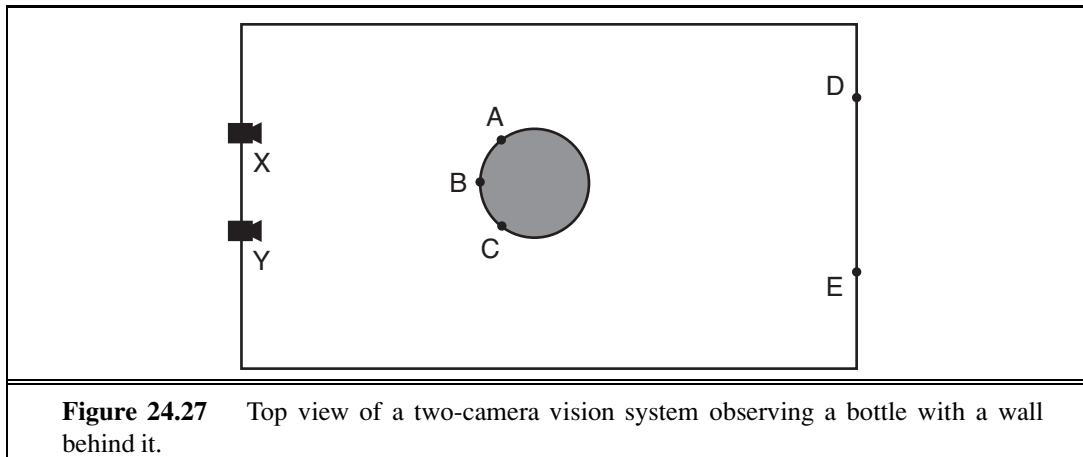
24.4 Edges in an image can correspond to a variety of events in a scene. Consider Figure 24.4 (page 933), and assume that it is a picture of a real three-dimensional scene. Identify ten different brightness edges in the image, and for each, state whether it corresponds to a discontinuity in (a) depth, (b) surface orientation, (c) reflectance, or (d) illumination.

24.5 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 lines from the object to the camera. The optical axes of the two cameras are parallel. The baseline between the cameras is 1 meter.

- a. If the nearest distance to be measured is 16 meters, what is the largest disparity that will occur (in pixels)?
- b. What is the distance resolution at 16 meters, due to the pixel spacing?
- c. What distance corresponds to a disparity of one pixel?

24.6 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. Lines with equal lengths in the scene always project to equal lengths in the image.
- d. Straight lines in the image necessarily correspond to straight lines in the scene.



24.7 (Courtesy of Pietro Perona.) Figure 24.27 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?

25 ROBOTICS

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

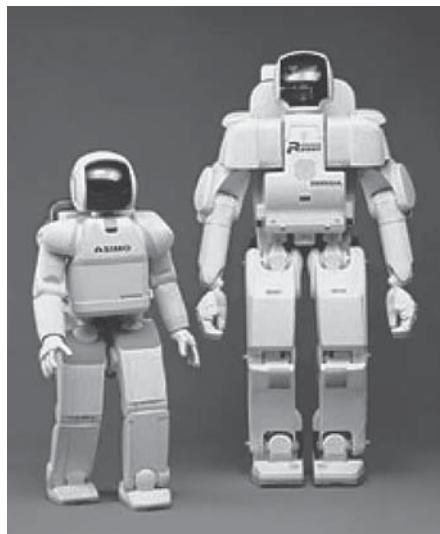
25.1 INTRODUCTION

ROBOT	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 lasers to measure the environment, and gyroscopes and accelerometers to measure the robot's own motion.
EFFECTOR	
SENSOR	
MANIPULATOR	Most of today's robots fall into one of three primary categories. Manipulators , or robot arms (Figure 25.1(a)), are physically anchored to their workplace, for example in a factory assembly line or on the International Space Station. Manipulator motion usually involves a 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 approximately one 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.
MOBILE ROBOT	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. Unmanned ground vehicles , or UGVs, drive autonomously on streets, highways, and off-road. The planetary rover shown in Figure 25.2(b) explored Mars for a period of 3 months in 1997. Subsequent NASA robots include the twin Mars Exploration Rovers (one is depicted on the cover of this book), which landed in 2003 and were still operating six years later. Other types of mobile robots include unmanned air vehicles (UAVs), commonly used for surveillance, crop-spraying, and
UGV	
PLANETARY ROVER	
UAV	

¹ In Chapter 2 we talked about **actuators**, not effectors. Here we distinguish the effector (the physical device) from the actuator (the control line that communicates a command to the effector).



(a)

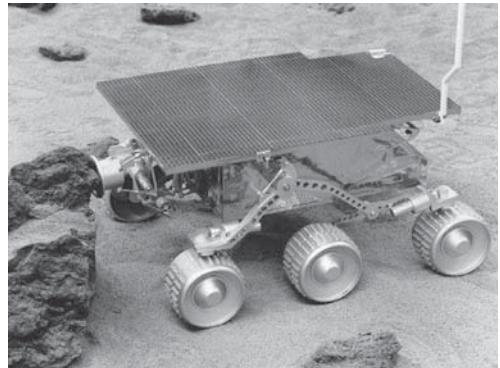


(b)

Figure 25.1 (a) An industrial robotic manipulator for stacking bags on a pallet. Image courtesy of Nachi Robotic Systems. (b) Honda's P3 and Asimo humanoid robots.



(a)



(b)

Figure 25.2 (a) Predator, an unmanned aerial vehicle (UAV) used by the U.S. Military. Image courtesy of General Atomics Aeronautical Systems. (b) NASA's Sojourner, a mobile robot that explored the surface of Mars in July 1997.

military operations. Figure 25.2(a) shows a UAV commonly used by the U.S. military. **Autonomous underwater vehicles** (AUVs) are used in deep sea exploration. Mobile robots deliver packages in the workplace and vacuum the floors at home.

The third type of robot combines mobility with manipulation, and is often called a **mobile manipulator**. **Humanoid robots** mimic the human torso. Figure 25.1(b) shows two early humanoid robots, both manufactured by Honda Corp. in Japan. Mobile manipulators

AUV

MOBILE
MANIPULATOR
HUMANOID ROBOT

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 must cope with environments that are partially observable, stochastic, dynamic, and continuous. Many robot environments are sequential and multiagent as well. Partial observability and stochasticity are the result of dealing with a large, complex world. Robot cameras 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.

Robotics brings together many of the concepts we have seen earlier in the book, including probabilistic state estimation, perception, planning, unsupervised learning, and reinforcement learning. For some of these concepts robotics serves as a challenging example application. For other concepts this chapter breaks new ground in introducing the continuous version of techniques that we previously saw only in the discrete case.

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.

25.2.1 Sensors

PASSIVE SENSOR

Sensors are the perceptual interface between robot and environment. **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 sense the environment, the robot's location, or the robot's internal configuration.

ACTIVE SENSOR

RANGE FINDER

SONAR SENSORS

Range finders are sensors that measure the distance to nearby objects. In the early days of robotics, robots were commonly equipped with **sonar sensors**. Sonar sensors emit directional sound waves, which are reflected by objects, with some of the sound making it

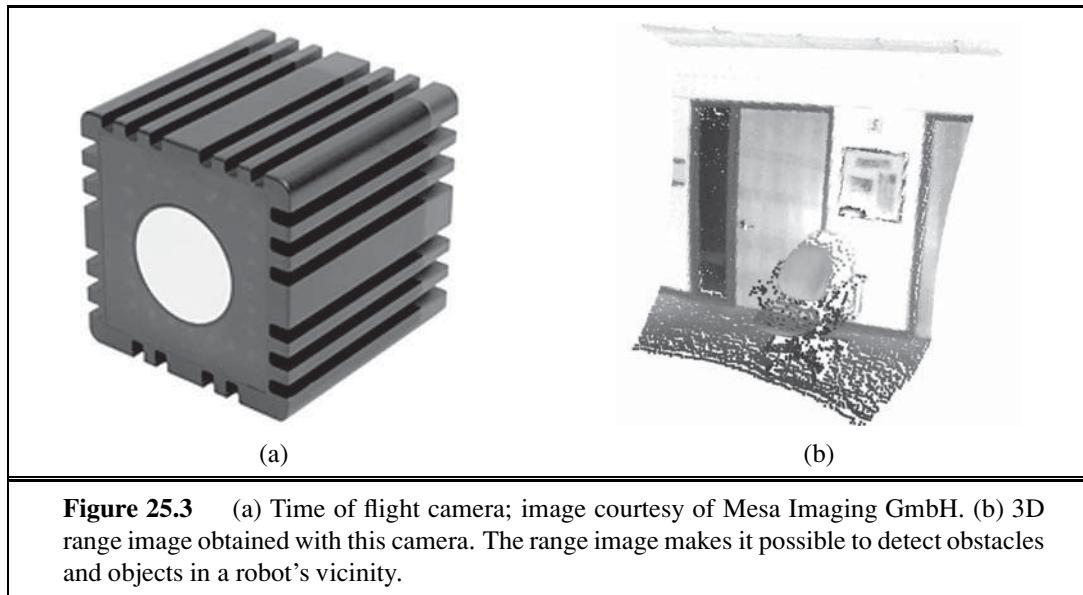


Figure 25.3 (a) Time of flight camera; image courtesy of Mesa Imaging GmbH. (b) 3D range image obtained with this camera. The range image makes it possible to detect obstacles and objects in a robot’s vicinity.

back into the sensor. The time and intensity of the returning signal indicates the distance to nearby objects. Sonar is the technology of choice for autonomous underwater vehicles.

STEREO VISION

Stereo vision (see Section 24.4.2) relies on multiple cameras to image the environment from slightly different viewpoints, analyzing the resulting parallax in these images to compute the range of surrounding objects. For mobile ground robots, sonar and stereo vision are now rarely used, because they are not reliably accurate.

TIME OF FLIGHT CAMERA

Most ground robots are now equipped with optical range finders. Just like sonar sensors, optical range sensors emit active signals (light) and measure the time until a reflection of this signal arrives back at the sensor. Figure 25.3(a) shows a **time of flight camera**. This camera acquires range images like the one shown in Figure 25.3(b) at up to 60 frames per second. Other range sensors use laser beams and special 1-pixel cameras that can be directed using complex arrangements of mirrors or rotating elements. These sensors are called **scanning lidars** (short for *light detection and ranging*). Scanning lidars tend to provide longer ranges than time of flight cameras, and tend to perform better in bright daylight.

SCANNING LIDARS

Other common range sensors include radar, which is often the sensor of choice for UAVs. Radar sensors can measure distances of multiple kilometers. On the other extreme end of range sensing are **tactile sensors** such as whiskers, bump panels, and touch-sensitive skin. These sensors measure range based on physical contact, and can be deployed only for sensing objects very close to the robot.

TACTILE SENSORS

A second important class of sensors is **location sensors**. Most location sensors use range sensing as a primary component to determine location. Outdoors, the **Global Positioning System** (GPS) is the most common solution to the localization problem. GPS measures the distance to satellites that emit pulsed signals. At present, there are 31 satellites in orbit, transmitting signals on multiple frequencies. GPS receivers can recover the distance to these satellites by analyzing phase shifts. By triangulating signals from multiple satellites, GPS

LOCATION SENSORS

GLOBAL POSITIONING SYSTEM

DIFFERENTIAL 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. Indoors, localization is often achieved by attaching beacons in the environment at known locations. Many indoor environments are full of wireless base stations, which can help robots localize through the analysis of the wireless signal. Underwater, active sonar beacons can provide a sense of location, using sound to inform AUVs of their relative distances to those beacons.

PROPRIOCEPTIVE SENSOR

SHAFT DECODER

ODOMETRY

INERTIAL SENSOR

FORCE SENSOR

TORQUE SENSOR

DEGREE OF FREEDOM

KINEMATIC STATE

POSE

DYNAMIC STATE

The third important class is **proprioceptive sensors**, which inform the robot of its own motion. 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 traveled. 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, rely on the resistance of mass to the change of velocity. They can help reduce uncertainty.

Other important aspects of robot state are measured by **force sensors** 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 all three translational and three rotational directions. They do this at a frequency of several hundred times a second, so that a robot can quickly detect unexpected forces and correct its actions before it breaks a light bulb.

25.2.2 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 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 mobile 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 these six plus an additional six dimensions for the rate of change of each kinematic dimension, that is, their velocities.

For nonrigid bodies, there are additional degrees of freedom within the robot itself. For example, the elbow of a human arm possesses two degrees of freedom. It can flex the upper arm towards or away, and can rotate right or left. The wrist has three degrees of freedom. It can move up and down, side to side, and can also rotate. Robot joints also have one, two, or three 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 25.4(a)

² “Kinematic” is from the Greek word for *motion*, as is “cinema.”

REVOLUTE JOINT
PRISMATIC JOINT

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 extra degrees of freedom are easier to control than robots with only the minimum number of DOFs. Many industrial manipulators therefore have seven DOFs, not six.

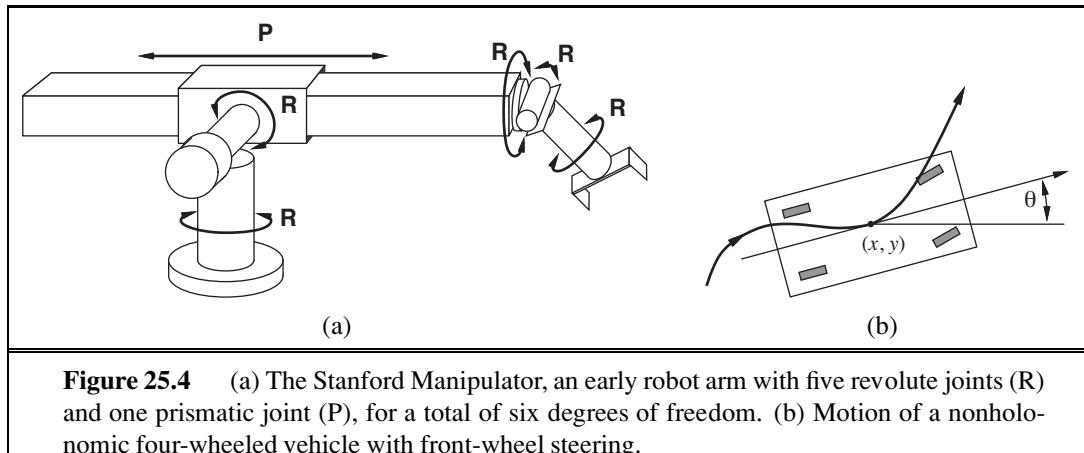


Figure 25.4 (a) The Stanford Manipulator, an early robot arm with five revolute joints (R) and one prismatic joint (P), for a total of six degrees of freedom. (b) Motion of a nonholonomic four-wheeled vehicle with front-wheel steering.

EFFECTIVE DOF
CONTROLLABLE DOF
NONHOLONOMIC

DIFFERENTIAL DRIVE
SYNCHRO DRIVE

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 25.4(b).) Thus, the car has three **effective degrees of freedom** but two **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.

Mobile robots have 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. To avoid chaos, the wheels are tightly coordinated. When moving straight, for example, all wheels 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 have three or more wheels that can be oriented and moved independently.

Some mobile robots possess arms. Figure 25.5(a) displays a two-armed robot. This robot's arms use springs to compensate for gravity, and they provide minimal resistance to

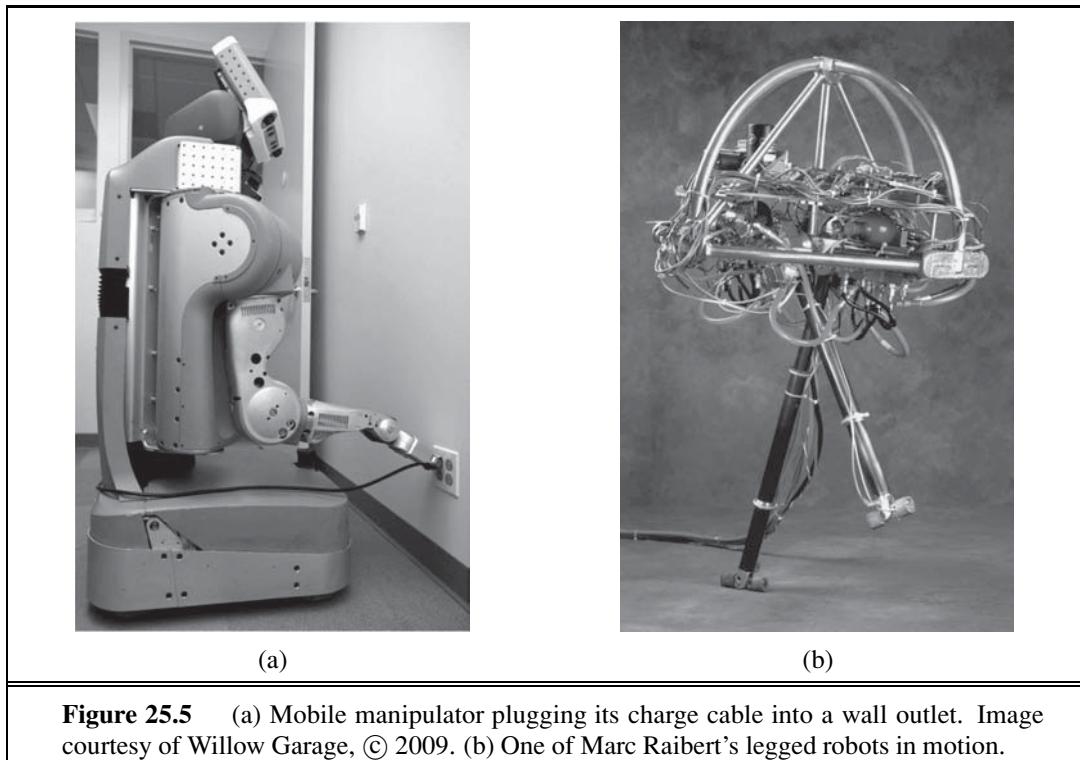


Figure 25.5 (a) Mobile manipulator plugging its charge cable into a wall outlet. Image courtesy of Willow Garage, © 2009. (b) One of Marc Raibert’s legged robots in motion.

external forces. Such a design minimizes the physical danger to people who might stumble into such a robot. This is a key consideration in deploying robots in domestic environments.

Legs, unlike wheels, can handle 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 25.5(b). 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. The quadruped (four-legged) robot shown in Figure 25.6(a) may appear statically stable. However, it walks by lifting multiple legs at the same time, which renders it dynamically stable. The robot can walk on snow and ice, and it will not fall over even if you kick it (as demonstrated in videos available online). Two-legged robots such as those in Figure 25.6(b) are dynamically stable.

Other methods of movement are possible: air vehicles use propellers or turbines; underwater vehicles use propellers or thrusters, similar to those used on submarines. Robotic blimps rely on thermal effects to keep themselves aloft.

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.

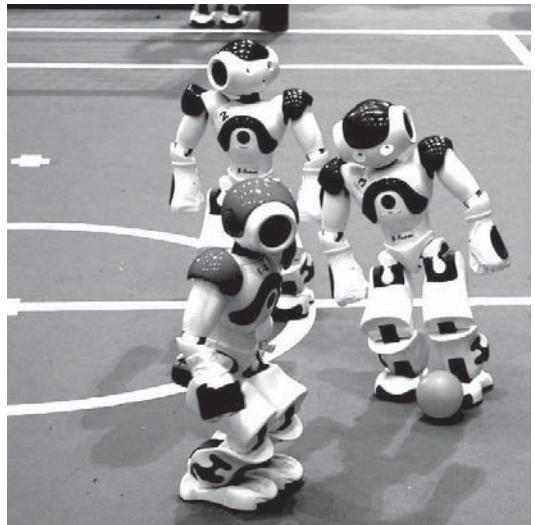
DYNAMICALLY
STABLE

STATICALLY STABLE

ELECTRIC MOTOR
PNEUMATIC
ACTUATION
HYDRAULIC
ACTUATION



(a)



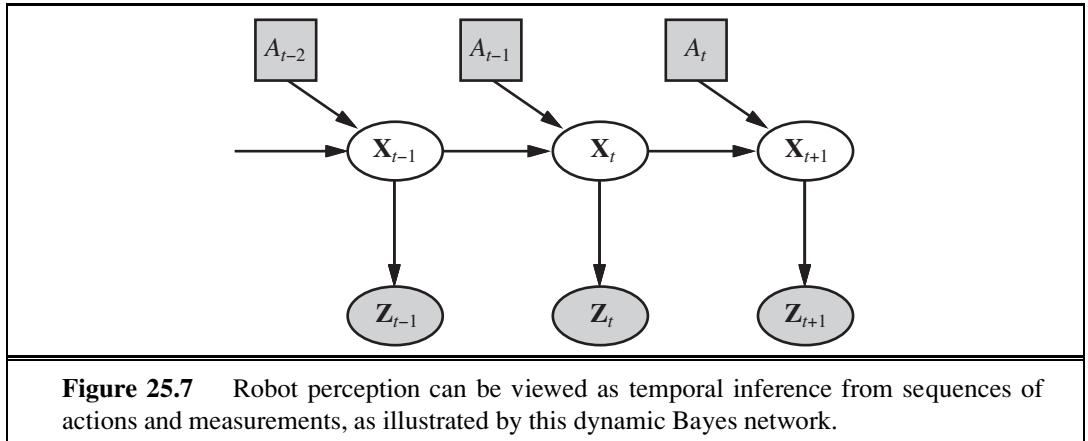
(b)

Figure 25.6 (a) Four-legged dynamically-stable robot “Big Dog.” Image courtesy Boston Dynamics, © 2009. (b) 2009 RoboCup Standard Platform League competition, showing the winning team, B-Human, from the DFKI center at the University of Bremen. Throughout the match, B-Human outscored their opponents 64:1. Their success was built on probabilistic state estimation using particle filters and Kalman filters; on machine-learning models for gait optimization; and on dynamic kicking moves. Image courtesy DFKI, © 2009.

25.3 ROBOTIC PERCEPTION

Perception is the process by which robots map sensor measurements into internal representations of the environment. Perception is difficult because sensors are noisy, and the environment is partially observable, unpredictable, and often dynamic. In other words, robots have all the problems of **state estimation** (or **filtering**) that we discussed in Section 15.2. As a rule of thumb, good internal representations for robots have three properties: they contain enough information for the robot to make good 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 include the robot’s own past actions as observed variables in the model. Figure 25.7 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.



We would like 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} . We did this in Section 15.2, but here there are two differences: we condition explicitly on the actions as well as the observations, and we deal with *continuous* rather than *discrete* variables. Thus, we modify the recursive filtering equation (15.5 on page 572) to use integration rather than summation:

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

This 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**.

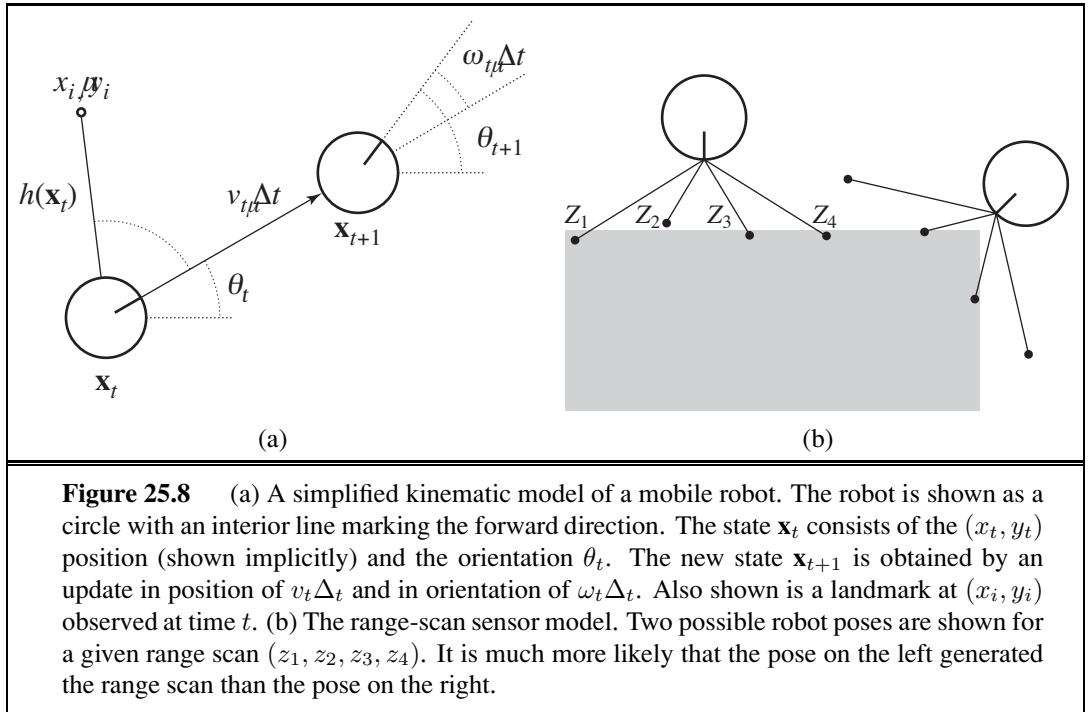
MOTION MODEL

25.3.1 Localization and mapping

LOCALIZATION

Localization is the problem of finding out where things are—including the robot itself. Knowledge about where things are is at the core of any successful physical interaction with the environment. For example, robot manipulators must know the location of objects they seek to manipulate; navigating robots must know where they are to find their way around.

To keep things simple, let us consider a mobile robot that moves slowly in a flat 2D world. Let us also assume the robot is given an exact map of the environment. (An example of such a map appears in Figure 25.10.) 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 25.8(a). If we arrange those three values in a vector, then any particular state is given by $\mathbf{X}_t = (x_t, y_t, \theta_t)^\top$. So far so good.



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.)

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

This probability distribution is the robot’s motion model. It models the effects of the motion a_t on the location of the robot.

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 25.8(a).) The exact prediction of the observed range and bearing would be

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

Again, noise distorts our measurements. To keep things simple, one might assume Gaussian noise with covariance Σ_z , giving us the sensor model

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

A somewhat different sensor model is used for an array of range sensors, each of which has a fixed bearing relative to the robot. Such sensors produce a vector of range values $\mathbf{z}_t = (z_1, \dots, z_M)^\top$. 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

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

Figure 25.8(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. 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 25.8(b), the robot faces a featureless wall. On the other hand, if there *are* visible, identifiable landmarks, they may provide instant 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 an instance of the particle-filtering algorithm of Figure 15.17 (page 598). All we need to do is supply the appropriate motion model and sensor model. Figure 25.9 shows one version using the range-scan model. The operation of the algorithm is illustrated in Figure 25.10 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 updating a filter is in general 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 25.11 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

```

function MONTE-CARLO-LOCALIZATION( $a, z, N, P(X'|X, v, \omega), P(z|z^*), m$ ) returns
  a set of samples for the next time step
  inputs:  $a$ , robot velocities  $v$  and  $\omega$ 
             $z$ , range scan  $z_1, \dots, z_M$ 
             $P(X'|X, v, \omega)$ , motion model
             $P(z|z^*)$ , range sensor noise model
             $m$ , 2D map of the environment
  persistent:  $S$ , a vector of samples of size  $N$ 
  local variables:  $W$ , a vector of weights of size  $N$ 
                     $S'$ , a temporary vector of particles of size  $N$ 
                     $W'$ , a vector of weights of size  $N$ 

  if  $S$  is empty then      /* initialization phase */
    for  $i = 1$  to  $N$  do
       $S[i] \leftarrow$  sample from  $P(X_0)$ 
    for  $i = 1$  to  $N$  do    /* update cycle */
       $S'[i] \leftarrow$  sample from  $P(X'|X = S[i], v, \omega)$ 
       $W'[i] \leftarrow 1$ 
      for  $j = 1$  to  $M$  do
         $z^* \leftarrow \text{RAYCAST}(j, X = S'[i], m)$ 
         $W'[i] \leftarrow W'[i] \cdot P(z_j | z^*)$ 
     $S \leftarrow \text{WEIGHTED-SAMPLE-WITH-REPLACEMENT}(N, S', W')$ 
  return  $S$ 

```

Figure 25.9 A Monte Carlo localization algorithm using a range-scan sensor model with independent noise.

