18 LEARNING FROM OBSERVATIONS

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

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

18.1 FORMS OF LEARNING

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

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

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

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

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

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

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

SUPERVISED LEARNING

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

UNSUPERVISED LEARNING

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

REINFORCEMENT LEARNING

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

REINFORCEMENT

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

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

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

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

18.2 Inductive Learning

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

EXAMPLE

PURE INDUCTIVE INFERENCE

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

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

HYPOTHESIS

GENERALIZATION

PROBLEM OF INDUCTION

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

HYPOTHESIS SPACE

CONSISTENT

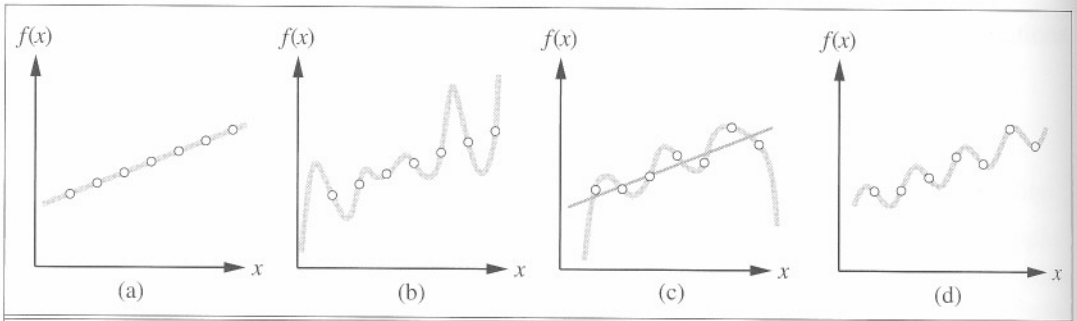

Figure 18.1 (a) Example $(x, f(x))$ pairs and a consistent, linear hypothesis. (b) A consistent, degree-7 polynomial hypothesis for the same data set. (c) A different data set that admits an exact degree-6 polynomial fit or an approximate linear fit. (d) A simple, exact sinusoidal fit to the same data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

OCKHAM'S RAZOR

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

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

For nondeterministic functions, there is an inevitable tradeoff between the complexity of the hypothesis and the degree of fit to the data. Chapter 20 explains how to make this tradeoff using probability theory.

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

REALIZABLE

UNREALIZABLE

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

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

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

18.3 LEARNING DECISION TREES

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

Decision trees as performance elements

DECISION TREE
ATTRIBUTES
CLASSIFICATION
REGRESSION
POSITIVE
NEGATIVE

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

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

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

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

18.5 Why Learning Works: Computational Learning Theory

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

COMPUTATIONAL LEARNING THEORY

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

PROBABLY APPROXIMATELY CORRECT

PAC-LEARNING

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

STATIONARITY

How many examples are needed?

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

- Let $\mathbf{X}$ be the set of all possible examples.
- Let D be the distribution from which examples are drawn.
- Let $\mathbf{H}$ be the set of possible hypotheses.
- Let N be the number of examples in the training set.

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

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

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

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

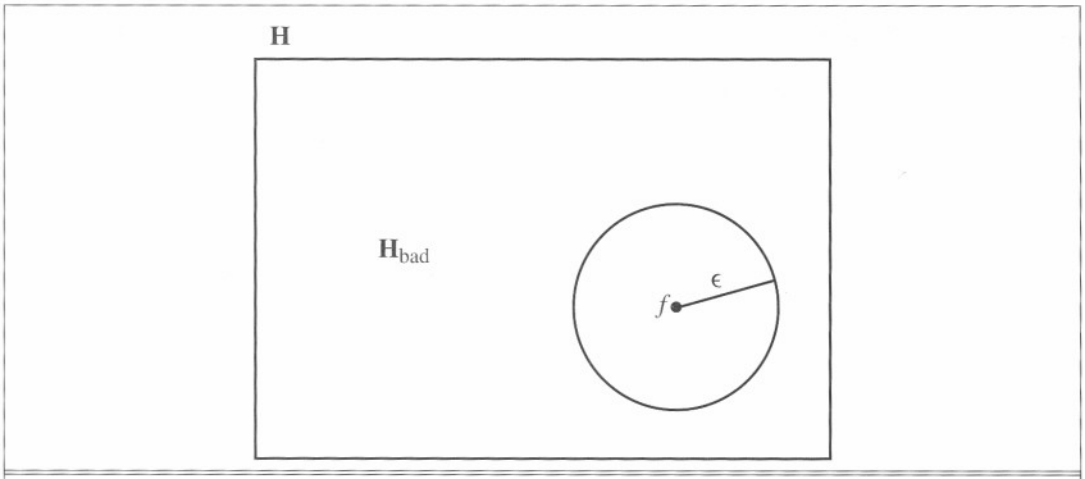

Figure 18.12 Schematic diagram of hypothesis space, showing the "ϵ-ball" around the true function f.

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

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

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

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

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

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

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

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

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

SAMPLE COMPLEXITY

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

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

Learning decision lists

DECISION LIST

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

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

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

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

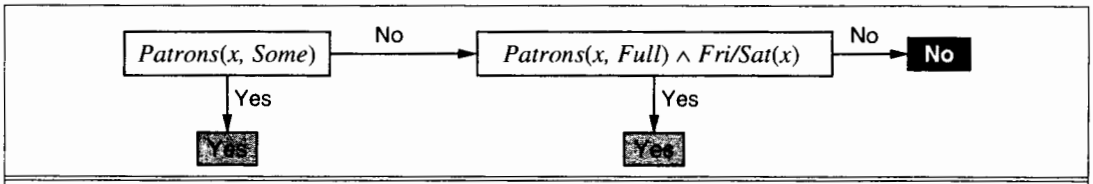

Figure 18.13 A decision list for the restaurant problem.

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

k-DL
k-DT

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

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

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

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

Hence