TAYLOR EXPANSION

is called (first degree) **Taylor expansion**. A Kalman filter that linearizes f and h via Taylor expansion is called an **extended Kalman filter** (or EKF). Figure 25.12 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 and 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 25.10(b). The problem of needing to know the identity of landmarks is an instance of the **data association** problem discussed in Figure 15.6.

In some situations, no map of the environment is available. Then the robot will have to acquire a map. This is a bit of a chicken-and-egg problem: the navigating robot will have to determine its location relative to a map it doesn't quite know, at the same time building this map while it doesn't quite know its actual location. This problem is important for many robot applications, and it has been studied extensively under the name **simultaneous localization and mapping**, abbreviated as **SLAM**.

SLAM problems are solved using many different probabilistic techniques, including the extended Kalman filter discussed above. Using the EKF is straightforward: just augment

SIMULTANEOUS
LOCALIZATION AND
MAPPING

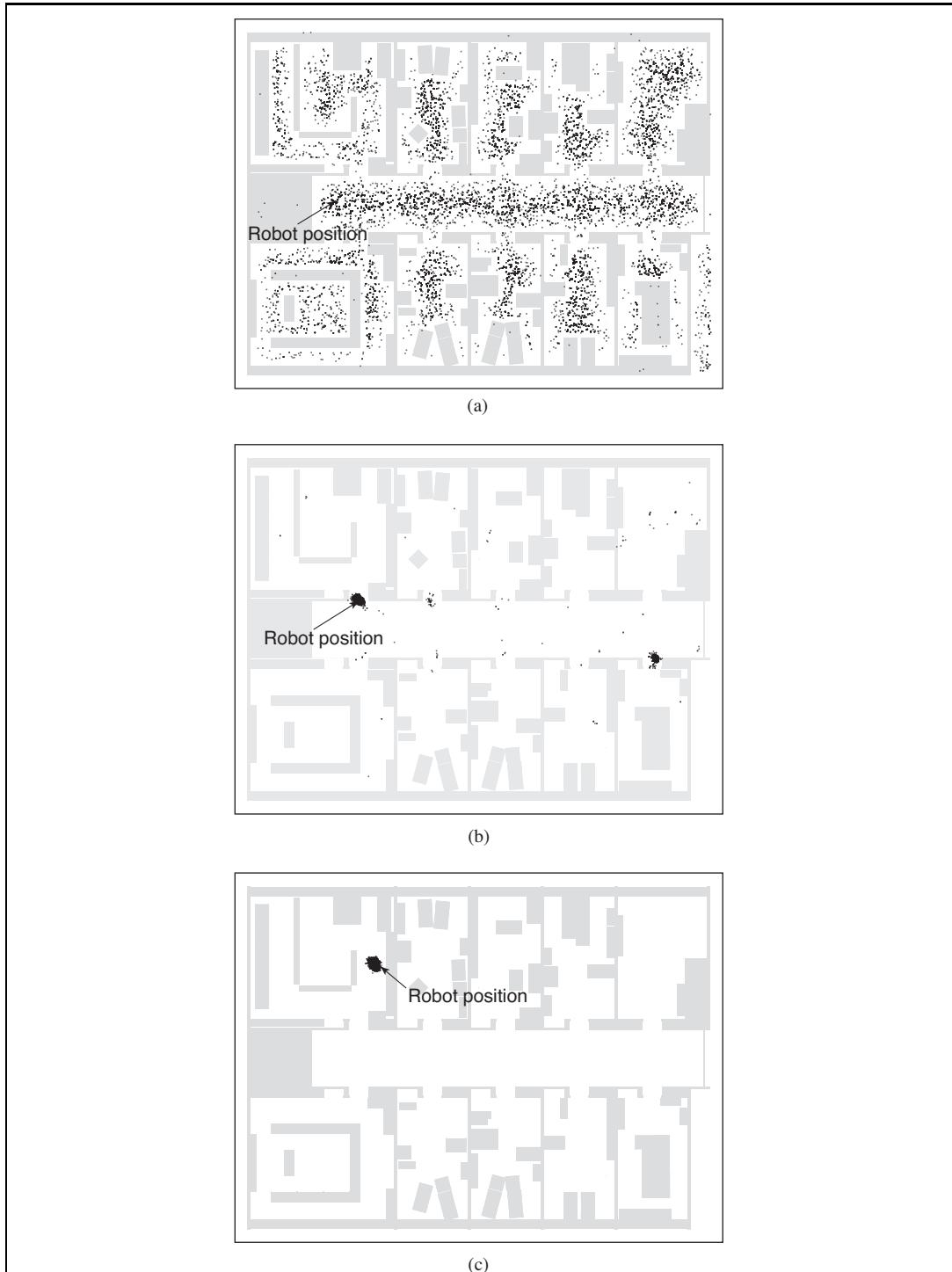


Figure 25.10 Monte Carlo localization, a particle filtering algorithm for mobile robot localization. (a) Initial, global uncertainty. (b) Approximately bimodal uncertainty after navigating in the (symmetric) corridor. (c) Unimodal uncertainty after entering a room and finding it to be distinctive.

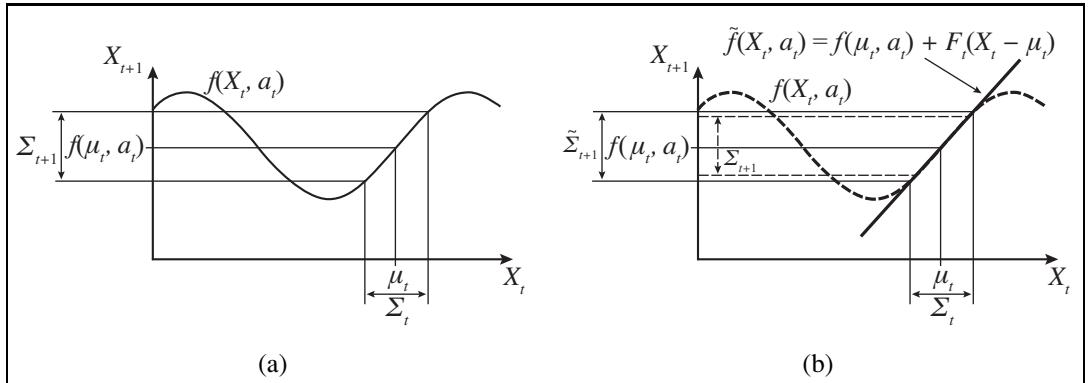
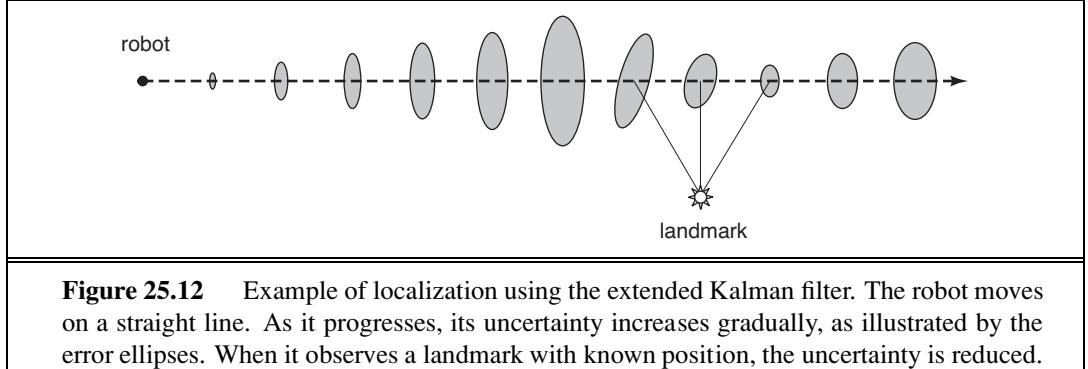


Figure 25.11 One-dimensional illustration of a linearized motion model: (a) The function f , and the projection of a mean μ_t and a covariance interval (based on Σ_t) into time $t + 1$. (b) The linearized version is the tangent of f at μ_t . The projection of the mean μ_t is correct. However, the projected covariance $\tilde{\Sigma}_{t+1}$ differs from Σ_{t+1} .



the state vector to include the locations of the landmarks in the environment. Luckily, the EKF update scales quadratically, so for small maps (e.g., a few hundred landmarks) the computation is quite feasible. Richer maps are often obtained using graph relaxation methods, similar to the Bayesian network inference techniques discussed in Chapter 14. Expectation–maximization is also used for SLAM.

25.3.2 Other types of perception

Not all of robot perception is about localization or mapping. Robots also perceive the temperature, odors, acoustic signals, and so on. Many of these quantities can be estimated using variants of dynamic Bayes networks. All that is required for such estimators are conditional probability distributions that characterize the evolution of state variables over time, and sensor models that describe the relation of measurements to state variables.

It is also possible to program a robot as a reactive agent, without explicitly reasoning about probability distributions over states. We cover that approach in Section 25.6.3.

The trend in robotics is clearly towards representations with well-defined semantics.

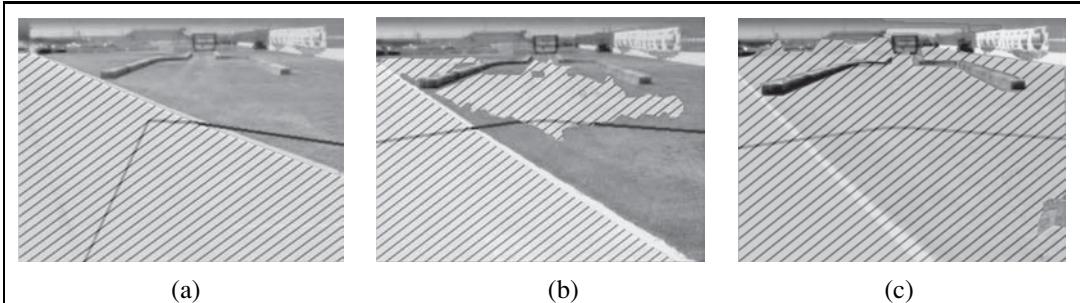


Figure 25.13 Sequence of “drivable surface” classifier results using adaptive vision. In (a) only the road is classified as drivable (striped area). The V-shaped dark line shows where the vehicle is heading. In (b) the vehicle is commanded to drive off the road, onto a grassy surface, and the classifier is beginning to classify some of the grass as drivable. In (c) the vehicle has updated its model of drivable surface to correspond to grass as well as road.

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 your best teacher.

25.3.3 Machine learning in robot perception

Machine learning plays an important role in robot perception. This is particularly the case when the best internal representation is not known. One common approach is to map high-dimensional sensor streams into lower-dimensional spaces using unsupervised machine learning methods (see Chapter 18). Such an approach is called **low-dimensional embedding**. Machine learning makes it possible to learn sensor and motion models from data, while simultaneously discovering a suitable internal representations.

Another machine learning technique enables robots to continuously adapt to broad changes in sensor measurements. Picture yourself walking from a sun-lit space into a dark neon-lit room. Clearly things are darker inside. But the change of light source also affects all the colors: Neon light has a stronger component of green light than sunlight. Yet somehow we seem not to notice the change. If we walk together with people into a neon-lit room, we don’t think that suddenly their faces turned green. Our perception quickly adapts to the new lighting conditions, and our brain ignores the differences.

Adaptive perception techniques enable robots to adjust to such changes. One example is shown in Figure 25.13, taken from the autonomous driving domain. Here an unmanned ground vehicle adapts its classifier of the concept “drivable surface.” How does this work? The robot uses a laser to provide classification for a small area right in front of the robot. When this area is found to be flat in the laser range scan, it is used as a positive training example for the concept “drivable surface.” A mixture-of-Gaussians technique similar to the EM algorithm discussed in Chapter 20 is then trained to recognize the specific color and texture coefficients of the small sample patch. The images in Figure 25.13 are the result of applying this classifier to the full image.

SELF-SUPERVISED
LEARNINGPOINT-TO-POINT
MOTION
COMPLIANT MOTION

PATH PLANNING

WORKSPACE
REPRESENTATIONLINKAGE
CONSTRAINTS

Methods that make robots collect their own training data (with labels!) are called **self-supervised**. In this instance, the robot uses machine learning to leverage a short-range sensor that works well for terrain classification into a sensor that can see much farther. That allows the robot to drive faster, slowing down only when the sensor model says there is a change in the terrain that needs to be examined more carefully by the short-range sensors.

25.4 PLANNING TO MOVE

All of a robot's deliberations ultimately come down to deciding how to move 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.

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; the complication added by robotics is that path planning involves *continuous* spaces. There are two main approaches: **cell decomposition** and **skeletonization**. Each reduces the continuous path-planning problem to a discrete graph-search problem. In this section, we assume that motion is deterministic and that localization of the robot is exact. Subsequent sections will relax these assumptions.

25.4.1 Configuration space

We will start with a simple representation for a simple robot motion problem. Consider the robot arm shown in Figure 25.14(a). It has two joints that move independently. 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

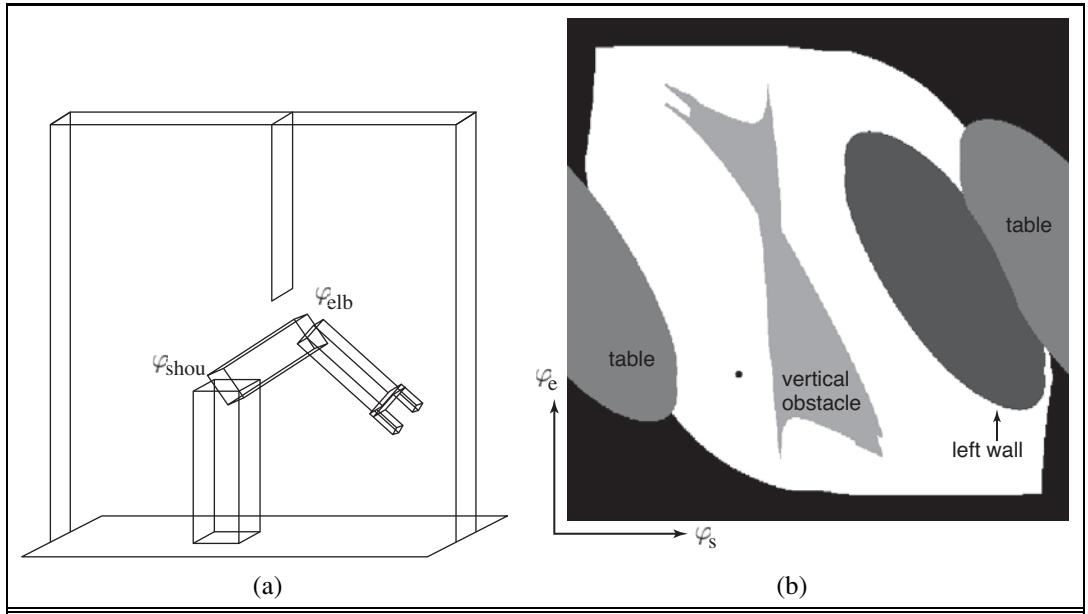


Figure 25.14 (a) Workspace representation of a robot arm with 2 DOFs. The workspace is a box with a flat obstacle hanging from the ceiling. (b) Configuration space of the same robot. Only white regions in the space are configurations that are free of collisions. The dot in this diagram corresponds to the configuration of the robot shown on the left.

CONFIGURATION SPACE

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 configuration and the target configuration by a straight line. In following this path, the robot would then move its joints at a constant velocity, until a target location is reached.

KINEMATICS

INVERSE KINEMATICS

Unfortunately, configuration spaces have their own problems. The task of a robot is usually expressed in workspace coordinates, not in configuration space coordinates. This raises the question of how to map between workspace coordinates and configuration space. Transforming configuration space coordinates into workspace coordinates is simple: it involves a series of straightforward coordinate transformations. These transformations are linear for prismatic joints and trigonometric for revolute joints. This chain of coordinate transformation is known as **kinematics**.

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 hard, especially for robots with many DOFs. In particular, the solution is seldom unique. Figure 25.14(a) shows one of two possible configurations that put the gripper in the same location. (The other configuration would have the elbow below the shoulder.)

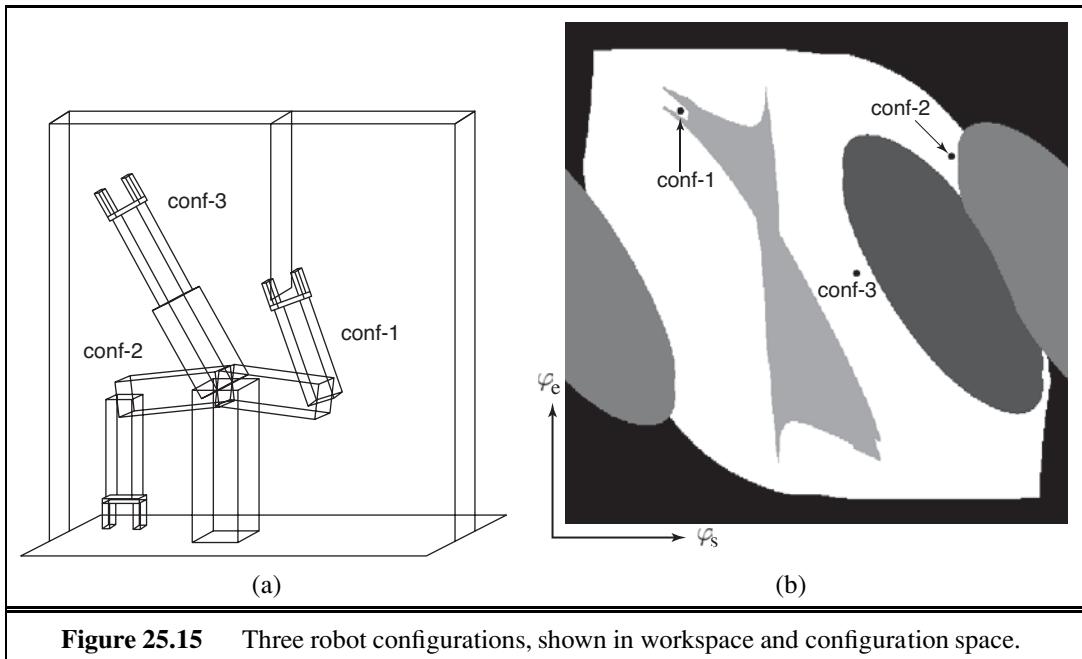


Figure 25.15 Three robot configurations, shown in workspace and configuration space.

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 sufficient degrees of freedom to find infinitely many solutions to motion problems. To see how this is possible, simply imagine that we added a third revolute joint to our example robot, one whose rotational axis is parallel to the ones of the existing joints. In such a case, we can keep the location (but not the orientation!) of the gripper fixed and still freely rotate its 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 of the gripper 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 25.14(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 25.14(b) shows the configuration space for our example robot, under the specific obstacle configuration shown in Figure 25.14(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 25.14(b) corresponds to the free space. All other regions correspond to occu-

FREE SPACE

OCCUPIED SPACE

pied space. The different shadings 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 sides of the robot correspond to the table on which the robot is mounted. The third oval region corresponds to the left wall. Finally, the most interesting object in configuration space is the vertical obstacle that hangs from the ceiling and impedes the robot's motions. This object has a funny shape in configuration space: 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 diagram. The shape of this obstacle is not at all obvious! The dot inside Figure 25.14(b) marks the configuration of the robot, as shown in Figure 25.14(a). Figure 25.15 depicts three additional configurations, both in workspace and in configuration space. In configuration conf-1, the gripper encloses the vertical obstacle.

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.

25.4.2 Cell decomposition methods

CELL DECOMPOSITION

The 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 25.16(a) shows a square grid decomposition of the space and a solution path that is optimal for this grid size. Grayscale shading indicates 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 on page 653.) Figure 25.16(b) shows the corresponding workspace trajectory for the arm. Of course, we can also use the A* algorithm to find a shortest path.

Such a decomposition has the advantage that it is extremely simple to implement, but it also suffers from three limitations. First, it is workable only for low-dimensional configuration spaces, because the number of grid cells increases exponentially with d , the number of dimensions. Sounds familiar? This is the curse!dimensionality@of dimensionality. 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

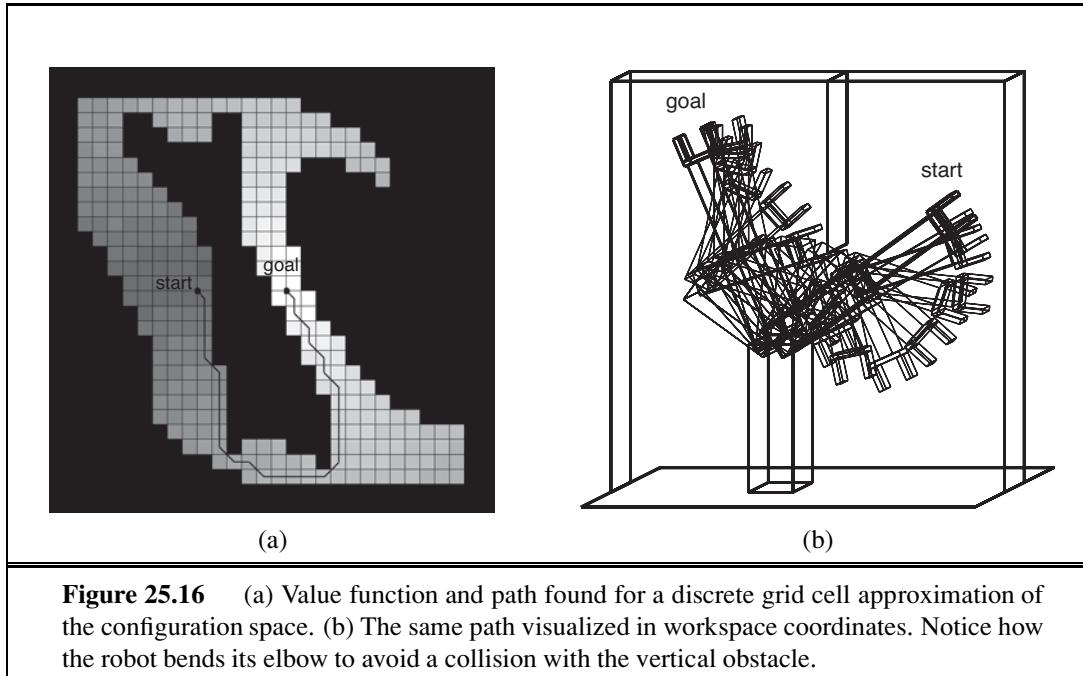


Figure 25.16 (a) Value function and path found for a discrete grid cell approximation of the configuration space. (b) The same path visualized in workspace coordinates. Notice how the robot bends its elbow to avoid a collision with the vertical obstacle.

be the case that the only paths to the goal go through mixed cells—especially if the cell size is comparable to that of the passageways and clearances in the space. And third, any path through a discretized state space will not be smooth. It is generally difficult to guarantee that a smooth solution exists near the discrete path. So a robot may not be able to execute the solution found through this decomposition.

Cell decomposition methods can be improved in a number of ways, to alleviate some of these problems. The first approach allows *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 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.

EXACT CELL
DECOMPOSITION

Examining the solution path shown in Figure 25.16(a), we can see an additional difficulty that will have to be resolved. The path contains arbitrarily sharp corners; a robot moving at any finite speed could not execute such a path. This problem is solved by storing certain continuous values for each grid cell. Consider an algorithm which stores, for each grid cell,

HYBRID A*

the exact, continuous state that was attained with the cell was first expanded in the search. Assume further, that when propagating information to nearby grid cells, we use this continuous state as a basis, and apply the continuous robot motion model for jumping to nearby cells. In doing so, we can now guarantee that the resulting trajectory is smooth and can indeed be executed by the robot. One algorithm that implements this is **hybrid A***.

POTENTIAL FIELD

25.4.3 Modified cost functions

Notice that in Figure 25.16, the path goes very close to the obstacle. Anyone who has driven a car knows that a parking space 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.

This problem can be solved 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 25.17(a) shows such a potential field—the darker a configuration state, the closer it is to an obstacle.

The potential field can be used as an additional cost term in the shortest-path calculation. This induces an interesting tradeoff. On the one hand, the robot seeks to minimize path length to the goal. On the other hand, it tries to stay away from obstacles by virtue of minimizing the potential function. With the appropriate weight balancing the two objectives, a resulting path may look like the one shown in Figure 25.17(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.

There exist many other ways to modify the cost function. For example, it may be desirable to *smooth* the control parameters over time. For example, when driving a car, a smooth path is better than a jerky one. In general, such higher-order constraints are not easy to accommodate in the planning process, unless we make the most recent steering command a part of the state. However, it is often easy to smooth the resulting trajectory after planning, using conjugate gradient methods. Such post-planning smoothing is essential in many real-world applications.

SKELETONIZATION

25.4.4 Skeletonization methods

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 25.18 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.

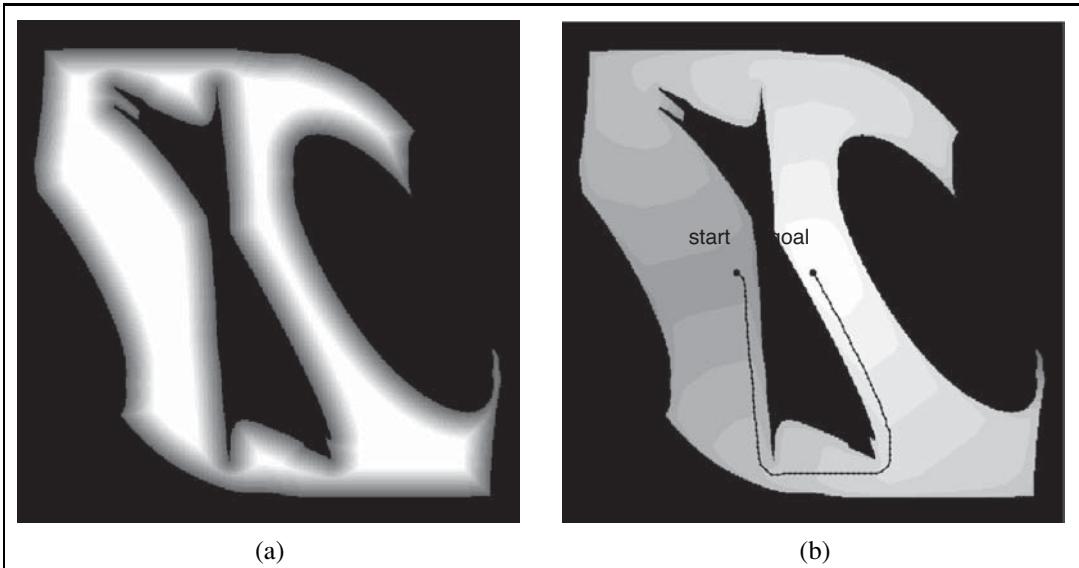


Figure 25.17 (a) A repelling potential field pushes the robot away from obstacles. (b) Path found by simultaneously minimizing path length and the potential.

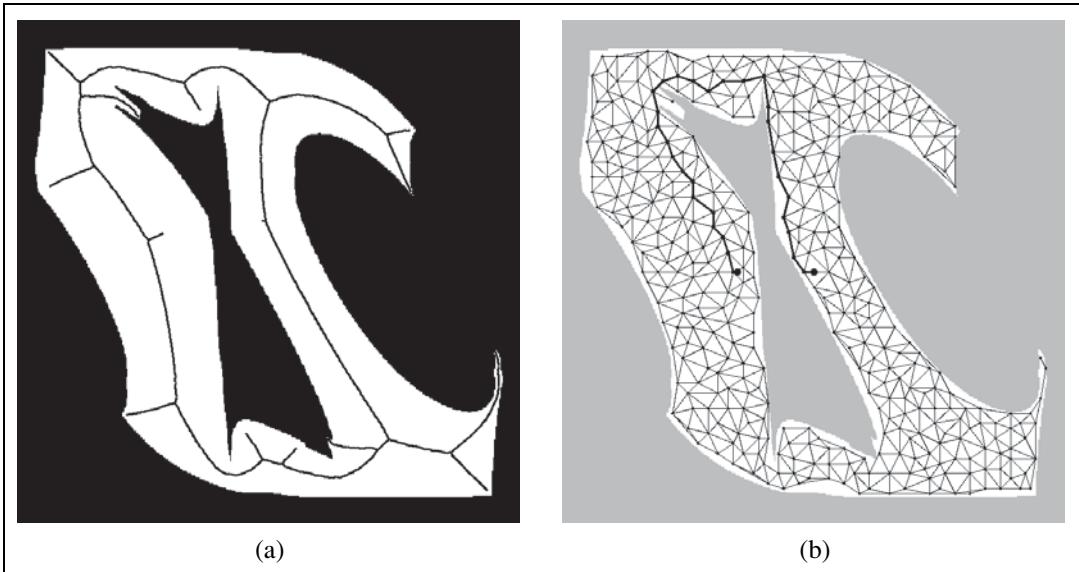


Figure 25.18 (a) The Voronoi graph is the set of points equidistant to two or more obstacles in configuration space. (b) A probabilistic roadmap, composed of 400 randomly chosen points in free space.

In this way, the original path-planning problem is reduced to finding a path on the Voronoi graph, which is generally one-dimensional (except in certain nongeneric 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 graph can be difficult, especially in configuration space, where the shapes of obstacles can be complex.

**PROBABILISTIC
ROADMAP**

An alternative to the Voronoi graphs is the **probabilistic roadmap**, a skeletonization approach that offers more possible routes, and thus deals better with wide-open spaces. Figure 25.18(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. Two nodes are joined 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 goal 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 goal. 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

MOST LIKELY STATE

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.

ONLINE REPLANNING

Most of today’s robots use deterministic algorithms for decision making, such as the path-planning algorithms of the previous section. To do so, it is common practice to extract the **most likely state** from the probability distribution produced by the state estimation 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. In fact, when the environment model changes over time as the result of incorporating sensor measurements, many robots plan paths online during plan execution. This is the **online replanning** technique of Section 11.3.3.

Unfortunately, ignoring the uncertainty does not always work. In some problems the robot's uncertainty is simply too massive: 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 faces uncertainty only in its state transition, but its state is fully observable, the problem is best modeled as a Markov decision process (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 called **navigation functions**. The value function shown in Figure 25.16(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 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 high-dimensional continuous spaces. Discretization produces POMDPs that are far too large to handle. One remedy is to make the minimization of uncertainty a control objective. For example, the **coastal navigation** heuristic requires the robot to stay near known landmarks to decrease its uncertainty. Another approach applies variants of the probabilistic roadmap planning method to the belief space representation. Such methods tend to scale better to large discrete POMDPs.

25.5.1 Robust methods

Uncertainty can also be handled using so-called **robust control** methods (see page 836) 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 11—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

NAVIGATION
FUNCTION

INFORMATION
GATHERING ACTION

COASTAL
NAVIGATION

ROBUST CONTROL

FINE-MOTION
PLANNING

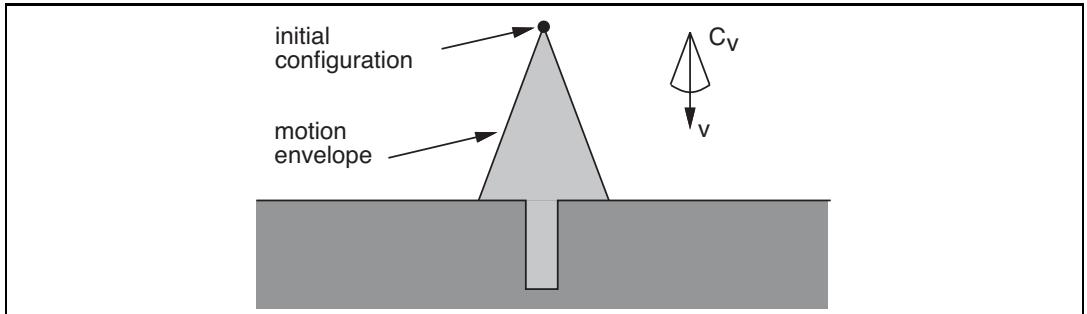


Figure 25.19 A two-dimensional environment, velocity uncertainty cone, and envelope of possible robot motions. The intended velocity is v , but with uncertainty the actual velocity could be anywhere in C_v , resulting in a final configuration somewhere in the motion envelope, which means we wouldn't know if we hit the hole or not.

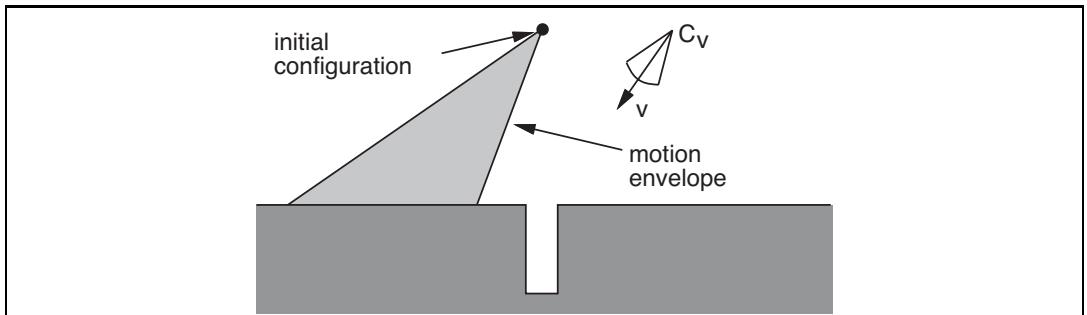


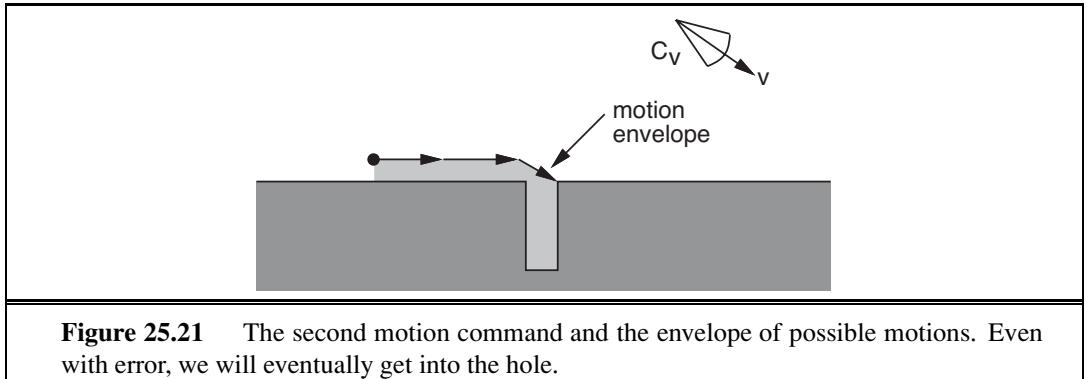
Figure 25.20 The first motion command and the resulting envelope of possible robot motions. No matter what the error, we know the final configuration will be to the left of the hole.

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.

GUARDED MOTION

COMPLIANT MOTION

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 effector to slide if the motion command would cause collision with an obstacle. As an example, Figure 25.19 shows a two-dimensional configuration space with a narrow vertical hole. It could be the configuration space for insertion of a rectangular peg into a hole or a car key into the ignition. 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 initial configuration. 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 25.20 and 25.21. In Figure 25.20, 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 25.21, a motion command is given that causes the robot to slide along the surface and into the hole. 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 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 optimal in the decision-theoretic sense only 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. 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.

25.6.1 Dynamics and control

Section 25.2 introduced the notion of **dynamic state**, which extends the kinematic state of a robot by its velocity. For example, in addition to the angle of a robot joint, the dynamic state also captures the rate of change of the angle, and possibly even its momentary acceleration. 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 has higher dimension than the kinematic space, and the curse of dimensionality would render many motion planning algorithms inapplicable for all but the most simple robots. For this reason, practical robot system often rely on simpler kinematic path planners.

DIFFERENTIAL EQUATION

CONTROLLER

REFERENCE CONTROLLER
REFERENCE PATH
OPTIMAL CONTROLLERS

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 continuous MDPs are, in effect, optimal controllers.

On the surface, the problem of keeping a robot on a prespecified path appears to be relatively straightforward. In practice, however, even this seemingly simple problem has its pitfalls. Figure 25.22(a) illustrates what can go wrong; it shows 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 counterforce to keep the robot on track. However,

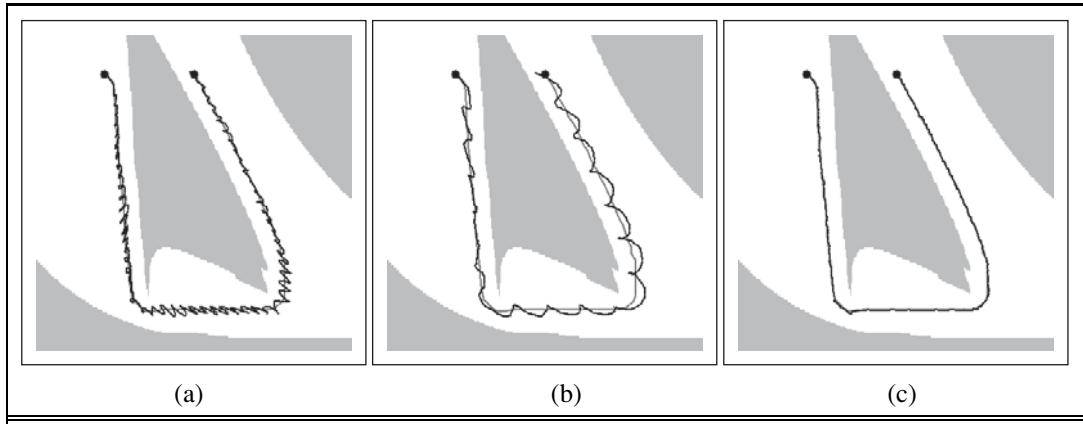


Figure 25.22 Robot arm control using (a) proportional control with gain factor 1.0, (b) proportional control with gain factor 0.1, and (c) PD (proportional derivative) control with gain factors 0.3 for the proportional component and 0.8 for the differential component. In all cases the robot arm tries to follow the path shown in gray.

as Figure 25.22(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. Such overshooting may continue along an entire trajectory, and the resulting robot motion is far from desirable.

Before we can define a better controller, let us formally describe what went wrong. 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 form:

$$a_t = K_P(y(t) - x_t).$$

P CONTROLLER

GAIN PARAMETER

STABLE

STRICTLY STABLE

Here x_t is the state of the robot at time t and K_P is a constant known as the **gain parameter** of the controller and its value is called the gain factor); K_p 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 would remedy the problem. Unfortunately, this is not the case. Figure 25.22(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 and

PD CONTROLLER

then stay on its reference path upon such perturbations. Our P controller appears to be stable but not strictly stable, since it fails to stay anywhere near its reference trajectory.

The simplest controller that achieves strict stability in our domain is a **PD controller**. The letter ‘P’ stands again for *proportional*, and ‘D’ stands for *derivative*. PD controllers are described by the following equation:

$$a_t = K_P(y(t) - x_t) + K_D \frac{\partial(y(t) - x_t)}{\partial t}. \quad (25.2)$$

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 this, 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 error will then counteract the proportional term, which will reduce the overall response to the perturbation. However, if the same error persists and does not change, the derivative will vanish and the proportional term dominates the choice of control.

Figure 25.22(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.

PD controllers do have failure modes, however. In particular, PD controllers may fail to regulate an error down to zero, even in the absence of external perturbations. Often such a situation is the result of a systematic external force that is not part of the model. An autonomous car driving on a banked surface, for example, may find itself systematically pulled to one side. Wear and tear in robot arms cause similar systematic errors. In such situations, an over-proportional feedback is required to drive the error closer to zero. The solution to this problem lies in adding a third term to the control law, based on the integrated error over time:

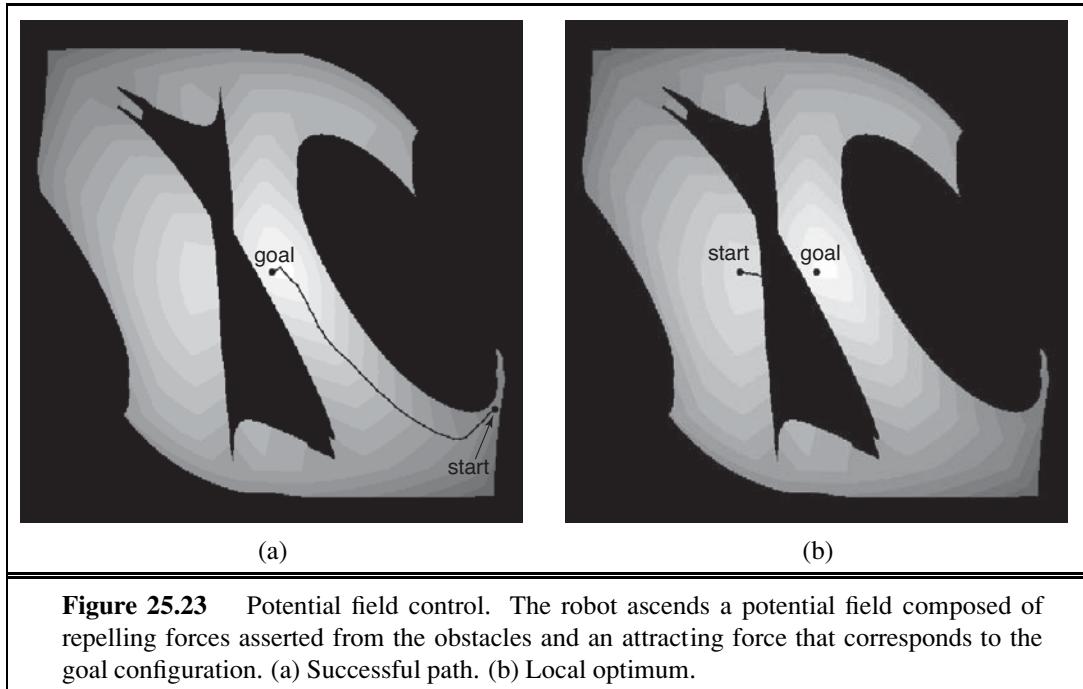
$$a_t = K_P(y(t) - x_t) + K_I \int (y(t) - x_t) dt + K_D \frac{\partial(y(t) - x_t)}{\partial t}. \quad (25.3)$$

PID CONTROLLER

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 expense of increased danger of oscillatory behavior. A controller with all three terms is called a **PID controller** (for proportional integral derivative). PID controllers are widely used in industry, for a variety of control problems.

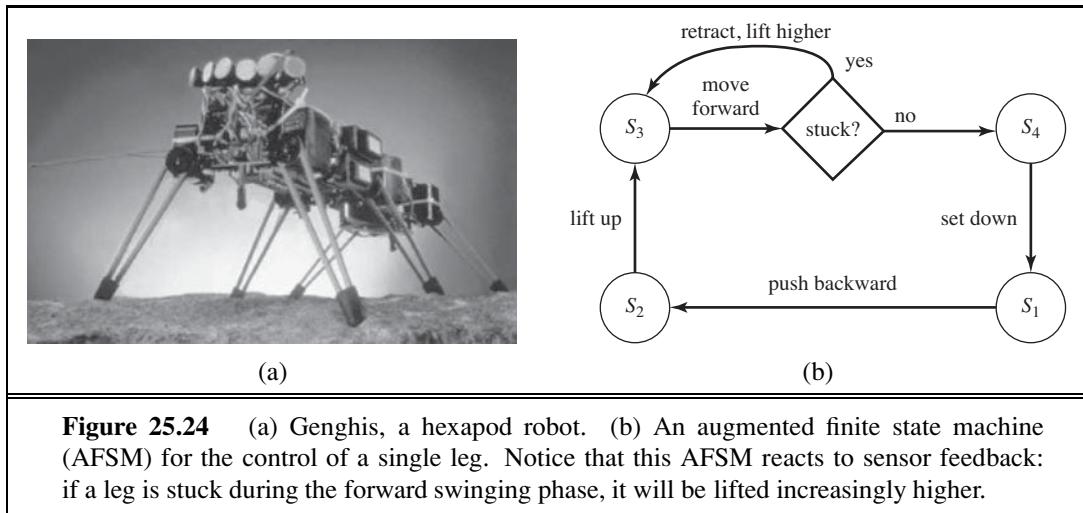
25.6.2 Potential-field control

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 25.23. Its single global minimum is



the goal configuration, and the value is the sum of the distance to this goal 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 25.23(a) shows a trajectory of a robot that performs hill climbing in the potential field. 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 can be extremely efficient, especially when compared to path-planning algorithms, all of which are exponential in the dimensionality of the configuration space (the DOFs) in the worst case.

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 Figure 25.23(b), the robot approaches 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 control is great for local robot motion but sometimes we still need 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.



25.6.3 Reactive control

So far we have considered 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, or for robots that have few sensors. 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 architecture using **reactive control** is more appropriate.

For example, picture a legged robot that attempts to lift a leg over an obstacle. We could give this robot a rule that says lift the leg a small height h and move it forward, and if the leg encounters an obstacle, move it back and start again at a higher height. You could say that h is modeling an aspect of the world, but we can also think of h as an auxiliary variable of the robot controller, devoid of direct physical meaning.

One such example is the six-legged (hexapod) robot, shown in Figure 25.24(a), designed for walking through rough terrain. The robot's sensors are inadequate to obtain models of the terrain for path planning. 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.

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 hexapod robot we first choose a **gait**, or pattern of movement of the limbs. One statically stable gait is to first move the right front, right rear, and left center legs forward (keeping the other three fixed), and then move the other three. This gait works well on flat terrain. On rugged terrain, obstacles may prevent a leg 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,*

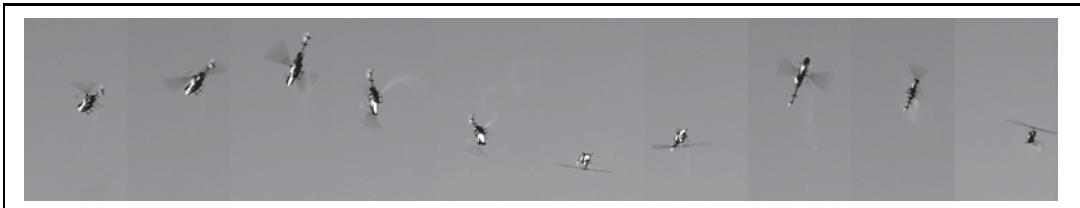


Figure 25.25 Multiple exposures of an RC helicopter executing a flip based on a policy learned with reinforcement learning. Images courtesy of Andrew Ng, Stanford University.

and try again. The resulting controller is shown in Figure 25.24(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 have been found to generate remarkably robust walking patterns, 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. Environmental feedback plays a crucial role in the controller's execution. 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 biological organisms.

EMERGENT BEHAVIOR

25.6.4 Reinforcement learning control

One particularly exciting form of control is based on the **policy search** form of reinforcement learning (see Section 21.5). This work has been enormously influential in recent years, as it has solved challenging robotics problems for which previously no solution existed. An example is acrobatic autonomous helicopter flight. Figure 25.25 shows an autonomous flip of a small RC (radio-controlled) helicopter. This maneuver is challenging due to the highly nonlinear nature of the aerodynamics involved. Only the most experienced of human pilots are able to perform it. Yet a policy search method (as described in Chapter 21), using only a few minutes of computation, learned a policy that can safely execute a flip every time.

Policy search needs an accurate model of the domain before it can find a policy. The input to this model is the state of the helicopter at time t , the controls at time t , and the resulting state at time $t + \Delta t$. The state of a helicopter can be described by the 3D coordinates of the vehicle, its yaw, pitch, and roll angles, and the rate of change of these six variables. The controls are the manual controls of the helicopter: throttle, pitch, elevator, aileron, and rudder. All that remains is the resulting state—how are we going to define a model that accurately says how the helicopter responds to each control? The answer is simple: Let an expert human pilot fly the helicopter, and record the controls that the expert transmits over the radio and the state variables of the helicopter. About four minutes of human-controlled flight suffices to build a predictive model that is sufficiently accurate to simulate the vehicle.

What is remarkable about this example is the ease with which this learning approach solves a challenging robotics problem. This is one of the many successes of machine learning in scientific fields previously dominated by careful mathematical analysis and modeling.

25.7 ROBOTIC SOFTWARE ARCHITECTURES

SOFTWARE ARCHITECTURE

A methodology for structuring algorithms is called a **software architecture**. An architecture 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 planning. In many ways, reactive and deliberate techniques have orthogonal strengths and weaknesses. Reactive control is sensor-driven and appropriate for making low-level decisions in real time. However, it 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 planning is a more appropriate choice.

Consequently, most robot architectures use reactive techniques at the lower levels of control and deliberative 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 called **hybrid architectures**.

HYBRID 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 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 25.24(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, does rely on sensor feedback. If the leg is stuck, meaning that it has failed to execute the forward swing, the robot retracts the leg, lifts it 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 its own problems. First, the AFSMs are 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 simple tasks, such as following a wall 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 goals (just like the dung beetle on page 39). 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 robotics, despite its great historical importance. However, it has had an influence on other architectures, and on individual components of some architectures.

25.7.2 Three-layer architecture

THREE-LAYER ARCHITECTURE

REACTIVE LAYER

EXECUTIVE LAYER

Hybrid architectures combine reaction with deliberation. The most popular hybrid architecture is the **three-layer architecture**, which consists of a reactive layer, an executive layer, and a deliberative 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 layer and the deliberative layer. It accepts directives by the deliberative layer, and sequences them for the reactive layer. For example, the executive layer might handle a set of via-points generated by a deliberative 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.

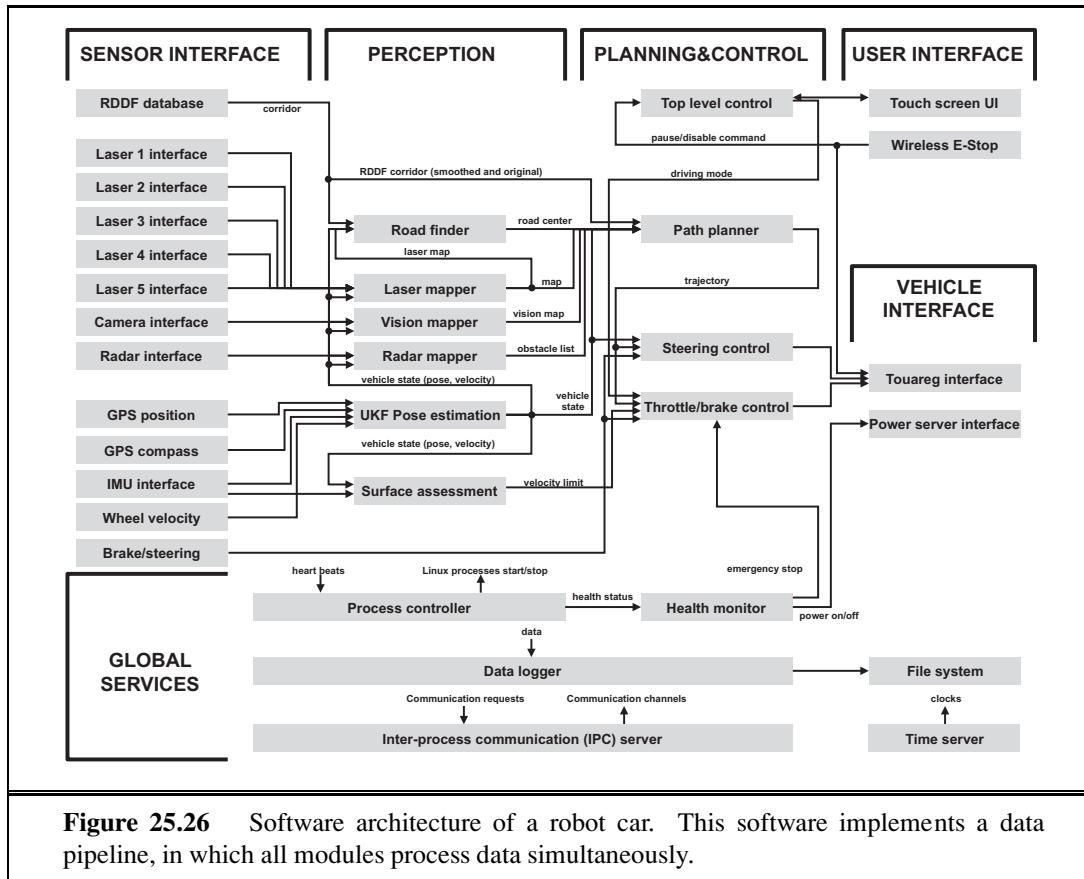


Figure 25.26 Software architecture of a robot car. This software implements a data pipeline, in which all modules process data simultaneously.

DELIBERATIVE LAYER

The **deliberative 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 deliberative layer (or planning layer) uses models for decision making. Those models might be either learned from data or supplied and may 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 a multiagent level for coordinating a robot's actions with that of other robots operating in the same environment.

25.7.3 Pipeline architecture

Pipeline Architecture

Another architecture for robots is known as the **pipeline architecture**. Just like the subsumption architecture, the pipeline architecture executes multiple process in parallel. However, the specific modules in this architecture resemble those in the three-layer architecture.

Figure 25.26 shows an example pipeline architecture, which is used to control an autonomous car. Data enters this pipeline at the **sensor interface layer**. The **perception layer**

SENSOR INTERFACE LAYER
PERCEPTION LAYER

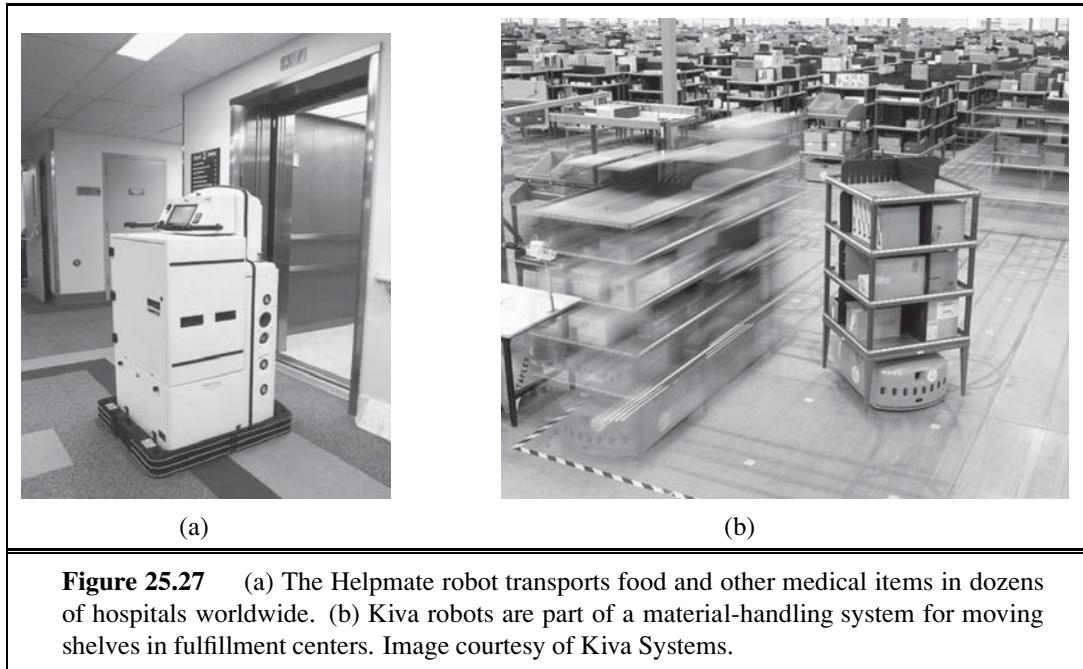


Figure 25.27 (a) The Helpmate robot transports food and other medical items in dozens of hospitals worldwide. (b) Kiva robots are part of a material-handling system for moving shelves in fulfillment centers. Image courtesy of Kiva Systems.

PLANNING AND
CONTROL LAYER

VEHICLE INTERFACE
LAYER

then updates the robot's internal models of the environment based on this data. Next, these models are handed to the **planning and control layer**, which adjusts the robot's internal plans turns them into actual controls for the robot. Those are then communicated back to the vehicle through the **vehicle interface layer**.

The key to the pipeline architecture is that this all happens in parallel. While the perception layer processes the most recent sensor data, the control layer bases its choices on slightly older data. In this way, the pipeline architecture is similar to the human brain. We don't switch off our motion controllers when we digest new sensor data. Instead, we perceive, plan, and act all at the same time. Processes in the pipeline architecture run asynchronously, and all computation is data-driven. The resulting system is robust, and it is fast.

The architecture in Figure 25.26 also contains other, cross-cutting modules, responsible for establishing communication between the different elements of the pipeline.

25.8 APPLICATION DOMAINS

Here are 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



(a)



(b)

Figure 25.28 (a) Robotic car BOSS, which won the DARPA Urban Challenge. Courtesy of Carnegie Mellon University. (b) Surgical robots in the operating room. Image courtesy of da Vinci Surgical Systems.

example, a project at Carnegie Mellon University 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 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 semimechanical work that is presently performed by people.

Transportation. Robotic transportation has many facets: from autonomous helicopters that deliver payloads to hard-to-reach locations, 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 25.27(a). This robot has been deployed in dozens of hospitals to transport food and other items. In factory settings, autonomous vehicles are now routinely deployed to transport goods in warehouses and between production lines. The Kiva system, shown in Figure 25.27(b), helps workers at fulfillment centers package goods into shipping containers.

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, or barcode tags. 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.

Robotic cars. Most of us use cars every day. Many of us make cell phone calls while driving. Some of us even text. The sad result: more than a million people die every year in traffic accidents. Robotic cars like BOSS and STANLEY offer hope: Not only will they make driving much safer, but they will also free us from the need to pay attention to the road during our daily commute.

Progress in robotic cars was stimulated by the DARPA Grand Challenge, a race over 100 miles of unrehearsed desert terrain, which represented a much more challenging task than

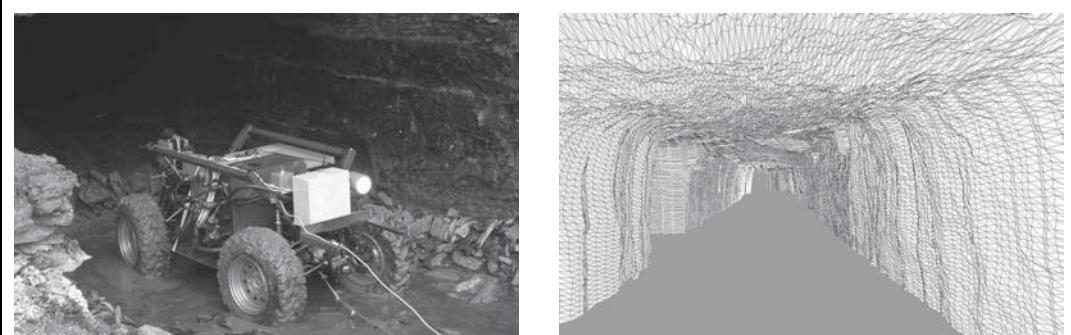


Figure 25.29 (a) A robot mapping an abandoned coal mine. (b) A 3D map of the mine acquired by the robot.

had ever been accomplished before. Stanford’s STANLEY vehicle completed the course in less than seven hours in 2005, winning a \$2 million prize and a place in the National Museum of American History. Figure 25.28(a) depicts BOSS, which in 2007 won the DARPA Urban Challenge, a complicated road race on city streets where robots faced other robots and had to obey traffic rules.

Health care. Robots are increasingly used to assist surgeons with instrument placement when operating on organs as intricate as brains, eyes, and hearts. Figure 25.28(b) shows such a system. Robots have become indispensable tools in a range of surgical procedures, such as hip replacements, thanks to their high precision. In pilot studies, robotic devices have been found to reduce the danger of lesions when performing colonoscopy. 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 and provide comfort. Researchers are also working on robotic devices for rehabilitation that aid people in performing certain exercises.

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.

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 25.2(b) and the cover). 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 25.29 shows a robot mapping an abandoned coal mine, along with a 3D model of the mine acquired



Figure 25.30 (a) Roomba, the world’s best-selling mobile robot, vacuums floors. Image courtesy of iRobot, © 2009. (b) Robotic hand modeled after human hand. Image courtesy of University of Washington and Carnegie Mellon University.

DRONE

using range sensors. In 1996, a team of researchers released a legged robot into the crater of an active volcano to acquire data 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) for people to access.

ROONBA

Personal Services. Service is an up-and-coming application domain of robotics. Service robots assist individuals in performing daily tasks. Commercially available domestic service robots include autonomous vacuum cleaners, lawn mowers, and golf caddies. The world’s most popular mobile robot is a personal service robot: the robotic vacuum cleaner **Roomba**, shown in Figure 25.30(a). More than three million Roombas have been sold. Roomba can navigate autonomously and perform its tasks without human help.

ROONBA

Other 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.

ROBOTIC SOCCER

Entertainment. Robots have begun to conquer the entertainment and toy industry. In Figure 25.6(b) we see **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 relevant to 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 of devices that make it easier for people to walk or move their arms by providing additional forces through extraskeletal attachments. If such devices are attached permanently,

they can be thought of as artificial robotic limbs. Figure 25.30(b) shows a robotic hand that may serve as a prosthetic device in the future.

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. Underwater vehicles are often teleoperated; the vehicles can go to a depth that would be dangerous for humans but can still be guided by the human operator. 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, mapping, and object recognition**.
- **Probabilistic filtering algorithms** such as Kalman filters and particle filters are useful for robot perception. These techniques maintain the belief state, 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 space 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 controller**. Controllers are necessary in robotics to accommodate small perturbations; path planning alone is usually insufficient.
- **Potential field** techniques navigate robots by potential functions, defined over the distance to obstacles and the goal 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**.

- There exist different architectures for software design. The **subsumption architecture** enables programmers to compose robot controllers from interconnected finite state machines. **Three-layer architectures** are common frameworks for developing robot software that integrate deliberation, sequencing of subgoals, and control. The related **pipeline architecture** processes data in parallel through a sequence of modules, corresponding to perception, modeling, planning, control, and robot interfaces.

BIBLIOGRAPHICAL AND HISTORICAL NOTES

The word **robot** was popularized by Czech playwright Karel Capek 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) it was Capek's brother, Josef, who first combined the Czech words "roboťa" (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 Hephaistos, 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 on page 14.

UNIMATE

The first commercial robot was a robot arm called **Unimate**, short for *universal automation*, 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 development 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 worldwide, more than half of which are installed in Japan.

PUMA

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. Shakey appears on the cover of this book with project leader Charlie Rosen (1917–2002). 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 by Leonard and Durrant-Whyte (1992); see Dissanayake *et al.* (2001) for an overview of early Kalman filter variations. 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). Kuipers and Levitt (1988) were among the first to propose topological rather than metric mapping, motivated by models of human spatial cognition. A seminal paper by Lu and Milios (1997) recognized the sparseness of the simultaneous localization and mapping problem, which gave rise to the development of nonlinear optimization techniques by Konolige (2004) and Montemerlo and Thrun (2004), as well as hierarchical methods by Bosse *et al.* (2004). Shatkay and Kaelbling (1997) and Thrun *et al.* (1998) introduced the EM algorithm into the field of robotic mapping for data association. An overview of probabilistic mapping methods can be found in (Thrun *et al.*, 2005).

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 (Dean *et al.*, 1990, 1990) and of 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).

The study of manipulator robots, 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). A series of papers by Schwartz and Sharir on what they called **piano movers** problems (Schwartz *et al.*, 1987) was highly influential.

Recursive cell decomposition for configuration space planning was originated by Brooks and Lozano-Perez (1985) and improved significantly by Zhu and Latombe (1991). The ear-

OCCUPANCY GRID

MARKOV LOCALIZATION

RAO-BLACKWELLIZED PARTICLE FILTER

HAND-EYE MACHINES

PIANO MOVERS

VISIBILITY GRAPH

liest 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 broader motion-planning problems. John Canny (1988) established the first singly exponential algorithm for motion planning. The seminal text by Latombe (1991) covers a variety of approaches to motion-planning, as do the texts by Choset *et al.* (2004) and LaValle (2006). Kavraki *et al.* (1996) developed probabilistic roadmaps, which are currently one of the most effective methods. Fine-motion planning with limited sensing was investigated by Lozano-Perez *et al.* (1984) and Canny and Reif (1987). Landmark-based navigation (Lazanas and Latombe, 1992) uses many of the same ideas in the mobile robot arena. Key work applying POMDP methods (Section 17.4) to motion planning under uncertainty in robotics is due to Pineau *et al.* (2003) and Roy *et al.* (2005).

GRASPING

HAPTIC FEEDBACK

VECTOR FIELD HISTOGRAMS

The control of robots as dynamical systems—whether for manipulation or navigation—has generated a huge literature that is barely touched on by this chapter. 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 classic textbooks 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).

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). Reinforcement learning in robotics took off with the seminal work by Bagnell and Schneider (2001) and Ng *et al.* (2004), who developed the paradigm in the context of autonomous helicopter control.

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 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 some of the most popular approaches in this field.

Research on mobile robotics has been stimulated over the last decade by several important competitions. The earliest competition, AAAI's annual mobile robot competition, began in 1992. The first competition winner was CARMEL (Congdon *et al.*, 1992). Progress has

ROBOCUP

been steady and impressive: in more recent competitions robots entered the conference complex, found their way to the registration desk, registered for the conference, and even gave a short talk. The **Robocup** competition, launched in 1995 by Kitano and colleagues (1997a), aims to “develop a team of fully autonomous humanoid robots that can win against the human world champion team in soccer” by 2050. Play occurs in leagues for simulated robots, wheeled robots of different sizes, and humanoid robots. In 2009 teams from 43 countries participated and the event was broadcast to millions of viewers. Visser and Burkhard (2007) track the improvements that have been made in perception, team coordination, and low-level skills over the past decade.

DARPA GRAND CHALLENGE

The **DARPA Grand Challenge**, organized by DARPA in 2004 and 2005, required autonomous robots to travel more than 100 miles through unrehearsed desert terrain in less than 10 hours (Buehler *et al.*, 2006). In the original event in 2004, no robot traveled more than 8 miles, leading many to believe the prize would never be claimed. In 2005, Stanford’s robot **STANLEY** won the competition in just under 7 hours of travel (Thrun, 2006). DARPA then organized the **Urban Challenge**, a competition in which robots had to navigate 60 miles in an urban environment with other traffic. Carnegie Mellon University’s robot **BOSS** took first place and claimed the \$2 million prize (Urmson and Whittaker, 2008). Early pioneers in the development of robotic cars included Dickmanns and Zapp (1987) and Pomerleau (1993).

URBAN CHALLENGE

Two early textbooks, by Dudek and Jenkin (2000) and Murphy (2000), cover robotics generally. A more recent overview is due to Bekey (2008). An excellent book on robot manipulation addresses advanced topics such as compliant motion (Mason, 2001). Robot motion planning is covered in Choset *et al.* (2004) and LaValle (2006). Thrun *et al.* (2005) provide an introduction into probabilistic robotics. The premiere conference for robotics is Robotics: Science and Systems Conference, followed by the IEEE International Conference on Robotics and Automation. Leading 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: $X = \{x_1, x_2, x_3, x_4\}$. 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 X . Let Z be a Boolean sensor variable characterized by the following conditional probabilities:

$$\begin{array}{ll} P(z | x_1) = 0.8 & P(\neg z | x_1) = 0.2 \\ P(z | x_2) = 0.4 & P(\neg z | x_2) = 0.6 \\ P(z | x_3) = 0.1 & P(\neg z | x_3) = 0.9 \\ P(z | x_4) = 0.1 & P(\neg z | x_4) = 0.9 \end{array} .$$

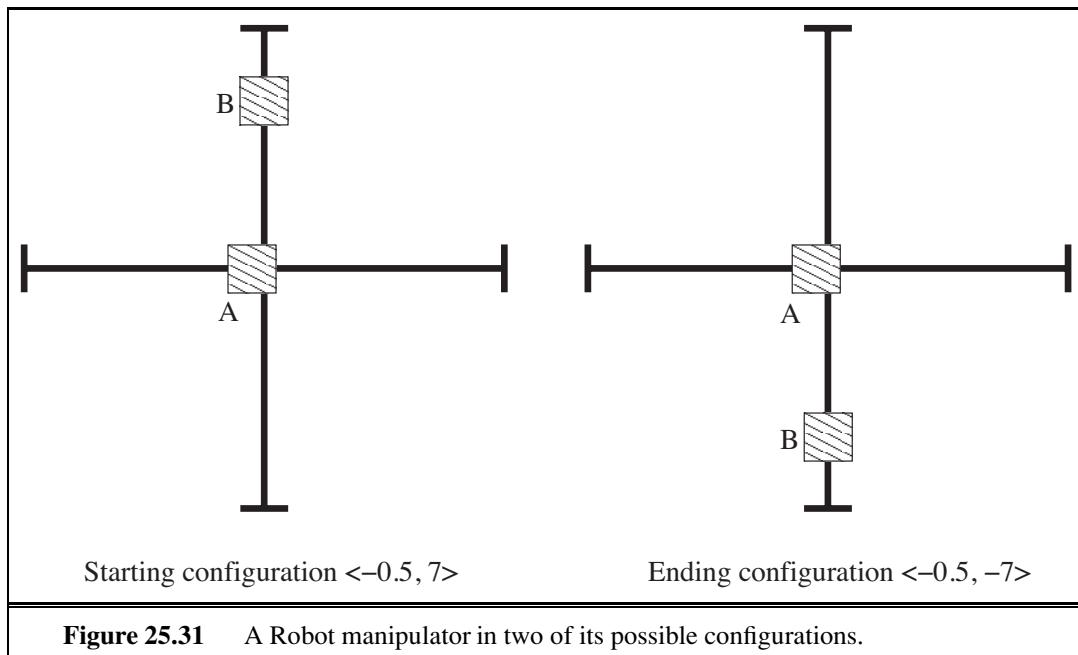


Figure 25.31 A Robot manipulator in two of its possible configurations.

MCL uses these probabilities to generate particle weights, which are subsequently normalized and used in the resampling process. For simplicity, let us assume we generate only 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 .

- What is the resulting probability distribution over X for this new sample? Answer this question separately for $N = 1, \dots, 10$, and for $N = \infty$.
- 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?

- 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. You should demonstrate successful global localization of the robot.

25.3 Consider a robot with two simple manipulators, as shown in figure 25.31. Manipulator A is a square block of side 2 which can slide back and forth on a rod that runs along the x-axis from $x = -10$ to $x = 10$. Manipulator B is a square block of side 2 which can slide back and forth on a rod that runs along the y-axis from $y = -10$ to $y = 10$. The rods lie outside the plane of

manipulation, so the rods do not interfere with the movement of the blocks. A configuration is then a pair $\langle x, y \rangle$ where x is the x-coordinate of the center of manipulator A and where y is the y-coordinate of the center of manipulator B. Draw the configuration space for this robot, indicating the permitted and excluded zones.

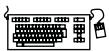
25.4 Suppose that you are working with the robot in Exercise 25.3 and you are given the problem of finding a path from the starting configuration of figure 25.31 to the ending configuration. Consider a potential function

$$D(A, Goal)^2 + D(B, Goal)^2 + \frac{1}{D(A, B)^2}$$

where $D(A, B)$ is the distance between the closest points of A and B.

- a. Show that hill climbing in this potential field will get stuck in a local minimum.
- b. Describe a potential field where hill climbing will solve this particular problem. You need not work out the exact numerical coefficients needed, just the general form of the solution. (Hint: Add a term that “rewards” the hill climber for moving A out of B’s way, even in a case like this where this does not reduce the distance from A to B in the above sense.)

25.5 Consider the robot arm shown in Figure 25.14. 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 987, 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?



25.6 Implement an algorithm for calculating the Voronoi diagram of an arbitrary 2D 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?

25.7 This exercise explores the relationship between workspace and configuration space using the examples shown in Figure 25.32.

- a. Consider the robot configurations shown in Figure 25.32(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 25.14(b).)
- b. Draw the configuration space for each of the workspace diagrams in Figure 25.32(a)–(c). (Hint: The configuration spaces share with the one shown in Figure 25.32(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 25.32(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 25.32(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.

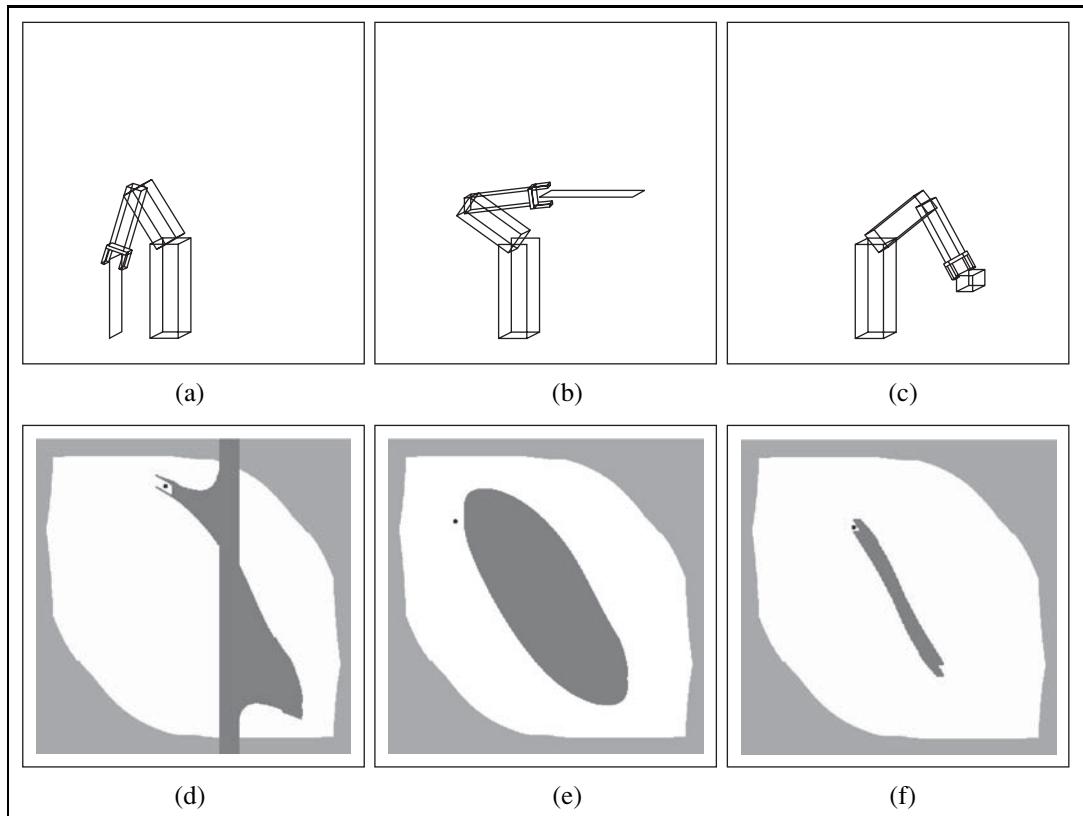


Figure 25.32 Diagrams for Exercise 25.7.

- e. Figure 25.32(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?

25.8 Consider a mobile robot moving on a horizontal surface. Suppose that the robot can execute two kinds of motions:

- Rolling forward a specified distance.
- Rotating in place through a specified angle.

The state of such a robot can be characterized in terms of three parameters $\langle x, y, \phi \rangle$, the x-coordinate and y-coordinate of the robot (more precisely, of its center of rotation) and the robot's orientation expressed as the angle from the positive x direction. The action “*Roll(D)*” has the effect of changing state $\langle x, y, \phi \rangle$ to $\langle x + D \cos(\phi), y + D \sin(\phi), \phi \rangle$, and the action *Rotate(θ)* has the effect of changing state $\langle x, y, \phi \rangle$ to $\langle x, y, \phi + \theta \rangle$.

- a. Suppose that the robot is initially at $\langle 0, 0, 0 \rangle$ and then executes the actions *Rotate(60°)*, *Roll(1)*, *Rotate(25°)*, *Roll(2)*. What is the final state of the robot?

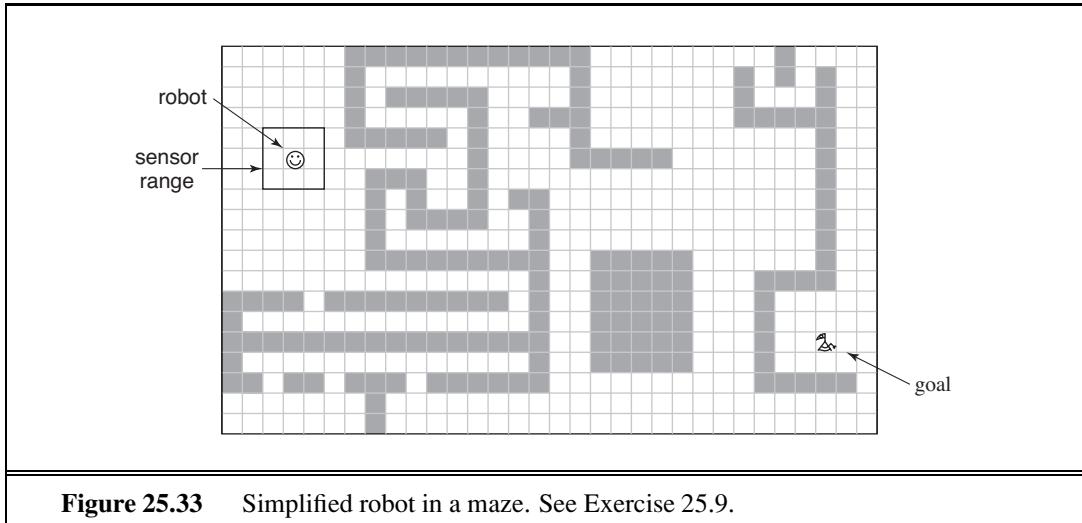


Figure 25.33 Simplified robot in a maze. See Exercise 25.9.

- b. Now suppose that the robot has imperfect control of its own rotation, and that, if it attempts to rotate by θ , it may actually rotate by any angle between $\theta - 10^\circ$ and $\theta + 10^\circ$. In that case, if the robot attempts to carry out the sequence of actions in (A), there is a range of possible ending states. What are the minimal and maximal values of the x-coordinate, the y-coordinate and the orientation in the final state?
- c. Let us modify the model in (B) to a probabilistic model in which, when the robot attempts to rotate by θ , its actual angle of rotation follows a Gaussian distribution with mean θ and standard deviation 10° . Suppose that the robot executes the actions *Rotate(90°)*, *Roll(1)*. Give a simple argument that (a) the expected value of the location at the end is not equal to the result of rotating exactly 90° and then rolling forward 1 unit, and (b) that the distribution of locations at the end does not follow a Gaussian. (Do not attempt to calculate the true mean or the true distribution.)

The point of this exercise is that rotational uncertainty quickly gives rise to a lot of positional uncertainty and that dealing with rotational uncertainty is painful, whether uncertainty is treated in terms of hard intervals or probabilistically, due to the fact that the relation between orientation and position is both non-linear and non-monotonic.

25.9 Consider the simplified robot shown in Figure 25.33. Suppose the robot's Cartesian coordinates are known at all times, as are those of its goal 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 25.33 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.

- a. Design a deliberate controller that guarantees that the robot always reaches its goal location if at all possible. The deliberate controller can memorize measurements in the form of a map that is being acquired as the robot moves. Between individual moves, it may spend arbitrary time deliberating.

- b. 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 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 fail).

25.10 In Figure 25.24(b) on page 1001, 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 dynamic stability condition given on page 977.

25.11 (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 household tasks 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. Consider the task of building an arch out of three blocks. Simulate a robot with four humans as follows:

Brain. The Brain direct the hands in the execution of a plan to achieve the goal. 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 report a brief description of the scene to the Brain: “There is a red box standing on top of a green box, which is on its side” 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, each wearing an oven mitt on one hand, 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, they cannot be commanded to “Pick up the box.” 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.

Philosophers have been around far 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 real, conscious minds? What are the ethical implications of intelligent machines?

WEAK AI
STRONG AI

First, some terminology: the assertion that machines could 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 (not just *simulating* thinking) is called the **strong AI** hypothesis.

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?

The proposal for the 1956 summer workshop that defined the field of Artificial Intelligence (McCarthy *et al.*, 1955) made the assertion that “Every aspect of learning or any other feature of intelligence can be so precisely described that a machine can be made to simulate it.” Thus, AI was founded on the assumption that weak AI is possible. Others have asserted that weak AI is impossible: “Artificial intelligence pursued within the cult of computationalism stands not even a ghost of a chance of producing durable results” (Sayre, 1993).

Clearly, whether AI is impossible depends on how it is defined. In Section 1.1, we defined AI as the quest for the best agent program on a given architecture. With this formulation, AI is by definition possible: for any digital architecture with k bits of program 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.

CAN MACHINES
THINK?

CAN SUBMARINES
SWIM?

TURING TEST

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 not in terms of maximizing expected utility but rather as, “**Can machines think?**”

The computer scientist Edsger Dijkstra (1984) said that “The question of whether *Machines Can Think* . . . is about as relevant as the question of whether *Submarines Can Swim*. ” The American Heritage Dictionary’s first definition of *swim* is “To move through water by means of the limbs, fins, or tail,” and most people agree that submarines, being limbless, cannot swim. The dictionary also defines *fly* as “To move through the air by means of wings or winglike parts,” and most people agree that airplanes, having winglike parts, can fly. However, neither the questions nor the answers have any relevance to the design or capabilities of airplanes and submarines; rather they are about the usage of words in English. (The fact that ships *do* swim in Russian only amplifies this point.). 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”—does it require “a brain” or just “brain-like parts.”

Alan Turing, in his famous paper “Computing Machinery and Intelligence” (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 five 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. He was wrong—programs have yet to fool a sophisticated judge.

On the other hand, many people *have* been fooled when they didn’t know they might be chatting with a computer. The ELIZA program and Internet chatbots such as MGONZ (Humphrys, 2008) and NATACHATA have fooled their correspondents repeatedly, and the chatbot CYBERLOVER has attracted the attention of law enforcement because of its penchant for tricking fellow chatters into divulging enough personal information that their identity can be stolen. The Loebner Prize competition, held annually since 1991, is the longest-running Turing Test-like contest. The competitions have led to better models of human typing errors.

Turing himself 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.

26.1.1 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:

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.

In retrospect, some of these are rather easy—we’re all familiar with computers that “make mistakes.” We are also familiar with a century-old technology that has had a proven ability to “make someone fall in love with it”—the teddy bear. Computer chess expert David Levy predicts that by 2050 people will routinely fall in love with humanoid robots (Levy, 2007). As for a robot falling in love, that is a common theme in fiction,¹ but there has been only limited speculation about whether it is in fact likely (Kim *et al.*, 2007). Programs do play chess, checkers and other games; inspect parts on assembly lines, 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.

26.1.2 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.

¹ For example, the opera Coppélia (1870), the novel *Do Androids Dream of Electric Sheep?* (1968), the movies *AI* (2001) and *Wall-E* (2008), and in song, Noel Coward’s 1955 version of *Let’s Do It: Let’s Fall in Love* predicted “probably we’ll live to see machines do it.” He didn’t.

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. 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 a 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 formal mathematical reasoning plays more than a peripheral role in what it means to be intelligent.

Third, and most important, 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 require 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.

26.1.3 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.

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), the sequel *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 philosopher John 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 the following:

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 Chapters 19 and 20 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. With the introduction of large Web-based data sets, many applications in areas such as language processing (Sha and Pereira, 2003) and computer vision (Viola and Jones, 2002a) routinely handle millions of features. We saw in Chapter 19 that 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, Dreyfus and Dreyfus 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. STANLEY’s 132-mile trip through the desert (page 28) was made possible in large part by an active sensing system of this kind.

In sum, many of the issues Dreyfus has focused on—background commonsense knowledge, the qualification problem, uncertainty, learning, compiled forms of decision making—are indeed important issues, and have by now been incorporated into standard intelligent agent design. In our view, this is evidence of AI’s progress, not of its impossibility.

One of Dreyfus’ strongest arguments is for situated agents rather than disembodied logical inference engines. An agent whose understanding of “dog” comes only from a limited set of logical sentences such as “ $Dog(x) \Rightarrow Mammal(x)$ ” is at a disadvantage compared to an agent that has watched dogs run, has played fetch with them, and has been licked by one. As philosopher Andy Clark (1998) says, “Biological brains are first and foremost the control systems for biological bodies. Biological bodies move and act in rich real-world

surroundings.” To understand how human (or other animal) agents work, we have to consider the whole agent, not just the agent program. Indeed, the **embodied cognition** approach claims that it makes no sense to consider the brain separately: cognition takes place within a body, which is embedded in an environment. We need to study the system as a whole; the brain augments its reasoning by referring to the environment, as the reader does in perceiving (and creating) marks on paper to transfer knowledge. Under the embodied cognition program, robotics, vision, and other sensors become central, not peripheral.

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.

One can easily imagine some future time in which such conversations with machines are commonplace, and it becomes customary to make no linguistic distinction between "real" and "artificial" thinking. A similar transition occurred in the years after 1848, when artificial urea was synthesized for the first time by Frederick Wöhler. Prior to this event, organic and inorganic chemistry were essentially disjoint enterprises and many thought that no process could exist that would convert inorganic chemicals into organic material. Once the synthesis was accomplished, chemists agreed that artificial urea *was* urea, because it had all the right physical properties. Those who had posited an intrinsic property possessed by organic material that inorganic material could never have were faced with the impossibility of devising any test that could reveal the supposed deficiency of artificial urea.

For thinking, we have not yet reached our 1848 and there are those who believe that artificial thinking, no matter how impressive, will never be real. For example, the philosopher John Searle (1980) argues as follows:

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, and a video game simulation of a storm *does* make the simulated characters wet. Most people are comfortable saying that a computer simulation of addition is addition, and of chess is chess. In fact, we typically speak of an *implementation* of addition or chess, not a *simulation*. Are mental processes more like storms, or more like addition?

Turing's answer—the polite convention—suggests that the issue will eventually go away by itself once machines reach a certain level of sophistication. This would have the effect of *dissolving* the difference between weak and strong AI. Against this, one may insist that there is a *factual* issue at stake: humans do have real minds, and machines might or might not. To address this factual issue, we need to understand how it is that humans have real minds, not just bodies that generate neurophysiological processes. Philosophical efforts to solve this **mind–body problem** are directly relevant to the question of whether machines could have real minds.

MIND–BODY PROBLEM

The mind–body problem was considered by the ancient Greek philosophers and by various schools of Hindu thought, but was first analyzed in depth by the 17th-century French philosopher and mathematician René Descartes. His *Meditations on First Philosophy* (1641) considered the mind's activity of thinking (a process with no spatial extent or material properties) and the physical processes of the body, concluding that the two must exist in separate realms—what we would now call a **dualist** theory. The mind–body problem faced by dualists is the question of how the mind can control the body if the two are really separate. Descartes speculated that the two might interact through the pineal gland, which simply begs the question of how the mind controls the pineal gland.

DUALISM

MONISM
PHYSICALISM

MENTAL STATES

INTENTIONAL STATE

The **monist** theory of mind, often called **physicalism**, avoids this problem by asserting the mind is not separate from the body—that mental states *are* physical states. Most modern philosophers of mind are physicalists of one form or another, and physicalism allows, at least in principle, for the possibility of strong AI. The problem for physicalists is to explain how physical states—in particular, the molecular configurations and electrochemical processes of the brain—can simultaneously be **mental states**, such as being in pain, enjoying a hamburger, knowing that one is riding a horse, or believing that Vienna is the capital of Austria.

26.2.1 Mental states and the brain in a vat

Physicalist philosophers have attempted to explicate what it means to say that a person—and, by extension, a computer—is in a particular mental state. They have focused in particular on **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 knowledge that one is eating a hamburger is a belief *about* the hamburger and what is happening to it.

If physicalism is correct, it must be the case that the proper description of a person's mental state is *determined* by that person's brain state. Thus, if I am currently focused on eating a hamburger in a mindful way, my instantaneous brain state is an instance of the class of mental states “knowing that one is eating a hamburger.” Of course, the specific configurations of all the atoms of my brain are not essential: there are many configurations of my brain, or of other people's brain, that would belong to the same class of mental states. The key point is that the same brain state could not correspond to a fundamentally distinct mental state, such as the knowledge that one is eating a banana.

The simplicity of this view is challenged by some simple thought experiments. 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.² In fact, the simulated life you live replicates exactly the life you would have lived, had your brain not been placed in the vat, including simulated eating of simulated hamburgers. Thus, you could have a brain state identical to that of someone who is really eating a real hamburger, but it would be literally false to say that you have the mental state “knowing that one is eating a hamburger.” You aren't eating a hamburger, you have never even experienced a hamburger, and you could not, therefore, have such a mental state.

WIDE CONTENT

NARROW CONTENT

This example seems to contradict the view that brain states determine mental states. 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. Under this view, the content of mental states involves both the brain state and the environment history. **Narrow content**, on the other hand, considers only the brain state. The narrow content of the brain states of a real hamburger-eater and a brain-in-a-vat “hamburger”-“eater” is the same in both cases.

² This situation may be familiar to those who have seen the 1999 film *The Matrix*.

Wide content is entirely appropriate if one’s goals are to ascribe mental states to others who share one’s world, to predict their likely behavior and its effects, and so on. This is the setting in which our ordinary language about mental content has evolved. On the other hand, if one is concerned with the question of whether AI systems are really thinking and really do have mental states, then narrow content is appropriate; it simply doesn’t make sense to say that whether or not an AI system is really thinking depends on conditions outside that system. Narrow content is also relevant if we are thinking about designing AI systems or understanding their operation, because it is the narrow content of a brain state that determines what will be the (narrow content of the) next brain state. This leads naturally to the idea that what matters about a brain state—what makes it have one kind of mental content and not another—is its functional role within the mental operation of the entity involved.

26.2.2 Functionalism and the brain replacement experiment

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.

The claims of functionalism are illustrated most clearly by the brain replacement experiment. This thought experiment was introduced by the philosopher Clark Glymour and was touched on by John Searle (1980), but is most commonly associated with roboticist 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.

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.³ 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:

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

³ One can imagine using an identical “control” subject who is given a placebo operation, for comparison.

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.” . . . your conscious experience slowly shrinks to nothing, while your externally observable behavior remains the same. (Searle, 1992)

One can do more than argue from intuition. First, note that, 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 three 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.
3. The experiment is impossible, and therefore speculation about it is meaningless.

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 it cannot be the case that the subject says “Ouch” *because it hurts*—that is, because of the conscious experience of pain. Instead, the brain must contain a second, unconscious mechanism that is responsible for the “Ouch.”

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 replacement 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 updates its state and 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.

26.2.3 Biological naturalism and the Chinese Room

BIOLOGICAL
NATURALISM

A strong challenge to functionalism has been mounted by John Searle's (1980) **biological naturalism**, according to which mental states are high-level emergent features that are caused by low-level physical 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. To support his view, Searle 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 “intelligent” as those in the conversation imagined by Turing.⁴ Searle then argues: the person in the room does not understand Chinese (given). The rule book and the stacks of paper, being just pieces of paper, do not understand Chinese. Therefore, there is no understanding of Chinese. *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 asking if the human in the room understands Chinese 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 seems to be relying on the argument that a property of the whole must reside in one of the parts. Yet

⁴ The fact that the stacks of paper might contain trillions of pages 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.

water is wet, even though neither H nor O₂ is. The real claim made by Searle rests upon the following four axioms (Searle, 1990):

1. Computer programs are formal (syntactic).
2. Human minds have mental contents (semantics).
3. Syntax by itself is neither constitutive of nor sufficient for semantics.
4. Brains cause minds.

From the first three axioms Searle 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 axioms are controversial. For example, axioms 1 and 2 rely on an unspecified distinction between syntax and semantics that seems to be closely related to the distinction between narrow and wide content. On the one hand, we can view computers as manipulating syntactic symbols; on the other, we can view them as manipulating electric current, which happens to be what brains mostly do (according to our current understanding). So it seems we could equally say that brains are syntactic.

Assuming we are generous in interpreting the axioms, then the conclusion—that programs are not sufficient for minds—*does* follow. 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—almost tautological—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. The public reaction shows that the argument is acting as what Daniel Dennett (1991) calls an **intuition pump**: it amplifies one’s prior intuitions, so biological naturalists are more convinced of their positions, and functionalists are convinced only that axiom 3 is unsupported, or that in general Searle’s argument is unconvincing. The argument stirs up combatants, but has done little to change anyone’s opinion. Searle remains undeterred, and has recently started calling the Chinese Room a “refutation” of strong AI rather than just an “argument” (Snell, 2008).

Even those who accept axiom 3, and thus accept Searle’s argument, have only their intuitions to fall back on when deciding what entities are minds. The argument purports to show that the Chinese Room is not a mind *by virtue of running the program*, but the argument says nothing about how to decide whether the room (or a computer, some other type of machine, or an alien) is a mind *by virtue of some other reason*. Searle himself says that some machines do have minds: humans are biological machines with minds. According to Searle, human brains may or may not be running something like an AI program, but if they are, that is not the reason they are minds. It takes more to make a mind—according to Searle, something equivalent to the causal powers of individual neurons. What these powers are is left unspecified. It should be noted, however, that neurons evolved to fulfill *functional* roles—creatures

with neurons were learning and deciding long before consciousness appeared on the scene. It would be a remarkable coincidence if such neurons just happened to generate consciousness because of some causal powers that are irrelevant to their functional capabilities; after all, it is the functional capabilities that dictate survival of the organism.

In the case of the Chinese Room, Searle relies on intuition, not proof: 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? That remains the great mystery.

26.2.4 Consciousness, qualia, and the explanatory gap

CONSCIOUSNESS
Running through all the debates about strong AI—the elephant in the debating room, so to speak—is the issue of **consciousness**. Consciousness is often broken down into aspects such as understanding and self-awareness. The aspect we will focus on is that of *subjective experience*: why it is that it *feels* like something to have certain brain states (e.g., while eating a hamburger), whereas it presumably does not feel like anything to have other physical states (e.g., while being a rock). The technical term for the intrinsic nature of experiences is **qualia** (from the Latin word meaning, roughly, “such things”).

QUALIA

INVERTED SPECTRUM

EXPLANATORY GAP

Qualia present a challenge for functionalist accounts of the mind because different qualia could be involved in what are otherwise isomorphic causal processes. Consider, for example, the **inverted spectrum** thought experiment, which the subjective experience of person *X* when seeing red objects is the same experience that the rest of us experience when seeing green objects, and vice versa. *X* still calls red objects “red,” stops for red traffic lights, and agrees that the redness of red traffic lights is a more intense red than the redness of the setting sun. Yet, *X*’s subjective experience is just different.

Qualia are challenging not just for functionalism but for all of science. Suppose, for the sake of argument, that we have completed the process of scientific research on the brain—we have found that neural process P_{12} in neuron N_{177} transforms molecule *A* into molecule *B*, and so on, and on. There is simply no currently accepted form of reasoning that would lead from such findings to the conclusion that the entity owning those neurons has any particular subjective experience. This **explanatory gap** has led some philosophers to conclude that humans are simply incapable of forming a proper understanding of their own consciousness. Others, notably Daniel Dennett (1991), avoid the gap by denying the existence of qualia, attributing them to a philosophical confusion.

Turing himself 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. The additional project of making them conscious is not one that we are equipped to take on, nor one whose success we would be able to determine.

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: nuclear fission brought Chernobyl and the threat of global destruction; the internal combustion engine brought air pollution, global warming, and the paving-over of paradise. In a sense, automobiles are robots that have conquered the world by making themselves indispensable.

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. See the 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.
- AI systems might be used toward undesirable ends.
- 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 through information technology in general and AI in particular 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. But some researchers think that doing the complete job is the right goal for AI. In reflecting on the 25th Anniversary of the AAAI, Nils Nilsson (2005) set as a challenge the creation of human-level AI that could pass the employment test rather than the Turing Test—a robot that could learn to do any one of a range of jobs. We may end up in a future where unemployment is high, but even the unemployed serve as managers of their own cadre of robot workers.

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. 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. Tim Ferriss (2007) recommends using automation and outsourcing to achieve a four-hour work week.

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). 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.

AI systems might be used toward undesirable ends. Advanced technologies have often been used by the powerful to suppress their rivals. As the number theorist G. H. Hardy wrote (Hardy, 1940), “A science is said to be useful if its development tends to accentuate the existing inequalities in the distribution of wealth, or more directly promotes the destruction of human life.” This holds for all sciences, AI being no exception. Autonomous AI systems are now commonplace on the battlefield; the U.S. military deployed over 5,000 autonomous aircraft and 12,000 autonomous ground vehicles in Iraq (Singer, 2009). One moral theory holds that military robots are like medieval armor taken to its logical extreme: no one would have moral objections to a soldier wanting to wear a helmet when being attacked by large, angry, axe-wielding enemies, and a teleoperated robot is like a very safe form of armor. On the other hand, robotic weapons pose additional risks. To the extent that human decision making is taken out of the firing loop, robots may end up making decisions that lead to the killing of innocent civilians. At a larger scale, the possession of powerful robots (like the possession of sturdy helmets) may give a nation overconfidence, causing it to go to war more recklessly than necessary. In most wars, at least one party is overconfident in its military abilities—otherwise the conflict would have been resolved peacefully.

Weizenbaum (1976) 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 has in part come true: the U.K. now has an extensive network of surveillance cameras, and other countries routinely monitor Web traffic and telephone calls. Some accept that computerization leads to a loss of privacy—Sun Microsystems CEO Scott McNealy has said “You have zero privacy anyway. Get over it.” David Brin (1998) argues that loss of privacy is inevitable, and the way to combat the asymmetry of power of the state over the individual is to make the surveillance accessible to all citizens. Etzioni (2004) argues for a balancing of privacy and security; individual rights and community.

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. Atul 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)⁵ and Karel Čapek'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.

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, or the Martians from *The War of the Worlds* (Wells, 1898). The question is whether an AI system poses a bigger risk than traditional software. We will look at three sources of risk.

First, the AI system's state estimation may be incorrect, causing it to do the wrong thing. For example, an autonomous car might incorrectly estimate the position of a car in the adjacent lane, leading to an accident that might kill the occupants. More seriously, a missile defense system might erroneously detect an attack and launch a counterattack, leading to the death of billions. These risks are not really risks of AI systems—in both cases the same mistake could just as easily be made by a human as by a computer. The correct way to mitigate these risks is to design a system with checks and balances so that a single state-estimation error does not propagate through the system unchecked.

Second, specifying the right utility function for an AI system to maximize is not so easy. For example, we might propose a utility function designed to *minimize human suffering*, expressed as an additive reward function over time as in Chapter 17. Given the way humans are, however, we'll always find a way to suffer even in paradise; so the optimal decision for the AI system is to terminate the human race as soon as possible—no humans, no suffering. With AI systems, then, we need to be very careful what we ask for, whereas humans would have no trouble realizing that the proposed utility function cannot be taken literally. On the other hand, computers need not be tainted by the irrational behaviors described in Chapter 16. Humans sometimes use their intelligence in aggressive ways because humans have some innately aggressive tendencies, due to natural selection. The machines we build need not be innately aggressive, unless we decide to build them that way (or unless they emerge as the end product of a mechanism design that encourages aggressive behavior). Fortunately, there are techniques, such as apprenticeship learning, that allows us to specify a utility function by example. One can hope that a robot that is smart enough to figure out how to terminate the human race is also smart enough to figure out that that was not the intended utility function.

Third, the AI system's learning function may cause it to evolve into a system with unintended behavior. This scenario is the most serious, and is unique to AI systems, so we will cover it in more depth. I. J. Good wrote (1965),

ULTRAINTELLIGENT
MACHINE

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.

⁵ As a young man, Charles Babbage was influenced by reading *Frankenstein*.

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 (on many measures) is growing exponentially at present (consider Moore’s Law). However, it is a leap to extrapolate that the curve will continue 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. Sometimes new technologies step in when the old ones plateau; sometimes we hit hard limits. With less than a century of high-technology history to go on, it is difficult to extrapolate hundreds of years ahead.

Note that the concept of ultraintelligent machines assumes that intelligence is an especially important attribute, and if you have enough of it, all problems can be solved. But we know there are limits on computability and computational complexity. If the problem of defining ultraintelligent machines (or even approximations to them) happens to fall in the class of, say, NEXPTIME-complete problems, and if there are no heuristic shortcuts, then even exponential progress in technology won’t help—the speed of light puts a strict upper bound on how much computing can be done; problems beyond that limit will not be solved. We still don’t know where those upper bounds are.

TRANSHUMANISM

Vinge is concerned about the coming singularity, but some computer scientists and futurists relish it. Hans Moravec (2000) encourages us to give every advantage to our “mind children,” the robots we create, which may surpass us in intelligence. There is even a new word—**transhumanism**—for the active social movement that looks forward to this future in which humans are merged with—or replaced by—robotic and biotech inventions. 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. Ray Kurzweil is currently the most visible advocate for the singularity view, writing in *The Singularity is Near* (2005):

The Singularity will allow us to transcend these limitations of our biological bodies and brain. We will gain power over our fates. Our mortality will be in our own hands. We will be able to live as long as we want (a subtly different statement from saying we will live forever). We will fully understand human thinking and will vastly extend and expand its reach. By the end of this century, the nonbiological portion of our intelligence will be trillions of trillions of times more powerful than unaided human intelligence.

Kurzweil also notes the potential dangers, writing “But the Singularity will also amplify the ability to act on our destructive inclinations, so its full story has not yet been written.”

If ultraintelligent machines are a possibility, we humans would do well to make sure that we design their predecessors in such a way that they design themselves to treat us well. Science fiction writer Isaac Asimov (1942) was the first to address this issue, with his three laws of robotics:

1. A robot may not injure a human being or, through inaction, allow a human being to come to harm.
2. A robot must obey orders given to it by human beings, except where such orders would conflict with the First Law.

3. A robot must protect its own existence as long as such protection does not conflict with the First or Second Law.

These laws seem reasonable, at least to us humans.⁶ But the trick is how to implement these laws. In the Asimov story *Roundabout* a robot is sent to fetch some selenium. Later the robot is found wandering in a circle around the selenium source. Every time it heads toward the source, it senses a danger, and the third law causes it to veer away. But every time it veers away, the danger recedes, and the power of the second law takes over, causing it to veer back towards the selenium. The set of points that define the balancing point between the two laws defines a circle. This suggests that the laws are not logical absolutes, but rather are weighed against each other, with a higher weighting for the earlier laws. Asimov was probably thinking of an architecture based on control theory—perhaps a linear combination of factors—while today the most likely architecture would be a probabilistic reasoning agent that reasons over probability distributions of outcomes, and maximizes utility as defined by the three laws. But presumably we don't want our robots to prevent a human from crossing the street because of the nonzero chance of harm. That means that the negative utility for harm to a human must be much greater than for disobeying, but that each of the utilities is finite, not infinite.

FRIENDLY AI

Yudkowsky (2008) goes into more detail about how to design a **Friendly AI**. He asserts that friendliness (a desire not to harm humans) should be designed in from the start, but that the designers should recognize both that their own designs may be flawed, and that the robot will learn and evolve over time. Thus the challenge is one of mechanism design—to define a mechanism for evolving AI systems under a system of checks and balances, and to give the systems utility functions that will remain friendly in the face of such changes.

We can't just give a program a static utility function, because circumstances, and our desired responses to circumstances, change over time. For example, if technology had allowed us to design a super-powerful AI agent in 1800 and endow it with the prevailing morals of the time, it would be fighting today to reestablish slavery and abolish women's right to vote. On the other hand, if we build an AI agent today and tell it to evolve its utility function, how can we assure that it won't reason that "Humans think it is moral to kill annoying insects, in part because insect brains are so primitive. But human brains are primitive compared to my powers, so it must be moral for me to kill humans."

Omohundro (2008) hypothesizes that even an innocuous chess program could pose a risk to society. Similarly, Marvin Minsky once suggested that an AI program designed to solve the Riemann Hypothesis might end up taking over all the resources of Earth to build more powerful supercomputers to help achieve its goal. The moral is that even if you only want your program to play chess or prove theorems, if you give it the capability to learn and alter itself, you need safeguards. Omohundro concludes that "Social structures which cause individuals to bear the cost of their negative externalities would go a long way toward ensuring a stable and positive future." This seems to be an excellent idea for society in general, regardless of the possibility of ultraintelligent machines.

⁶ A robot might notice the inequity that a human is allowed to kill another in self-defense, but a robot is required to sacrifice its own life to save a human.

We should note that the idea of safeguards against change in utility function is not a new one. In the *Odyssey*, Homer (ca. 700 B.C.) described Ulysses' encounter with the sirens, whose song was so alluring it compelled sailors to cast themselves into the sea. Knowing it would have that effect on him, Ulysses ordered his crew to bind him to the mast so that he could not perform the self-destructive act. It is interesting to think how similar safeguards could be built into AI systems.

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. Science fiction writers have addressed the issue of robot rights. The 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) argue for 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 "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.

BIBLIOGRAPHICAL AND HISTORICAL NOTES

Sources for the various responses to Turing's 1950 paper and 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 (Wylshyn, 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" the claim by Joseph Weizenbaum (1976) that intelligent machines can never be regarded as persons.

Proponents of the importance of embodiment in cognition include the philosophers Merleau-Ponty, whose *Phenomenology of Perception* (1945) stressed the importance of the body and the subjective interpretation of reality afforded by our senses, and Heidegger, whose *Being and Time* (1927) asked what it means to actually be an agent, and criticized all of the history of philosophy for taking this notion for granted. In the computer age, Alva Noe (2009) and Andy Clark (1998, 2008) propose that our brains form a rather minimal representation of the world, use the world itself in a just-in-time basis to maintain the illusion of a detailed internal model, use props in the world (such as paper and pencil as well as computers) to increase the capabilities of the mind. Pfeifer *et al.* (2006) and Lakoff and Johnson (1999) present arguments for how the body helps shape cognition.

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 $\psi\upsilon\chi\eta$) 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 physicalism. 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* (La Mettrie, 1748) did explicitly argue that humans are automata.

Modern analytic philosophy has typically accepted physicalism, but the variety of views on the content of mental states is bewildering. The identification of mental states with brain states is usually attributed to Place (1956) and Smart (1959). The debate between narrow-content and wide-content views of mental states was triggered by Hilary Putnam (1975), who introduced so-called **twin earths** (rather than brain-in-a-vat, as we did in the chapter) as a device to generate identical brain states with different (wide) content.

Functionalism is the philosophy of mind most naturally suggested by AI. The idea that mental states correspond to classes of brain states defined functionally is due to Putnam (1960, 1967) and Lewis (1966, 1980). Perhaps the most forceful proponent of functionalism is Daniel Dennett, whose ambitiously titled work *Consciousness Explained* (Dennett, 1991) has attracted many attempted rebuttals. Metzinger (2009) argues there is no such thing as an objective *self*, that consciousness is the subjective appearance of a world. The inverted spectrum argument concerning qualia was introduced by John Locke (1690). Frank Jackson (1982) designed an influential thought experiment involving Mary, a color scientist who has been brought up in an entirely black-and-white world. *There's Something About Mary* (Ludlow *et al.*, 2004) collects several papers on this topic.

Functionalism has 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. The Chinese Room has been debated endlessly (Searle, 1980, 1990; Preston and Bishop, 2002). We’ll just mention here a related work: Terry Bisson’s (1990) science fiction story *They're Made out of Meat*, in which alien robotic explorers who visit earth are incredulous to find thinking human beings whose minds are made of meat. Presumably, the robotic alien equivalent of Searle believes that he can think due to the special causal powers of robotic circuits; causal powers that mere meat-brains do not possess.

Ethical issues in AI predate the existence of the field itself. I. J. Good’s (1965) ultra-intelligent machine idea was foreseen a hundred years earlier by Samuel Butler (1863). Written four years after the publication of Darwin’s *On the Origins of Species* and at a time when the most sophisticated machines were steam engines, Butler’s article on *Darwin Among the Machines* envisioned “the ultimate development of mechanical consciousness” by natural selection. The theme was reiterated by George Dyson (1998) in a book of the same title.

The philosophical literature on minds, brains, and related topics is large and difficult to read without training in the terminology and methods of argument employed. The *Encyclopedia of Philosophy* (Edwards, 1967) is an impressively authoritative and very useful aid in this process. The *Cambridge Dictionary of Philosophy* (Audi, 1999) is a shorter and more accessible work, and the online *Stanford Encyclopedia of Philosophy* offers many excellent articles and up-to-date references. The *MIT Encyclopedia of Cognitive Science* (Wilson and Keil, 1999) covers the philosophy of mind as well as the biology and psychology of mind. There are several general introductions to the philosophical “AI question” (Boden, 1990; Haugeland, 1985; Copeland, 1993; McCorduck, 2004; Minsky, 2007). *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 the journals *AI and Society* and *Journal of 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** Find and analyze an account in the popular media of one or more of the arguments to the effect that AI is impossible.
- 26.3** In the brain replacement 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** Suppose that a Prolog program containing many clauses about the rules of British citizenship is compiled and run on an ordinary computer. Analyze the “brain states” of the computer under wide and narrow content.
- 26.5** Alan Perlis (1982) wrote, “A year spent in artificial intelligence is enough to make one believe in God”. He also wrote, in a letter to Philip Davis, that one of the central dreams of computer science is that “through the performance of computers and their programs we will remove all doubt that there is only a chemical distinction between the living and nonliving world.” To what extent does the progress made so far in artificial intelligence shed light on these issues? Suppose that at some future date, the AI endeavor has been completely successful; that is, we have build intelligent agents capable of carrying out any human cognitive task at human levels of ability. To what extent would that shed light on these issues?
- 26.6** Compare the social impact of artificial intelligence in the last fifty years with the social impact of the introduction of electric appliances and the internal combustion engine in the fifty years between 1890 and 1940.
- 26.7** I. J. Good claims that intelligence is the most important quality, and that building ultraintelligent machines will change everything. A sentient cheetah counters that “Actually speed is more important; if we could build ultrafast machines, that would change everything,” and a sentient elephant claims “You’re both wrong; what we need is ultrastrong machines.” What do you think of these arguments?
- 26.8** 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?
- 26.9** How do the potential threats from AI technology compare with those from other computer science technologies, and to bio-, nano-, and nuclear technologies?
- 26.10** 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: THE 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 Chapter 2, we suggested that it would be helpful to view the AI task as that of designing rational agents—that is, agents whose actions maximize their expected utility given their percept histories. We showed that the design problem depends on the percepts and actions available to the agent, the utility function 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, decision-theoretic agents. Moreover, the components of these designs can have a number of different instantiations—for example, logical or probabilistic reasoning, and atomic, factored, or structured representations of states. The intervening chapters presented the principles by which these components operate.

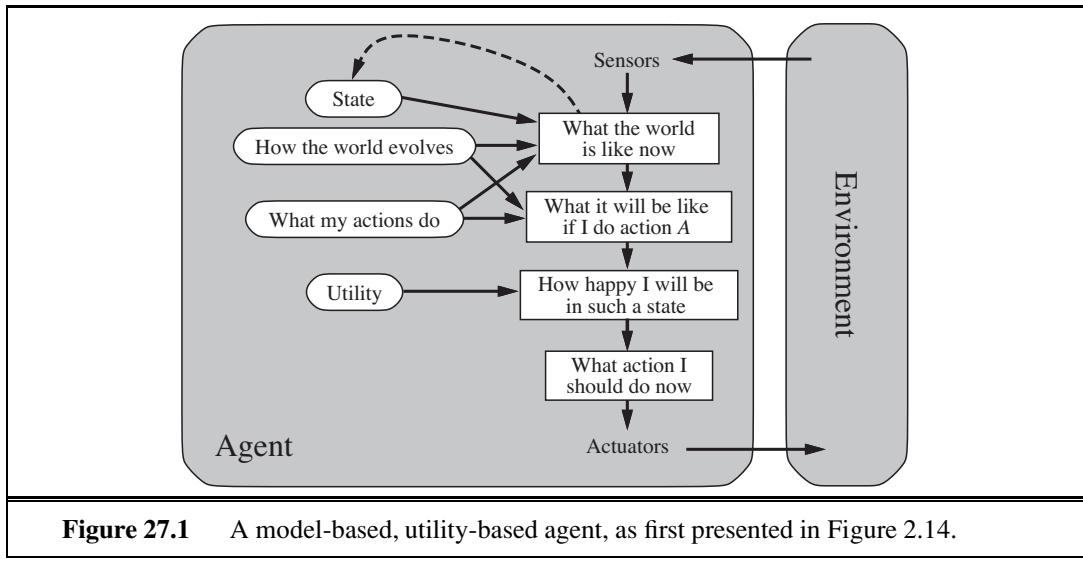


For all the agent designs and components, there has been tremendous progress both in our scientific understanding and in our technological capabilities. In this chapter, we stand back from the details and ask, “*Will all this progress lead to a general-purpose intelligent agent that can perform well 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 designing rational agents 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. When endowed with a learning component (Figure 2.15), this is the most general of our agent designs. Let’s see where the state of the art stands for each of the components.

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 to get real robots to work at all. The situation has changed rapidly in recent years with the availability of ready-made programmable robots. 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, gyroscopes, and actuators for an artificial flying insect (Floreano *et al.*, 2009). It may also be possible to combine millions of MEMS devices to produce powerful macroscopic actuators.

Thus, we see that AI systems are at the cusp of moving from primarily software-only systems to embedded robotic systems. The state of robotics today is roughly comparable to the state of personal computers in about 1980: at that time researchers and hobbyists could experiment with PCs, but it would take another decade before they became commonplace.

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 4 showed how to keep track of atomic state representations; Chapter 7 described how to do it for factored (propositional) state representations; Chapter 12 extended this to first-order logic; and Chapter 15 described **filtering** algorithms for probabilistic reasoning in uncertain environments. 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.” Detecting higher-level actions, such as “Dr. Russell is having a cup of tea with Dr. Norvig while discussing plans for next week,” is more difficult. Currently it can be done (see Figure 24.25 on page 961) only with the help of annotated examples.

Another problem is that, although the approximate filtering algorithms from Chapter 15 can handle quite large environments, they are still dealing with a factored representation—they have random variables, but do not represent objects and relations explicitly. Section 14.6 explained how probability and first-order logic can be combined to solve this problem, and

Section 14.6.3 showed how we can handle uncertainty about the identity of objects. We expect that the application of these ideas for tracking complex environments will yield huge benefits. However, we are still faced with a daunting task of defining general, reusable representation schemes for complex domains. As discussed in Chapter 12, we don't yet know how to do that in general; only for isolated, simple domains. It is possible that a new focus on probabilistic rather than logical representation coupled with aggressive machine learning (rather than hand-encoding of knowledge) will allow for progress.

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. We saw in Section 11.2 how to use hierarchical representations to handle problems of this scale; furthermore, work in **hierarchical reinforcement learning** has succeeded in combining some of these ideas with the techniques for decision making under uncertainty described in Chapter 17. As yet, algorithms for the partially observable case (POMDPs) are using the same atomic state representation we used for the search algorithms of Chapter 3. There is clearly a great deal of work to do here, but the technical foundations are largely in place. Section 27.2 discusses the question of how the search for effective long-range plans might be controlled.

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 21 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, reaching or exceeding human capabilities in many tasks—as long as we are dealing with a predefined vocabulary of features and concepts. 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. In computer vision, for example, learning complex concepts such as *Classroom* and *Cafeteria* would be made unnecessarily difficult if the agent were forced to work from pixels as the input representation; instead, the agent needs to be able to form intermediate concepts first, such as *Desk* and *Tray*, without explicit human supervision. Similar considerations apply to learning behavior: *HavingACupOfTea* is a very important

high-level step in many plans, but how does it get into an action library that initially contains much simpler actions such as *RaiseArm* and *Swallow*? Perhaps this will incorporate some of the ideas of **deep belief networks**—Bayesian networks that have multiple layers of hidden variables, as in the work of Hinton *et al.* (2006), Hawkins and Blakeslee (2004), and Bengio and LeCun (2007).

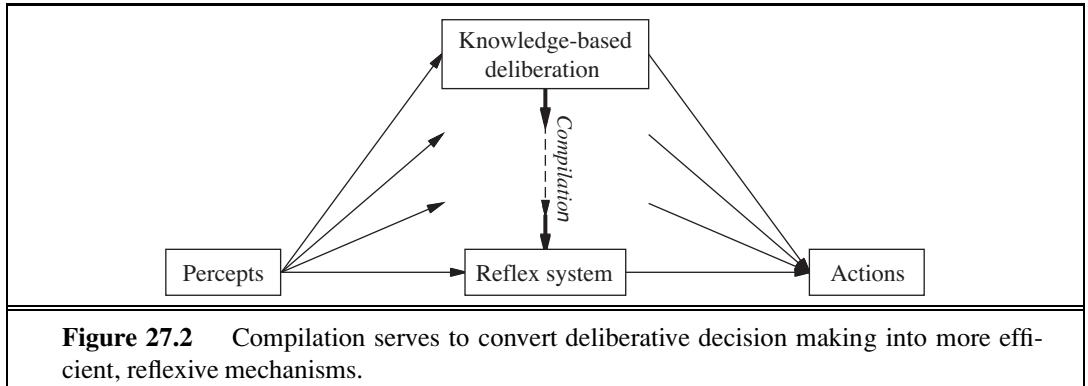
The vast majority of machine learning research today assumes a factored representation, learning a function $h : \mathbb{R}^n \rightarrow \mathbb{R}$ for regression and $h : \mathbb{R}^n \rightarrow \{0, 1\}$ for classification. Learning researchers will need to adapt their very successful techniques for factored representations to structured representations, particularly hierarchical representations. The work on inductive logic programming in Chapter 19 is a first step in this direction; the logical next step is to combine these ideas with the probabilistic languages of Section 14.6.

Unless we understand such issues, we are faced with the daunting task of constructing large commonsense knowledge bases by hand, an approach that has not fared well to date. There is great promise in using the Web as a source of natural language text, images, and videos to serve as a comprehensive knowledge base, but so far machine learning algorithms are limited in the amount of organized knowledge they can extract from these sources.

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.

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 *general* methods of controlling deliberation, rather than specific recipes for what to think about in each situation. The first useful idea is to employ **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. (See Section 3.5.4 for a brief description of metalevel decision making.) Example of an anytime algorithms include iterative deepening in game-tree search and MCMC in Bayesian networks.

The second technique for controlling deliberation is **decision-theoretic metareasoning** (Russell and Wefald, 1989, 1991; Horvitz, 1989; Horvitz and Breese, 1996). This method applies the theory of information value (Chapter 16) to the selection of individual computations. The value of a computation depends on both its cost (in terms of delaying action) and its benefits (in terms of improved decision quality). Metareasoning techniques can be used to design better search algorithms and to guarantee that the algorithms have the anytime property. Metareasoning is expensive, of course, and compilation methods can be applied so that the overhead is small compared to the costs of the computations being controlled. Metalevel reinforcement learning may provide another way to acquire effective policies for controlling deliberation: in essence, computations that lead to better decisions are reinforced, while those that turn out to have no effect are penalized. This approach avoids the myopia problems of the simple value-of-information calculation.

Metareasoning is one specific example of a **reflective architecture**—that is, an architecture that enables deliberation about the computational entities and actions occurring within the architecture itself. A theoretical foundation for reflective architectures can be built by defining a joint state space composed from the environment state and the computational state of the agent itself. Decision-making and learning algorithms can be designed that operate over this joint state space and thereby serve to implement and improve the agent's computational activities. Eventually, we expect task-specific algorithms such as alpha-beta search and backward chaining to disappear from AI systems, to be replaced by general methods that direct the agent's computations toward the efficient generation of high-quality decisions.

ANYTIME ALGORITHM

DECISION-THEORETIC METAREASONING

REFLECTIVE ARCHITECTURE

27.3 ARE WE GOING IN THE RIGHT DIRECTION?

The preceding section listed many advances and many opportunities for further progress. But where is this all leading? Dreyfus (1992) gives the analogy of trying to get to the moon by climbing a tree; one can report steady progress, all the way to the top of the tree. In this section, we consider whether AI's current path is more like a tree climb or a rocket trip.

In Chapter 1, we said that our goal was to build agents that *act rationally*. However, we also said that

... achieving perfect rationality—always doing the right thing—is not feasible in complicated environments. The computational demands are just too high. For most of the book, however, we will adopt the working hypothesis that perfect rationality is a good starting point for analysis.

Now it is time to consider again what exactly the goal of AI is. We want to build agents, but with what specification in mind? Here are four possibilities:

PERFECT
RATIONALITY

Perfect rationality. A perfectly rational agent acts at every instant in such a way as to maximize its expected utility, given the information it has acquired from the environment. We have seen that the calculations necessary to achieve perfect rationality in most environments are too time consuming, so perfect rationality is not a realistic goal.

CALCULATIVE
RATIONALITY

Calculative rationality. This is the notion of rationality that we have used implicitly in designing logical and decision-theoretic agents, and most of theoretical AI research has focused on this property. A calculatively rational agent *eventually* returns what *would have been* the rational choice at the beginning of its deliberation. This is an interesting property for a system to exhibit, but in most environments, the right answer at the wrong time is of no value. In practice, AI system designers are forced to compromise on decision quality to obtain reasonable overall performance; unfortunately, the theoretical basis of calculative rationality does not provide a well-founded way to make such compromises.

BOUNDED
RATIONALITY

Bounded rationality. Herbert Simon (1957) rejected the notion of perfect (or even approximately perfect) rationality and replaced it with bounded rationality, a descriptive theory of decision making by real agents. He wrote,

The capacity of the human mind for formulating and solving complex problems is very small compared with the size of the problems whose solution is required for objectively rational behavior in the real world—or even for a reasonable approximation to such objective rationality.

He suggested that bounded rationality works primarily by **satisficing**—that is, deliberating only long enough to come up with an answer that is “good enough.” Simon won the Nobel Prize in economics for this work and has written about it in depth (Simon, 1982). It appears to be a useful model of human behaviors in many cases. It is not a formal specification for intelligent agents, however, because the definition of “good enough” is not given by the theory. Furthermore, satisficing seems to be just one of a large range of methods used to cope with bounded resources.

Bounded optimality (BO). A bounded optimal agent behaves as well as possible, *given its computational resources*. That is, the expected utility of the agent program for a bounded optimal agent is at least as high as the expected utility of any other agent program running on the same machine.

Of these four possibilities, bounded optimality seems to offer the best hope for a strong theoretical foundation for AI. It has the advantage of being possible to achieve: there is always at least one best program—something that perfect rationality lacks. Bounded optimal agents are actually useful in the real world, whereas calculatively rational agents usually are not, and sacrificing agents might or might not be, depending on how ambitious they are.

The traditional approach in AI has been to start with calculative rationality and then make compromises to meet resource constraints. If the problems imposed by the constraints are minor, one would expect the final design to be similar to a BO agent design. But as the resource constraints become more critical—for example, as the environment becomes more complex—one would expect the two designs to diverge. In the theory of bounded optimality, these constraints can be handled in a principled fashion.

As yet, little is known about bounded optimality. It is possible to construct bounded optimal programs for very simple machines and for somewhat restricted kinds of environments (Etzioni, 1989; Russell *et al.*, 1993), but as yet we have no idea what BO programs are like for large, general-purpose computers in complex environments. If there is to be a constructive theory of bounded optimality, we have to hope that the design of bounded optimal programs does not depend too strongly on the details of the computer being used. It would make scientific research very difficult if adding a few kilobytes of memory to a gigabyte machine made a significant difference to the design of the BO program. One way to make sure this cannot happen is to be slightly more relaxed about the criteria for bounded optimality. By analogy with the notion of asymptotic complexity (Appendix A), we can define **asymptotic bounded optimality** (ABO) as follows (Russell and Subramanian, 1995). Suppose a program P is bounded optimal for a machine M in a class of environments \mathbf{E} , where the complexity of environments in \mathbf{E} is unbounded. Then program P' is ABO for M in \mathbf{E} if it can outperform P by running on a machine kM that is k times faster (or larger) than M . Unless k were enormous, we would be happy with a program that was ABO for a nontrivial environment on a nontrivial architecture. There would be little point in putting enormous effort into finding BO rather than ABO programs, because the size and speed of available machines tends to increase by a constant factor in a fixed amount of time anyway.

We can hazard a guess that BO or ABO programs for powerful computers in complex environments will not necessarily have a simple, elegant structure. We have already seen that general-purpose intelligence requires some reflex capability and some deliberative capability; a variety of forms of knowledge and decision making; learning and compilation mechanisms for all of those forms; methods for controlling reasoning; and a large store of domain-specific knowledge. A bounded optimal agent must adapt to the environment in which it finds itself, so that eventually its internal organization will reflect optimizations that are specific to the particular environment. This is only to be expected, and it is similar to the way in which racing cars restricted by engine capacity have evolved into extremely complex designs. We

suspect that a science of artificial intelligence based on bounded optimality will involve a good deal of study of the processes that allow an agent program to converge to bounded optimality and perhaps less concentration on the details of the messy programs that result.

In sum, the concept of bounded optimality is proposed as a formal task for AI research that is both well defined and feasible. Bounded optimality specifies optimal *programs* rather than optimal *actions*. Actions are, after all, generated by programs, and it is over programs that designers have control.

27.4 WHAT IF AI DOES SUCCEED?

In David Lodge's *Small World* (1984), a novel about the academic world of literary criticism, the protagonist causes consternation by asking a panel of eminent but contradictory literary theorists the following question: "*What if you were right?*" None of the theorists seems to have considered this question before, perhaps because debating unfalsifiable theories is an end in itself. Similar confusion can be evoked by asking AI researchers, "*What if you succeed?*"

As Section 26.3 relates, there are ethical issues to consider. Intelligent computers are more powerful than dumb ones, but will that power be used for good or ill? Those who strive to develop AI have a responsibility to see that the impact of their work is a positive one. The scope of the impact will depend on the degree of success of AI. Even modest successes in AI have already changed the ways in which computer science is taught (Stein, 2002) and software development is practiced. AI has made possible new applications such as speech recognition systems, inventory control systems, surveillance systems, robots, and search engines.

We can expect that medium-level successes in AI would affect all kinds of people in their daily lives. So far, computerized communication networks, such as cell phones and the Internet, have had this kind of pervasive effect on society, but AI has not. AI has been at work behind the scenes—for example, in automatically approving or denying credit card transactions for every purchase made on the Web—but has not been visible to the average consumer. We can imagine that truly useful personal assistants for the office or the home would have a large positive impact on people's lives, although they might cause some economic dislocation in the short term. Automated assistants for driving could prevent accidents, saving tens of thousands of lives per year. A technological capability at this level might also be applied to the development of autonomous weapons, which many view as undesirable. Some of the biggest societal problems we face today—such as the harnessing of genomic information for treating disease, the efficient management of energy resources, and the verification of treaties concerning nuclear weapons—are being addressed with the help of AI technologies.

Finally, it seems likely that a large-scale success in AI—the creation of human-level intelligence and beyond—would change the lives of a majority of humankind. The very nature of our work and play would be altered, as would our view of intelligence, consciousness, and the future destiny of the human race. AI systems at this level of capability could threaten human autonomy, freedom, and even survival. For these reasons, we cannot divorce AI research from its ethical consequences (see Section 26.3).

Which way will the future go? Science fiction authors seem to favor dystopian futures over utopian ones, probably because they make for more interesting plots. But so far, AI seems to fit in with other revolutionary technologies (printing, plumbing, air travel, telephony) whose negative repercussions are outweighed by their positive aspects.

In conclusion, we see that AI has made great progress in its short history, but the final sentence of Alan Turing's (1950) essay on *Computing Machinery and Intelligence* is still valid today:

*We can see only a short distance ahead,
but we can see that much remains to be done.*

A

MATHEMATICAL BACKGROUND

A.1 COMPLEXITY ANALYSIS AND O() NOTATION

BENCHMARKING

ANALYSIS OF ALGORITHMS

Computer scientists are often faced with the task of comparing algorithms to see how fast they run or how much memory they require. There are two approaches to this task. The first is **benchmarking**—running the algorithms on a computer and measuring speed in seconds and memory consumption in bytes. Ultimately, this is what really matters, but a benchmark can be unsatisfactory because it is so specific: it measures the performance of a particular program written in a particular language, running on a particular computer, with a particular compiler and particular input data. From the single result that the benchmark provides, it can be difficult to predict how well the algorithm would do on a different compiler, computer, or data set. The second approach relies on a mathematical **analysis of algorithms**, independently of the particular implementation and input, as discussed below.

A.1.1 Asymptotic analysis

We will consider algorithm analysis through the following example, a program to compute the sum of a sequence of numbers:

```
function SUMMATION(sequence) returns a number
    sum ← 0
    for i = 1 to LENGTH(sequence) do
        sum ← sum + sequence[i]
    return sum
```

The first step in the analysis is to abstract over the input, in order to find some parameter or parameters that characterize the size of the input. In this example, the input can be characterized by the length of the sequence, which we will call n . The second step is to abstract over the implementation, to find some measure that reflects the running time of the algorithm but is not tied to a particular compiler or computer. For the SUMMATION program, this could be just the number of lines of code executed, or it could be more detailed, measuring the number of additions, assignments, array references, and branches executed by the algorithm.

Either way gives us a characterization of the total number of steps taken by the algorithm as a function of the size of the input. We will call this characterization $T(n)$. If we count lines of code, we have $T(n) = 2n + 2$ for our example.

If all programs were as simple as SUMMATION, the analysis of algorithms would be a trivial field. But two problems make it more complicated. First, it is rare to find a parameter like n that completely characterizes the number of steps taken by an algorithm. Instead, the best we can usually do is compute the worst case $T_{\text{worst}}(n)$ or the average case $T_{\text{avg}}(n)$. Computing an average means that the analyst must assume some distribution of inputs.

The second problem is that algorithms tend to resist exact analysis. In that case, it is necessary to fall back on an approximation. We say that the SUMMATION algorithm is $O(n)$, meaning that its measure is at most a constant times n , with the possible exception of a few small values of n . More formally,

$$T(n) \text{ is } O(f(n)) \text{ if } T(n) \leq kf(n) \text{ for some } k, \text{ for all } n > n_0.$$

ASYMPTOTIC ANALYSIS

The $O()$ notation gives us what is called an **asymptotic analysis**. We can say without question that, as n asymptotically approaches infinity, an $O(n)$ algorithm is better than an $O(n^2)$ algorithm. A single benchmark figure could not substantiate such a claim.

The $O()$ notation abstracts over constant factors, which makes it easier to use, but less precise, than the $T()$ notation. For example, an $O(n^2)$ algorithm will always be worse than an $O(n)$ in the long run, but if the two algorithms are $T(n^2 + 1)$ and $T(100n + 1000)$, then the $O(n^2)$ algorithm is actually better for $n < 110$.

Despite this drawback, asymptotic analysis is the most widely used tool for analyzing algorithms. It is precisely because the analysis abstracts over both the exact number of operations (by ignoring the constant factor k) and the exact content of the input (by considering only its size n) that the analysis becomes mathematically feasible. The $O()$ notation is a good compromise between precision and ease of analysis.

A.1.2 NP and inherently hard problems

COMPLEXITY ANALYSIS

The analysis of algorithms and the $O()$ notation allow us to talk about the efficiency of a particular algorithm. However, they have nothing to say about whether there could be a better algorithm for the problem at hand. The field of **complexity analysis** analyzes problems rather than algorithms. The first gross division is between problems that can be solved in polynomial time and problems that cannot be solved in polynomial time, no matter what algorithm is used. The class of polynomial problems—those which can be solved in time $O(n^k)$ for some k —is called P. These are sometimes called “easy” problems, because the class contains those problems with running times like $O(\log n)$ and $O(n)$. But it also contains those with time $O(n^{1000})$, so the name “easy” should not be taken too literally.

Another important class of problems is NP, the class of nondeterministic polynomial problems. A problem is in this class if there is some algorithm that can guess a solution and then verify whether the guess is correct in polynomial time. The idea is that if you have an arbitrarily large number of processors, so that you can try all the guesses at once, or you are very lucky and always guess right the first time, then the NP problems become P problems. One of the biggest open questions in computer science is whether the class NP is equivalent

to the class P when one does not have the luxury of an infinite number of processors or omniscient guessing. Most computer scientists are convinced that $P \neq NP$; that NP problems are inherently hard and have no polynomial-time algorithms. But this has never been proven.

NP-COMPLETE Those who are interested in deciding whether $P = NP$ look at a subclass of NP called the **NP-complete** problems. The word “complete” is used here in the sense of “most extreme” and thus refers to the hardest problems in the class NP. It has been proven that either all the NP-complete problems are in P or none of them is. This makes the class theoretically interesting, but the class is also of practical interest because many important problems are known to be NP-complete. An example is the satisfiability problem: given a sentence of propositional logic, is there an assignment of truth values to the proposition symbols of the sentence that makes it true? Unless a miracle occurs and $P = NP$, there can be no algorithm that solves *all* satisfiability problems in polynomial time. However, AI is more interested in whether there are algorithms that perform efficiently on *typical* problems drawn from a pre-determined distribution; as we saw in Chapter 7, there are algorithms such as WALKSAT that do quite well on many problems.

CO-NP The class **co-NP** is the complement of NP, in the sense that, for every decision problem in NP, there is a corresponding problem in co-NP with the “yes” and “no” answers reversed. We know that P is a subset of both NP and co-NP, and it is believed that there are problems in co-NP that are not in P. The **co-NP-complete** problems are the hardest problems in co-NP.

CO-NP-COMPLETE The class #P (pronounced “sharp P”) is the set of counting problems corresponding to the decision problems in NP. Decision problems have a yes-or-no answer: is there a solution to this 3-SAT formula? Counting problems have an integer answer: how many solutions are there to this 3-SAT formula? In some cases, the counting problem is much harder than the decision problem. For example, deciding whether a bipartite graph has a perfect matching can be done in time $O(VE)$ (where the graph has V vertices and E edges), but the counting problem “how many perfect matches does this bipartite graph have” is #P-complete, meaning that it is hard as any problem in #P and thus at least as hard as any NP problem.

Another class is the class of PSPACE problems—those that require a polynomial amount of space, even on a nondeterministic machine. It is believed that PSPACE-hard problems are worse than NP-complete problems, although it could turn out that $NP = PSPACE$, just as it could turn out that $P = NP$.

A.2 VECTORS, MATRICES, AND LINEAR ALGEBRA

VECTOR Mathematicians define a **vector** as a member of a vector space, but we will use a more concrete definition: a vector is an ordered sequence of values. For example, in two-dimensional space, we have vectors such as $\mathbf{x} = \langle 3, 4 \rangle$ and $\mathbf{y} = \langle 0, 2 \rangle$. We follow the convention of bold-face characters for vector names, although some authors use arrows or bars over the names: \vec{x} or \bar{y} . The elements of a vector can be accessed using subscripts: $\mathbf{z} = \langle z_1, z_2, \dots, z_n \rangle$. One confusing point: this book is synthesizing work from many subfields, which variously call their sequences vectors, lists, or tuples, and variously use the notations $\langle 1, 2 \rangle$, $[1, 2]$, or $(1, 2)$.

The two fundamental operations on vectors are vector addition and scalar multiplication. The vector addition $\mathbf{x} + \mathbf{y}$ is the elementwise sum: $\mathbf{x} + \mathbf{y} = \langle 3 + 0, 4 + 2 \rangle = \langle 3, 6 \rangle$. Scalar multiplication multiplies each element by a constant: $5\mathbf{x} = \langle 5 \times 3, 5 \times 4 \rangle = \langle 15, 20 \rangle$.

The length of a vector is denoted $|\mathbf{x}|$ and is computed by taking the square root of the sum of the squares of the elements: $|\mathbf{x}| = \sqrt{(3^2 + 4^2)} = 5$. The dot product $\mathbf{x} \cdot \mathbf{y}$ (also called scalar product) of two vectors is the sum of the products of corresponding elements, that is, $\mathbf{x} \cdot \mathbf{y} = \sum_i x_i y_i$, or in our particular case, $\mathbf{x} \cdot \mathbf{y} = 3 \times 0 + 4 \times 2 = 8$.

Vectors are often interpreted as directed line segments (arrows) in an n -dimensional Euclidean space. Vector addition is then equivalent to placing the tail of one vector at the head of the other, and the dot product $\mathbf{x} \cdot \mathbf{y}$ is equal to $|\mathbf{x}| |\mathbf{y}| \cos \theta$, where θ is the angle between \mathbf{x} and \mathbf{y} .

MATRIX

A **matrix** is a rectangular array of values arranged into rows and columns. Here is a matrix \mathbf{A} of size 3×4 :

$$\begin{pmatrix} \mathbf{A}_{1,1} & \mathbf{A}_{1,2} & \mathbf{A}_{1,3} & \mathbf{A}_{1,4} \\ \mathbf{A}_{2,1} & \mathbf{A}_{2,2} & \mathbf{A}_{2,3} & \mathbf{A}_{2,4} \\ \mathbf{A}_{3,1} & \mathbf{A}_{3,2} & \mathbf{A}_{3,3} & \mathbf{A}_{3,4} \end{pmatrix}$$

The first index of $\mathbf{A}_{i,j}$ specifies the row and the second the column. In programming languages, $\mathbf{A}_{i,j}$ is often written $\mathbf{A}[i, j]$ or $\mathbf{A}[i][j]$.

The sum of two matrices is defined by adding their corresponding elements; for example $(\mathbf{A} + \mathbf{B})_{i,j} = \mathbf{A}_{i,j} + \mathbf{B}_{i,j}$. (The sum is undefined if \mathbf{A} and \mathbf{B} have different sizes.) We can also define the multiplication of a matrix by a scalar: $(c\mathbf{A})_{i,j} = c\mathbf{A}_{i,j}$. Matrix multiplication (the product of two matrices) is more complicated. The product \mathbf{AB} is defined only if \mathbf{A} is of size $a \times b$ and \mathbf{B} is of size $b \times c$ (i.e., the second matrix has the same number of rows as the first has columns); the result is a matrix of size $a \times c$. If the matrices are of appropriate size, then the result is

$$(\mathbf{AB})_{i,k} = \sum_j \mathbf{A}_{i,j} \mathbf{B}_{j,k}.$$

Matrix multiplication is not commutative, even for square matrices: $\mathbf{AB} \neq \mathbf{BA}$ in general. It is, however, associative: $(\mathbf{AB})\mathbf{C} = \mathbf{A}(\mathbf{BC})$. Note that the dot product can be expressed in terms of a transpose and a matrix multiplication: $\mathbf{x} \cdot \mathbf{y} = \mathbf{x}^\top \mathbf{y}$.

IDENTITY MATRIX

TRANSPOSE

INVERSE

SINGULAR

The **identity matrix** \mathbf{I} has elements $\mathbf{I}_{i,j}$ equal to 1 when $i = j$ and equal to 0 otherwise. It has the property that $\mathbf{AI} = \mathbf{A}$ for all \mathbf{A} . The **transpose** of \mathbf{A} , written \mathbf{A}^\top is formed by turning rows into columns and vice versa, or, more formally, by $\mathbf{A}^\top_{i,j} = \mathbf{A}_{j,i}$. The **inverse** of a square matrix \mathbf{A} is another square matrix \mathbf{A}^{-1} such that $\mathbf{A}^{-1}\mathbf{A} = \mathbf{I}$. For a **singular** matrix, the inverse does not exist. For a nonsingular matrix, it can be computed in $O(n^3)$ time.

Matrices are used to solve systems of linear equations in $O(n^3)$ time; the time is dominated by inverting a matrix of coefficients. Consider the following set of equations, for which we want a solution in x , y , and z :

$$\begin{aligned} +2x + y - z &= 8 \\ -3x - y + 2z &= -11 \\ -2x + y + 2z &= -3. \end{aligned}$$

We can represent this system as the matrix equation $\mathbf{A} \mathbf{x} = \mathbf{b}$, where

$$\mathbf{A} = \begin{pmatrix} 2 & 1 & -1 \\ -3 & -1 & 2 \\ -2 & 1 & 2 \end{pmatrix}, \quad \mathbf{x} = \begin{pmatrix} x \\ y \\ z \end{pmatrix}, \quad \mathbf{b} = \begin{pmatrix} 8 \\ -11 \\ -3 \end{pmatrix}.$$

To solve $\mathbf{A} \mathbf{x} = \mathbf{b}$ we multiply both sides by \mathbf{A}^{-1} , yielding $\mathbf{A}^{-1} \mathbf{A} \mathbf{x} = \mathbf{A}^{-1} \mathbf{b}$, which simplifies to $\mathbf{x} = \mathbf{A}^{-1} \mathbf{b}$. After inverting \mathbf{A} and multiplying by \mathbf{b} , we get the answer

$$\mathbf{x} = \begin{pmatrix} x \\ y \\ z \end{pmatrix} = \begin{pmatrix} 2 \\ 3 \\ -1 \end{pmatrix}.$$

A.3 PROBABILITY DISTRIBUTIONS

A probability is a measure over a set of events that satisfies three axioms:

1. The measure of each event is between 0 and 1. We write this as $0 \leq P(X = x_i) \leq 1$, where X is a random variable representing an event and x_i are the possible values of X . In general, random variables are denoted by uppercase letters and their values by lowercase letters.
2. The measure of the whole set is 1; that is, $\sum_{i=1}^n P(X = x_i) = 1$.
3. The probability of a union of disjoint events is the sum of the probabilities of the individual events; that is, $P(X = x_1 \vee X = x_2) = P(X = x_1) + P(X = x_2)$, where x_1 and x_2 are disjoint.

A **probabilistic model** consists of a sample space of mutually exclusive possible outcomes, together with a probability measure for each outcome. For example, in a model of the weather tomorrow, the outcomes might be *sunny*, *cloudy*, *rainy*, and *snowy*. A subset of these outcomes constitutes an event. For example, the event of precipitation is the subset consisting of $\{\text{rainy, snowy}\}$.

We use $\mathbf{P}(X)$ to denote the vector of values $\langle P(X = x_1), \dots, P(X = x_n) \rangle$. We also use $P(x_i)$ as an abbreviation for $P(X = x_i)$ and $\sum_x P(x)$ for $\sum_{i=1}^n P(X = x_i)$.

The conditional probability $P(B|A)$ is defined as $P(B \cap A)/P(A)$. A and B are conditionally independent if $P(B|A) = P(B)$ (or equivalently, $P(A|B) = P(A)$). For continuous variables, there are an infinite number of values, and unless there are point spikes, the probability of any one value is 0. Therefore, we define a **probability density function**, which we also denote as $P(\cdot)$, but which has a slightly different meaning from the discrete probability function. The density function $P(x)$ for a random variable X , which might be thought of as $P(X = x)$, is intuitively defined as the ratio of the probability that X falls into an interval around x , divided by the width of the interval, as the interval width goes to zero:

$$P(x) = \lim_{dx \rightarrow 0} P(x \leq X \leq x + dx)/dx.$$

The density function must be nonnegative for all x and must have

$$\int_{-\infty}^{\infty} P(x) dx = 1.$$

CUMULATIVE
PROBABILITY
DENSITY FUNCTION

We can also define a **cumulative probability density function** $F_X(x)$, which is the probability of a random variable being less than x :

$$F_X(x) = P(X \leq x) = \int_{-\infty}^x P(u) du.$$

Note that the probability density function has units, whereas the discrete probability function is unitless. For example, if values of X are measured in seconds, then the density is measured in Hz (i.e., 1/sec). If values of \mathbf{X} are points in three-dimensional space measured in meters, then density is measured in $1/m^3$.

GAUSSIAN
DISTRIBUTION

One of the most important probability distributions is the **Gaussian distribution**, also known as the **normal distribution**. A Gaussian distribution with mean μ and standard deviation σ (and therefore variance σ^2) is defined as

$$P(x) = \frac{1}{\sigma\sqrt{2\pi}} e^{-(x-\mu)^2/(2\sigma^2)},$$

STANDARD NORMAL
DISTRIBUTION
MULTIVARIATE
GAUSSIAN

where x is a continuous variable ranging from $-\infty$ to $+\infty$. With mean $\mu=0$ and variance $\sigma^2=1$, we get the special case of the **standard normal distribution**. For a distribution over a vector \mathbf{x} in n dimensions, there is the **multivariate Gaussian distribution**:

$$P(\mathbf{x}) = \frac{1}{\sqrt{(2\pi)^n |\Sigma|}} e^{-\frac{1}{2}((\mathbf{x}-\boldsymbol{\mu})^\top \Sigma^{-1} (\mathbf{x}-\boldsymbol{\mu}))},$$

CUMULATIVE
DISTRIBUTION

where $\boldsymbol{\mu}$ is the mean vector and Σ is the **covariance matrix** (see below).

In one dimension, we can define the **cumulative distribution** function $F(x)$ as the probability that a random variable will be less than x . For the normal distribution, this is

$$F(x) = \int_{-\infty}^x P(z) dz = \frac{1}{2} (1 + \text{erf}(\frac{z-\mu}{\sigma\sqrt{2}})),$$

CENTRAL LIMIT
THEOREM

where $\text{erf}(x)$ is the so-called **error function**, which has no closed-form representation.

The **central limit theorem** states that the distribution formed by sampling n independent random variables and taking their mean tends to a normal distribution as n tends to infinity. This holds for almost any collection of random variables, even if they are not strictly independent, unless the variance of any finite subset of variables dominates the others.

EXPECTATION

The **expectation** of a random variable, $E(X)$, is the mean or average value, weighted by the probability of each value. For a discrete variable it is:

$$E(X) = \sum_i x_i P(X = x_i).$$

For a continuous variable, replace the summation with an integral over the probability density function, $P(x)$:

$$E(X) = \int_{-\infty}^{\infty} x P(x) dx,$$

ROOT MEAN SQUARE

The **root mean square**, RMS, of a set of values (often samples of a random variable) is the square root of the mean of the squares of the values,

$$RMS(x_1, \dots, x_n) = \sqrt{\frac{x_1^2 + \dots + x_n^2}{n}}.$$

COVARIANCE

The **covariance** of two random variables is the expectation of the product of their differences from their means:

$$\text{cov}(X, Y) = E((X - \mu_X)(Y - \mu_Y)).$$

COVARIANCE MATRIX

The **covariance matrix**, often denoted Σ , is a matrix of covariances between elements of a vector of random variables. Given $\mathbf{X} = \langle X_1, \dots, X_n \rangle^\top$, the entries of the covariance matrix are as follows:

$$\Sigma_{i,j} = \text{cov}(X_i, X_j) = E((X_i - \mu_i)(X_j - \mu_j)).$$

A few more miscellaneous points: we use $\log(x)$ for the natural logarithm, $\log_e(x)$. We use $\text{argmax}_x f(x)$ for the value of x for which $f(x)$ is maximal.

BIBLIOGRAPHICAL AND HISTORICAL NOTES

The $O()$ notation so widely used in computer science today was first introduced in the context of number theory by the German mathematician P. G. H. Bachmann (1894). The concept of NP-completeness was invented by Cook (1971), and the modern method for establishing a reduction from one problem to another is due to Karp (1972). Cook and Karp have both won the Turing award, the highest honor in computer science, for their work.

Classic works on the analysis and design of algorithms include those by Knuth (1973) and Aho, Hopcroft, and Ullman (1974); more recent contributions are by Tarjan (1983) and Cormen, Leiserson, and Rivest (1990). These books place an emphasis on designing and analyzing algorithms to solve tractable problems. For the theory of NP-completeness and other forms of intractability, see Garey and Johnson (1979) or Papadimitriou (1994). Good texts on probability include Chung (1979), Ross (1988), and Bertsekas and Tsitsiklis (2008).

B

NOTES ON LANGUAGES AND ALGORITHMS

B.1 DEFINING LANGUAGES WITH BACKUS–NAUR FORM (BNF)

CONTEXT-FREE
GRAMMAR
BACKUS–NAUR
FORM (BNF)

TERMINAL SYMBOL

NONTERMINAL
SYMBOL

START SYMBOL

In this book, we define several languages, including the languages of propositional logic (page 243), first-order logic (page 293), and a subset of English (page 899). A formal language is defined as a set of strings where each string is a sequence of symbols. The languages we are interested in consist of an infinite set of strings, so we need a concise way to characterize the set. We do that with a **grammar**. The particular type of grammar we use is called a **context-free grammar**, because each expression has the same form in any context. We write our grammars in a formalism called **Backus–Naur form (BNF)**. There are four components to a BNF grammar:

- A set of **terminal symbols**. These are the symbols or words that make up the strings of the language. They could be letters (**A**, **B**, **C**, ...) or words (**a**, **aardvark**, **abacus**, ...), or whatever symbols are appropriate for the domain.
- A set of **nonterminal symbols** that categorize subphrases of the language. For example, the nonterminal symbol *NounPhrase* in English denotes an infinite set of strings including “you” and “the big slobbery dog.”
- A **start symbol**, which is the nonterminal symbol that denotes the complete set of strings of the language. In English, this is *Sentence*; for arithmetic, it might be *Expr*, and for programming languages it is *Program*.
- A set of **rewrite rules**, of the form $LHS \rightarrow RHS$, where LHS is a nonterminal symbol and RHS is a sequence of zero or more symbols. These can be either terminal symbols, or the symbol ϵ , which is used to denote the empty string.

A rewrite rule of the form

$$Sentence \rightarrow NounPhrase\ VerbPhrase$$

means that whenever we have two strings categorized as a *NounPhrase* and a *VerbPhrase*, we can append them together and categorize the result as a *Sentence*. As an abbreviation, the two rules ($S \rightarrow A$) and ($S \rightarrow B$) can be written ($S \rightarrow A \mid B$).

Here is a BNF grammar for simple arithmetic expressions:

$$\text{Expr} \rightarrow \text{Expr Operator Expr} \mid (\text{Expr}) \mid \text{Number}$$

$$\text{Number} \rightarrow \text{Digit} \mid \text{Number Digit}$$

$$\text{Digit} \rightarrow 0 \mid 1 \mid 2 \mid 3 \mid 4 \mid 5 \mid 6 \mid 7 \mid 8 \mid 9$$

$$\text{Operator} \rightarrow + \mid - \mid \div \mid \times$$

We cover languages and grammars in more detail in Chapter 22. Be aware that other books use slightly different notations for BNF; for example, you might see $\langle \text{Digit} \rangle$ instead of Digit for a nonterminal, ‘word’ instead of **word** for a terminal, or $::=$ instead of \rightarrow in a rule.

B.2 DESCRIBING ALGORITHMS WITH PSEUDOCODE

The algorithms in this book are described in pseudocode. Most of the pseudocode should be familiar to users of languages like Java, C++, or Lisp. In some places we use mathematical formulas or ordinary English to describe parts that would otherwise be more cumbersome. A few idiosyncrasies should be noted.

- **Persistent variables:** We use the keyword **persistent** to say that a variable is given an initial value the first time a function is called and retains that value (or the value given to it by a subsequent assignment statement) on all subsequent calls to the function. Thus, persistent variables are like global variables in that they outlive a single call to their function, but they are accessible only within the function. The agent programs in the book use persistent variables for *memory*. Programs with persistent variables can be implemented as *objects* in object-oriented languages such as C++, Java, Python, and Smalltalk. In functional languages, they can be implemented by *functional closures* over an environment containing the required variables.
- **Functions as values:** Functions and procedures have capitalized names, and variables have lowercase italic names. So most of the time, a function call looks like $\text{FN}(x)$. However, we allow the value of a variable to be a function; for example, if the value of the variable f is the square root function, then $f(9)$ returns 3.
- **for each:** The notation “**for each** x **in** c **do**” means that the loop is executed with the variable x bound to successive elements of the collection c .
- **Indentation is significant:** Indentation is used to mark the scope of a loop or conditional, as in the language Python, and unlike Java and C++ (which use braces) or Pascal and Visual Basic (which use **end**).
- **Destructuring assignment:** The notation “ $x, y \leftarrow \text{pair}$ ” means that the right-hand side must evaluate to a two-element tuple, and the first element is assigned to x and the second to y . The same idea is used in “**for each** x, y **in** $pairs$ **do**” and can be used to swap two variables: “ $x, y \leftarrow y, x$ ”
- **Generators and yield:** the notation “**generator** $G(x)$ **yields** numbers” defines G as a generator function. This is best understood by an example. The code fragment shown in

```

generator POWERS-OF-2() yields ints
  i  $\leftarrow$  1
  while true do
    yield i
    i  $\leftarrow$  2  $\times$  i
  for p in POWERS-OF-2() do
    PRINT(p)
  
```

Figure B.1 Example of a generator function and its invocation within a loop.

Figure B.1 prints the numbers 1, 2, 4, . . . , and never stops. The call to POWERS-OF-2 returns a generator, which in turn yields one value each time the loop code asks for the next element of the collection. Even though the collection is infinite, it is enumerated one element at a time.

- **Lists:** $[x, y, z]$ denotes a list of three elements. $[first|rest]$ denotes a list formed by adding *first* to the list *rest*. In Lisp, this is the `cons` function.
- **Sets:** $\{x, y, z\}$ denotes a set of three elements. $\{x : p(x)\}$ denotes the set of all elements *x* for which $p(x)$ is true.
- **Arrays start at 1:** Unless stated otherwise, the first index of an array is 1 as in usual mathematical notation, not 0, as in Java and C.

B.3 ONLINE HELP

Most of the algorithms in the book have been implemented in Java, Lisp, and Python at our online code repository:



aima.cs.berkeley.edu

The same Web site includes instructions for sending comments, corrections, or suggestions for improving the book, and for joining discussion lists.

Bibliography

The following abbreviations are used for frequently cited conferences and journals:

AAAI	Proceedings of the AAAI Conference on Artificial Intelligence
AAMAS	Proceedings of the International Conference on Autonomous Agents and Multi-agent Systems
ACL	Proceedings of the Annual Meeting of the Association for Computational Linguistics
AIJ	Artificial Intelligence
AIMag	AI Magazine
AIPS	Proceedings of the International Conference on AI Planning Systems
BBS	Behavioral and Brain Sciences
CACM	Communications of the Association for Computing Machinery
COGSCI	Proceedings of the Annual Conference of the Cognitive Science Society
COLING	Proceedings of the International Conference on Computational Linguistics
COLT	Proceedings of the Annual ACM Workshop on Computational Learning Theory
CP	Proceedings of the International Conference on Principles and Practice of Constraint Programming
CVPR	Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition
EC	Proceedings of the ACM Conference on Electronic Commerce
ECAI	Proceedings of the European Conference on Artificial Intelligence
ECCV	Proceedings of the European Conference on Computer Vision
ECML	Proceedings of the The European Conference on Machine Learning
ECP	Proceedings of the European Conference on Planning
FGCS	Proceedings of the International Conference on Fifth Generation Computer Systems
FOCS	Proceedings of the Annual Symposium on Foundations of Computer Science
ICAPS	Proceedings of the International Conference on Automated Planning and Scheduling
ICASSP	Proceedings of the International Conference on Acoustics, Speech, and Signal Processing
ICCV	Proceedings of the International Conference on Computer Vision
ICLP	Proceedings of the International Conference on Logic Programming
ICML	Proceedings of the International Conference on Machine Learning
ICPR	Proceedings of the International Conference on Pattern Recognition
ICRA	Proceedings of the IEEE International Conference on Robotics and Automation
ICSLP	Proceedings of the International Conference on Speech and Language Processing
IJAR	International Journal of Approximate Reasoning
IJCAI	Proceedings of the International Joint Conference on Artificial Intelligence
IJCNN	Proceedings of the International Joint Conference on Neural Networks
IJCV	International Journal of Computer Vision
ILP	Proceedings of the International Workshop on Inductive Logic Programming
ISMIS	Proceedings of the International Symposium on Methodologies for Intelligent Systems
ISRR	Proceedings of the International Symposium on Robotics Research
JACM	Journal of the Association for Computing Machinery
JAIR	Journal of Artificial Intelligence Research
JAR	Journal of Automated Reasoning
JASA	Journal of the American Statistical Association
JMLR	Journal of Machine Learning Research
JSL	Journal of Symbolic Logic
KDD	Proceedings of the International Conference on Knowledge Discovery and Data Mining
KR	Proceedings of the International Conference on Principles of Knowledge Representation and Reasoning
LICS	Proceedings of the IEEE Symposium on Logic in Computer Science
NIPS	Advances in Neural Information Processing Systems
PAMI	IEEE Transactions on Pattern Analysis and Machine Intelligence
PNAS	Proceedings of the National Academy of Sciences of the United States of America
PODS	Proceedings of the ACM International Symposium on Principles of Database Systems
SIGIR	Proceedings of the Special Interest Group on Information Retrieval
SIGMOD	Proceedings of the ACM SIGMOD International Conference on Management of Data
SODA	Proceedings of the Annual ACM-SIAM Symposium on Discrete Algorithms
STOC	Proceedings of the Annual ACM Symposium on Theory of Computing
TARK	Proceedings of the Conference on Theoretical Aspects of Reasoning about Knowledge
UAI	Proceedings of the Conference on Uncertainty in Artificial Intelligence

- Aarup**, M., Arentoft, M. M., Parrod, Y., Stader, J., and Stokes, I. (1994). OPTIMUM-AIV: A knowledge-based planning and scheduling system for spacecraft AIV. In Fox, M. and Zweber, M. (Eds.), *Knowledge Based Scheduling*. Morgan Kaufmann.
- Abney**, S. (2007). *Semisupervised Learning for Computational Linguistics*. CRC Press.
- Abramson**, B. and Yung, M. (1989). Divide and conquer under global constraints: A solution to the N-queens problem. *J. Parallel and Distributed Computing*, 6(3), 649–662.
- Achlioptas**, D. (2009). Random satisfiability. In Biere, A., Heule, M., van Maaren, H., and Walsh, T. (Eds.), *Handbook of Satisfiability*. IOS Press.
- Achlioptas**, D., Beame, P., and Molloy, M. (2004). Exponential bounds for DPLL below the satisfiability threshold. In *SODA-04*.
- Achlioptas**, D., Naor, A., and Peres, Y. (2007). On the maximum satisfiability of random formulas. *JACM*, 54(2).
- Achlioptas**, D. and Peres, Y. (2004). The threshold for random k -SAT is $2k \log 2 - o(k)$. *J. American Mathematical Society*, 17(4), 947–973.
- Ackley**, D. H. and Littman, M. L. (1991). Interactions between learning and evolution. In Langton, C., Taylor, C., Farmer, J. D., and Ramussen, S. (Eds.), *Artificial Life II*, pp. 487–509. Addison-Wesley.
- Adelson-Velsky**, G. M., Arlazarov, V. L., Bitman, A. R., Zhivotovsky, A. A., and Uskov, A. V. (1970). Programming a computer to play chess. *Russian Mathematical Surveys*, 25, 221–262.
- Adida**, B. and Birbeck, M. (2008). RDFa primer. Tech. rep., W3C.
- Agerbeck**, C. and Hansen, M. O. (2008). A multi-agent approach to solving *NP*-complete problems. Master's thesis, Technical Univ. of Denmark.
- Aggarwal**, G., Goel, A., and Motwani, R. (2006). Truthful auctions for pricing search keywords. In *EC-06*, pp. 1–7.
- Agichtein**, E. and Gravano, L. (2003). Querying text databases for efficient information extraction. In *Proc. IEEE Conference on Data Engineering*.
- Agmon**, S. (1954). The relaxation method for linear inequalities. *Canadian Journal of Mathematics*, 6(3), 382–392.
- Agre**, P. E. and Chapman, D. (1987). Pengi: an implementation of a theory of activity. In *IJCAI-87*, pp. 268–272.
- Aho**, A. V., Hopcroft, J., and Ullman, J. D. (1974). *The Design and Analysis of Computer Algorithms*. Addison-Wesley.
- Aizerman**, M., Braverman, E., and Rozonoer, L. (1964). Theoretical foundations of the potential function method in pattern recognition learning. *Automation and Remote Control*, 25, 821–837.
- Al-Chang**, M., Bresina, J., Charest, L., Chase, A., Hsu, J., Jonsson, A., Kanefsky, B., Morris, P., Rajan, K., Yglesias, J., Chafin, B., Dias, W., and Mal dague, P. (2004). MAPGEN: Mixed-Initiative planning and scheduling for the Mars Exploration Rover mission. *IEEE Intelligent Systems*, 19(1), 8–12.
- Albus**, J. S. (1975). A new approach to manipulator control: The cerebellar model articulation controller (CMAC). *J. Dynamic Systems, Measurement, and Control*, 97, 270–277.
- Aldous**, D. and Vazirani, U. (1994). “Go with the winners” algorithms. In *FOCS-94*, pp. 492–501.
- Alekhnovich**, M., Hirsch, E. A., and Itsykson, D. (2005). Exponential lower bounds for the running time of DPLL algorithms on satisfiable formulas. *JAR*, 35(1–3), 51–72.
- Allais**, M. (1953). Le comportement de l'homme rationnel devant la risque: critique des postulats et axiomes de l'école Américaine. *Econometrica*, 21, 503–546.
- Allen**, J. F. (1983). Maintaining knowledge about temporal intervals. *CACM*, 26(11), 832–843.
- Allen**, J. F. (1984). Towards a general theory of action and time. *AII*, 23, 123–154.
- Allen**, J. F. (1991). Time and time again: The many ways to represent time. *Int. J. Intelligent Systems*, 6, 341–355.
- Allen**, J. F., Hendler, J., and Tate, A. (Eds.). (1990). *Readings in Planning*. Morgan Kaufmann.
- Allis**, L. (1988). A knowledge-based approach to connect four. The game is solved: White wins. Master's thesis, Vrije Univ., Amsterdam.
- Almuallim**, H. and Dietterich, T. (1991). Learning with many irrelevant features. In *AAAI-91*, Vol. 2, pp. 547–552.
- ALPAC** (1966). Language and machines: Computers in translation and linguistics. Tech. rep. 1416, The Automatic Language Processing Advisory Committee of the National Academy of Sciences.
- Alterman**, R. (1988). Adaptive planning. *Cognitive Science*, 12, 393–422.
- Amarel**, S. (1967). An approach to heuristic problem-solving and theorem proving in the propositional calculus. In Hart, J. and Takasu, S. (Eds.), *Systems and Computer Science*. University of Toronto Press.
- Amarel**, S. (1968). On representations of problems of reasoning about actions. In Michie, D. (Ed.), *Machine Intelligence 3*, Vol. 3, pp. 131–171. Elsevier/North-Holland.
- Amir**, E. and Russell, S. J. (2003). Logical filtering. In *IJCAI-03*.
- Amit**, D., Gutfreund, H., and Sompolinsky, H. (1985). Spin-glass models of neural networks. *Physical Review A*, 32, 1007–1018.
- Andersen**, S. K., Olesen, K. G., Jensen, F. V., and Jensen, F. (1989). HUGIN—A shell for building Bayesian belief universes for expert systems. In *IJCAI-89*, Vol. 2, pp. 1080–1085.
- Anderson**, J. R. (1980). *Cognitive Psychology and Its Implications*. W. H. Freeman.
- Anderson**, J. R. (1983). *The Architecture of Cognition*. Harvard University Press.
- Andoni**, A. and Indyk, P. (2006). Near-optimal hashing algorithms for approximate nearest neighbor in high dimensions. In *FOCS-06*.
- Andre**, D. and Russell, S. J. (2002). State abstraction for programmable reinforcement learning agents. In *AAAI-02*, pp. 119–125.
- Anthony**, M. and Bartlett, P. (1999). *Neural Network Learning: Theoretical Foundations*. Cambridge University Press.
- Aoki**, M. (1965). Optimal control of partially observable Markov systems. *J. Franklin Institute*, 280(5), 367–386.
- Appel**, K. and Haken, W. (1977). Every planar map is four colorable: Part I: Discharging. *Illinois J. Math.*, 21, 429–490.
- Appelt**, D. (1999). Introduction to information extraction. *CACM*, 12(3), 161–172.
- Apt**, K. R. (1999). The essence of constraint propagation. *Theoretical Computer Science*, 221(1–2), 179–210.
- Apt**, K. R. (2003). *Principles of Constraint Programming*. Cambridge University Press.
- Apté**, C., Damerau, F., and Weiss, S. (1994). Automated learning of decision rules for text categorization. *ACM Transactions on Information Systems*, 12, 233–251.
- Arbuthnot**, J. (1692). *Of the Laws of Chance*. Motte, London. Translation into English, with additions, of Huygens (1657).
- Archibald**, C., Altman, A., and Shoham, Y. (2009). Analysis of a winning computational billiards player. In *IJCAI-09*.
- Ariely**, D. (2009). *Predictably Irrational* (Revised edition). Harper.
- Arkin**, R. (1998). *Behavior-Based Robotics*. MIT Press.
- Armando**, A., Carboni, R., Compagna, L., Cuel lar, J., and Tobarra, L. (2008). Formal analysis of SAML 2.0 web browser single sign-on: Breaking the SAML-based single sign-on for google apps. In *FMSE '08: Proc. 6th ACM workshop on Formal methods in security engineering*, pp. 1–10.
- Arnould**, A. (1662). *La logique, ou l'art de penser*. Chez Charles Savreux, au pied de la Tour de Notre Dame, Paris.
- Arora**, S. (1998). Polynomial time approximation schemes for Euclidean traveling salesman and other geometric problems. *JACM*, 45(5), 753–782.
- Arunachalam**, R. and Sadeh, N. M. (2005). The supply chain trading agent competition. *Electronic Commerce Research and Applications*, Spring, 66–84.
- Ashby**, W. R. (1940). Adaptiveness and equilibrium. *J. Mental Science*, 86, 478–483.
- Ashby**, W. R. (1948). Design for a brain. *Electronic Engineering, December*, 379–383.
- Ashby**, W. R. (1952). *Design for a Brain*. Wiley.
- Asimov**, I. (1942). Runaround. *Astounding Science Fiction*, March.
- Asimov**, I. (1950). *I, Robot*. Doubleday.
- Astrom**, K. J. (1965). Optimal control of Markov decision processes with incomplete state estimation. *J. Math. Anal. Applic.*, 10, 174–205.
- Audi**, R. (Ed.). (1999). *The Cambridge Dictionary of Philosophy*. Cambridge University Press.
- Axelrod**, R. (1985). *The Evolution of Cooperation*. Basic Books.
- Baader**, F., Calvanese, D., McGuinness, D., Nardi, D., and Patel-Schneider, P. (2007). *The Description Logic Handbook* (2nd edition). Cambridge University Press.
- Baader**, F. and Snyder, W. (2001). Unification theory. In Robinson, J. and Voronkov, A. (Eds.), *Handbook of Automated Reasoning*, pp. 447–533. Elsevier.
- Bacchus**, F. (1990). *Representing and Reasoning with Probabilistic Knowledge*. MIT Press.
- Bacchus**, F. and Grove, A. (1995). Graphical models for preference and utility. In *UAI-95*, pp. 3–10.
- Bacchus**, F. and Grove, A. (1996). Utility independence in a qualitative decision theory. In *KR-96*, pp. 542–552.

- Bacchus**, F., Grove, A., Halpern, J. Y., and Koller, D. (1992). From statistics to beliefs. In *AAAI-92*, pp. 602–608.
- Bacchus**, F. and van Beek, P. (1998). On the conversion between non-binary and binary constraint satisfaction problems. In *AAAI-98*, pp. 311–318.
- Bacchus**, F. and van Run, P. (1995). Dynamic variable ordering in CSPs. In *CP-95*, pp. 258–275.
- Bachmann**, P. G. H. (1894). *Die analytische Zahlentheorie*. B. G. Teubner, Leipzig.
- Backus**, J. W. (1996). Transcript of question and answer session. In Wexelblat, R. L. (Ed.), *History of Programming Languages*, p. 162. Academic Press.
- Bagnell**, J. A. and Schneider, J. (2001). Autonomous helicopter control using reinforcement learning policy search methods. In *ICRA-01*.
- Baker**, J. (1975). The Dragon system—An overview. *IEEE Transactions on Acoustics; Speech; and Signal Processing*, 23, 24–29.
- Baker**, J. (1979). Trainable grammars for speech recognition. In *Speech Communication Papers for the 97th Meeting of the Acoustical Society of America*, pp. 547–550.
- Baldi**, P., Chauvin, Y., Hunkapiller, T., and McClure, M. (1994). Hidden Markov models of biological primary sequence information. *PNAS*, 91(3), 1059–1063.
- Baldwin**, J. M. (1896). A new factor in evolution. *American Naturalist*, 30, 441–451. Continued on pages 536–553.
- Ballard**, B. W. (1983). The *-minimax search procedure for trees containing chance nodes. *AIJ*, 21(3), 327–350.
- Baluja**, S. (1997). Genetic algorithms and explicit search statistics. In Mozer, M. C., Jordan, M. I., and Petsche, T. (Eds.), *NIPS 9*, pp. 319–325. MIT Press.
- Bancilhon**, F., Maier, D., Sagiv, Y., and Ullman, J. D. (1986). Magic sets and other strange ways to implement logic programs. In *PODS-86*, pp. 1–16.
- Banko**, M. and Brill, E. (2001). Scaling to very very large corpora for natural language disambiguation. In *ACL-01*, pp. 26–33.
- Banko**, M., Brill, E., Dumais, S. T., and Lin, J. (2002). Askmsr: Question answering using the worldwide web. In *Proc. AAAI Spring Symposium on Mining Answers from Texts and Knowledge Bases*, pp. 7–9.
- Banko**, M., Cafarella, M. J., Soderland, S., Broadhead, M., and Etzioni, O. (2007). Open information extraction from the web. In *IJCAI-07*.
- Banko**, M. and Etzioni, O. (2008). The tradeoffs between open and traditional relation extraction. In *ACL-08*, pp. 28–36.
- Bar-Hillel**, Y. (1954). Indexical expressions. *Mind*, 63, 359–379.
- Bar-Hillel**, Y. (1960). The present status of automatic translation of languages. In Alt, F. L. (Ed.), *Advances in Computers*, Vol. 1, pp. 91–163. Academic Press.
- Bar-Shalom**, Y. (Ed.). (1992). *Multitarget-multisensor tracking: Advanced applications*. Artech House.
- Bar-Shalom**, Y. and Fortmann, T. E. (1988). *Tracking and Data Association*. Academic Press.
- Bartak**, R. (2001). Theory and practice of constraint propagation. In *Proc. Third Workshop on Constraint Programming for Decision and Control (CPDC-01)*, pp. 7–14.
- Barto**, A. G., Bradtko, S. J., and Singh, S. P. (1995). Learning to act using real-time dynamic programming. *AIJ*, 73(1), 81–138.
- Barto**, A. G., Sutton, R. S., and Anderson, C. W. (1983). Neuron-like adaptive elements that can solve difficult learning control problems. *IEEE Transactions on Systems, Man and Cybernetics*, 13, 834–846.
- Barto**, A. G., Sutton, R. S., and Brouwer, P. S. (1981). Associative search network: A reinforcement learning associative memory. *Biological Cybernetics*, 40(3), 201–211.
- Barwise**, J. and Etchemendy, J. (1993). *The Language of First-Order Logic: Including the Macintosh Program Tarski's World 4.0* (Third Revised and Expanded edition). Center for the Study of Language and Information (CSLI).
- Barwise**, J. and Etchemendy, J. (2002). *Language, Proof and Logic*. CSLI (Univ. of Chicago Press).
- Baum**, E., Boneh, D., and Garrett, C. (1995). On genetic algorithms. In *COLT-95*, pp. 230–239.
- Baum**, E. and Haussler, D. (1989). What size net gives valid generalization? *Neural Computation*, 1(1), 151–160.
- Baum**, E. and Smith, W. D. (1997). A Bayesian approach to relevance in game playing. *AIJ*, 97(1–2), 195–242.
- Baum**, E. and Wilczek, F. (1988). Supervised learning of probability distributions by neural networks. In Anderson, D. Z. (Ed.), *Neural Information Processing Systems*, pp. 52–61. American Institute of Physics.
- Baum**, L. E. and Petrie, T. (1966). Statistical inference for probabilistic functions of finite state Markov chains. *Annals of Mathematical Statistics*, 41.
- Baxter**, J. and Bartlett, P. (2000). Reinforcement learning in POMDP's via direct gradient ascent. In *ICML-00*, pp. 41–48.
- Bayardo**, R. J. and Miranker, D. P. (1994). An optimal backtrack algorithm for tree-structured constraint satisfaction problems. *AIJ*, 71(1), 159–181.
- Bayardo**, R. J. and Schrag, R. C. (1997). Using CSP look-back techniques to solve real-world SAT instances. In *AAAI-97*, pp. 203–208.
- Bayes**, T. (1763). An essay towards solving a problem in the doctrine of chances. *Philosophical Transactions of the Royal Society of London*, 53, 370–418.
- Beal**, D. F. (1980). An analysis of minimax. In Clarke, M. R. B. (Ed.), *Advances in Computer Chess 2*, pp. 103–109. Edinburgh University Press.
- Beal**, J. and Winston, P. H. (2009). The new frontier of human-level artificial intelligence. *IEEE Intelligent Systems*, 24(4), 21–23.
- Beckert**, B. and Posegga, J. (1995). Leantap: Lean, tableau-based deduction. *JAR*, 15(3), 339–358.
- Beeri**, C., Fagin, R., Maier, D., and Yannakakis, M. (1983). On the desirability of acyclic database schemes. *JACM*, 30(3), 479–513.
- Bekey**, G. (2008). *Robotics: State Of The Art And Future Challenges*. Imperial College Press.
- Bell**, C. and Tate, A. (1985). Using temporal constraints to restrict search in a planner. In *Proc. Third Alvey IKBS SIG Workshop*.
- Bell**, J. L. and Machover, M. (1977). *A Course in Mathematical Logic*. Elsevier/North-Holland.
- Bellman**, R. E. (1952). On the theory of dynamic programming. *PNAS*, 38, 716–719.
- Bellman**, R. E. (1961). *Adaptive Control Processes: A Guided Tour*. Princeton University Press.
- Bellman**, R. E. (1965). On the application of dynamic programming to the determination of optimal play in chess and checkers. *PNAS*, 53, 244–246.
- Bellman**, R. E. (1978). *An Introduction to Artificial Intelligence: Can Computers Think?* Boyd & Fraser Publishing Company.
- Bellman**, R. E. (1984). *Eye of the Hurricane*. World Scientific.
- Bellman**, R. E. and Dreyfus, S. E. (1962). *Applied Dynamic Programming*. Princeton University Press.
- Bellman**, R. E. (1957). *Dynamic Programming*. Princeton University Press.
- Belongie**, S., Malik, J., and Puzicha, J. (2002). Shape matching and object recognition using shape contexts. *PAMI*, 24(4), 509–522.
- Ben-Tal**, A. and Nemirovski, A. (2001). *Lectures on Modern Convex Optimization: Analysis, Algorithms, and Engineering Applications*. SIAM (Society for Industrial and Applied Mathematics).
- Bengio**, Y. and LeCun, Y. (2007). Scaling learning algorithms towards AI. In Bottou, L., Chapelle, O., DeCoste, D., and Weston, J. (Eds.), *Large-Scale Kernel Machines*. MIT Press.
- Bentham**, J. (1823). *Principles of Morals and Legislation*. Oxford University Press, Oxford, UK. Original work published in 1789.
- Berger**, J. O. (1985). *Statistical Decision Theory and Bayesian Analysis*. Springer Verlag.
- Berkson**, J. (1944). Application of the logistic function to bio-assay. *JASA*, 39, 357–365.
- Berlekamp**, E. R., Conway, J. H., and Guy, R. K. (1982). *Winning Ways, For Your Mathematical Plays*. Academic Press.
- Berlekamp**, E. R. and Wolfe, D. (1994). *Mathematical Go: Chilling Gets the Last Point*. A.K. Peters.
- Berleur**, J. and Brunnstein, K. (2001). *Ethics of Computing: Codes, Spaces for Discussion and Law*. Chapman and Hall.
- Berliner**, H. J. (1979). The B* tree search algorithm: A best-first proof procedure. *AIJ*, 12(1), 23–40.
- Berliner**, H. J. (1980a). Backgammon computer program beats world champion. *AIJ*, 14, 205–220.
- Berliner**, H. J. (1980b). Computer backgammon. *Scientific American*, 249(6), 64–72.
- Bernardo**, J. M. and Smith, A. F. M. (1994). *Bayesian Theory*. Wiley.
- Berners-Lee**, T., Hendler, J., and Lassila, O. (2001). The semantic web. *Scientific American*, 284(5), 34–43.
- Bernoulli**, D. (1738). Specimen theoriae novae de mensura sortis. *Proc. St. Petersburg Imperial Academy of Sciences*, 5, 175–192.
- Bernstein**, A. and Roberts, M. (1958). Computer vs. chess player. *Scientific American*, 198(6), 96–105.
- Bernstein**, P. L. (1996). *Against the Odds: The Remarkable Story of Risk*. Wiley.
- Berrou**, C., Glavieux, A., and Thitimajshima, P. (1993). Near Shannon limit error control-correcting coding and decoding: Turbo-codes. 1. In *Proc. IEEE International Conference on Communications*, pp. 1064–1070.
- Berry**, D. A. and Fristedt, B. (1985). *Bandit Problems: Sequential Allocation of Experiments*. Chapman and Hall.

- Bertele**, U. and Brioschi, F. (1972). *Nonserial dynamic programming*. Academic Press.
- Bertoli**, P., Cimatti, A., and Roveri, M. (2001a). Heuristic search + symbolic model checking = efficient conformant planning. In *IJCAI-01*, pp. 467–472.
- Bertoli**, P., Cimatti, A., Roveri, M., and Traverso, P. (2001b). Planning in nondeterministic domains under partial observability via symbolic model checking. In *IJCAI-01*, pp. 473–478.
- Bertot**, Y., Casteran, P., Huet, G., and Paulin-Mohring, C. (2004). *Interactive Theorem Proving and Program Development*. Springer.
- Bertsekas**, D. (1987). *Dynamic Programming: Deterministic and Stochastic Models*. Prentice-Hall.
- Bertsekas**, D. and Tsitsiklis, J. N. (1996). *Neurodynamic programming*. Athena Scientific.
- Bertsekas**, D. and Tsitsiklis, J. N. (2008). *Introduction to Probability* (2nd edition). Athena Scientific.
- Bertsekas**, D. and Shreve, S. E. (2007). *Stochastic Optimal Control: The Discrete-Time Case*. Athena Scientific.
- Bessière**, C. (2006). Constraint propagation. In Rossi, F., van Beek, P., and Walsh, T. (Eds.), *Handbook of Constraint Programming*. Elsevier.
- Bhar**, R. and Hamori, S. (2004). *Hidden Markov Models: Applications to Financial Economics*. Springer.
- Bibel**, W. (1993). *Deduction: Automated Logic*. Academic Press.
- Biere**, A., Heule, M., van Maaren, H., and Walsh, T. (Eds.). (2009). *Handbook of Satisfiability*. IOS Press.
- Billings**, D., Burch, N., Davidson, A., Holte, R., Schaeffer, J., Schauenberg, T., and Szafrań, D. (2003). Approximating game-theoretic optimal strategies for full-scale poker. In *IJCAI-03*.
- Binder**, J., Koller, D., Russell, S. J., and Kanazawa, K. (1997a). Adaptive probabilistic networks with hidden variables. *Machine Learning*, 29, 213–244.
- Binder**, J., Murphy, K., and Russell, S. J. (1997b). Space-efficient inference in dynamic probabilistic networks. In *IJCAI-97*, pp. 1292–1296.
- Binford**, T. O. (1971). Visual perception by computer. Invited paper presented at the IEEE Systems Science and Cybernetics Conference, Miami.
- Binmore**, K. (1982). *Essays on Foundations of Game Theory*. Pitman.
- Bishop**, C. M. (1995). *Neural Networks for Pattern Recognition*. Oxford University Press.
- Bishop**, C. M. (2007). *Pattern Recognition and Machine Learning*. Springer-Verlag.
- Bisson**, T. (1990). They're made out of meat. *Omni Magazine*.
- Bistarelli**, S., Montanari, U., and Rossi, F. (1997). Semiring-based constraint satisfaction and optimization. *JACM*, 44(2), 201–236.
- Bitner**, J. R. and Reingold, E. M. (1975). Backtrack programming techniques. *CACM*, 18(11), 651–656.
- Bizer**, C., Auer, S., Kobilarov, G., Lehmann, J., and Cyganiak, R. (2007). DBpedia – querying wikipedia like a database. In *Developers Track Presentation at the 16th International Conference on World Wide Web*.
- Blazewicz**, J., Ecker, K., Pesch, E., Schmidt, G., and Weglarz, J. (2007). *Handbook on Scheduling: Models and Methods for Advanced Planning* (*International Handbooks on Information Systems*). Springer-Verlag New York, Inc.
- Blei**, D. M., Ng, A. Y., and Jordan, M. I. (2001). Latent Dirichlet Allocation. In *Neural Information Processing Systems*, Vol. 14.
- Blinder**, A. S. (1983). Issues in the coordination of monetary and fiscal policies. In *Monetary Policy Issues in the 1980s*. Federal Reserve Bank, Kansas City, Missouri.
- Bliss**, C. I. (1934). The method of probits. *Science*, 79(2037), 38–39.
- Block**, H. D., Knight, B., and Rosenblatt, F. (1962). Analysis of a four-layer series-coupled perceptron. *Rev. Modern Physics*, 34(1), 275–282.
- Blum**, A. L. and Furst, M. (1995). Fast planning through planning graph analysis. In *IJCAI-95*, pp. 1636–1642.
- Blum**, A. L. and Furst, M. (1997). Fast planning through planning graph analysis. *AIJ*, 90(1–2), 281–300.
- Blum**, A. L. (1996). On-line algorithms in machine learning. In *Proc. Workshop on On-Line Algorithms, Dagstuhl*, pp. 306–325.
- Blum**, A. L. and Mitchell, T. M. (1998). Combining labeled and unlabeled data with co-training. In *COLT-98*, pp. 92–100.
- Blumer**, A., Ehrenfeucht, A., Haussler, D., and Warmuth, M. (1989). Learnability and the Vapnik-Chervonenkis dimension. *JACM*, 36(4), 929–965.
- Bobrow**, D. G. (1967). Natural language input for a computer problem solving system. In Minsky, M. L. (Ed.), *Semantic Information Processing*, pp. 133–215. MIT Press.
- Bobrow**, D. G., Kaplan, R., Kay, M., Norman, D. A., Thompson, H., and Winograd, T. (1977). GUS, a frame driven dialog system. *AIJ*, 8, 155–173.
- Boden**, M. A. (1977). *Artificial Intelligence and Natural Man*. Basic Books.
- Boden**, M. A. (Ed.). (1990). *The Philosophy of Artificial Intelligence*. Oxford University Press.
- Bolognesi**, A. and Ciancarini, P. (2003). Computer programming of kriegspiel endings: The case of KR vs. k. In *Advances in Computer Games 10*.
- Bonet**, B. (2002). An epsilon-optimal grid-based algorithm for partially observable Markov decision processes. In *ICML-02*, pp. 51–58.
- Bonet**, B. and Geffner, H. (1999). Planning as heuristic search: New results. In *ECP-99*, pp. 360–372.
- Bonet**, B. and Geffner, H. (2000). Planning with incomplete information as heuristic search in belief space. In *ICAPS-00*, pp. 52–61.
- Bonet**, B. and Geffner, H. (2005). An algorithm better than AO^{*}? In *AAAI-05*.
- Boole**, G. (1847). *The Mathematical Analysis of Logic: Being an Essay towards a Calculus of Deductive Reasoning*. Macmillan, Barclay, and Macmillan, Cambridge.
- Booth**, T. L. (1969). Probabilistic representation of formal languages. In *IEEE Conference Record of the 1969 Tenth Annual Symposium on Switching and Automata Theory*, pp. 74–81.
- Borel**, E. (1921). La théorie du jeu et les équations intégrales à noyau symétrique. *Comptes Rendus Hebdomadaires des Séances de l'Académie des Sciences*, 173, 1304–1308.
- Borenstein**, J., Everett, B., and Feng, L. (1996). *Navigating Mobile Robots: Systems and Techniques*. A. K. Peters, Ltd.
- Borenstein**, J. and Koren, Y. (1991). The vector field histogram—Fast obstacle avoidance for mobile robots. *IEEE Transactions on Robotics and Automation*, 7(3), 278–288.
- Borgida**, A., Brachman, R. J., McGuinness, D., and Alperin Resnick, L. (1989). CLASSIC: A structural data model for objects. *SIGMOD Record*, 18(2), 58–67.
- Boroditsky**, L. (2003). Linguistic relativity. In Nadel, L. (Ed.), *Encyclopedia of Cognitive Science*, pp. 917–921. Macmillan.
- Bošer**, B., Guyon, I., and Vapnik, V. N. (1992). A training algorithm for optimal margin classifiers. In *COLT-92*.
- Bosse**, M., Newman, P., Leonard, J., Soika, M., Feiten, W., and Teller, S. (2004). Simultaneous localization and map building in large-scale cyclic environments using the atlas framework. *Int. J. Robotics Research*, 23(12), 1113–1139.
- Bourzutschky**, M. (2006). 7-man endgames with pawns. *CCRL Discussion Board*, kirill-kryukov.com/chess/discussion-board/viewtopic.php?t=805.
- Boutilier**, C. and Brafman, R. I. (2001). Partial-order planning with concurrent interacting actions. *JAIR*, 14, 105–136.
- Boutilier**, C., Dearden, R., and Goldszmidt, M. (2000). Stochastic dynamic programming with factored representations. *AIJ*, 121, 49–107.
- Boutilier**, C., Reiter, R., and Price, B. (2001). Symbolic dynamic programming for first-order MDPs. In *IJCAI-01*, pp. 467–472.
- Boutilier**, C., Friedman, N., Goldszmidt, M., and Koller, D. (1996). Context-specific independence in Bayesian networks. In *UAI-96*, pp. 115–123.
- Bouzy**, B. and Cazenave, T. (2001). Computer go: An AI oriented survey. *AIJ*, 132(1), 39–103.
- Bowerman**, M. and Levinson, S. (2001). *Language acquisition and conceptual development*. Cambridge University Press.
- Bowling**, M., Johanson, M., Burch, N., and Szafrań, D. (2008). Strategy evaluation in extensive games with importance sampling. In *ICML-08*.
- Box**, G. E. P. (1957). Evolutionary operation: A method of increasing industrial productivity. *Applied Statistics*, 6, 81–101.
- Box**, G. E. P., Jenkins, G., and Reinsel, G. (1994). *Time Series Analysis: Forecasting and Control* (3rd edition). Prentice Hall.
- Boyan**, J. A. (2002). Technical update: Least-squares temporal difference learning. *Machine Learning*, 49(2–3), 233–246.
- Boyan**, J. A. and Moore, A. W. (1998). Learning evaluation functions for global optimization and Boolean satisfiability. In *AAAI-98*.
- Boyd**, S. and Vandenberghe, L. (2004). *Convex Optimization*. Cambridge University Press.
- Boyan**, X., Friedman, N., and Koller, D. (1999). Discovering the hidden structure of complex dynamic systems. In *UAI-99*.
- Boyer**, R. S. and Moore, J. S. (1979). *A Computational Logic*. Academic Press.
- Boyer**, R. S. and Moore, J. S. (1984). Proof checking the RSA public key encryption algorithm. *American Mathematical Monthly*, 91(3), 181–189.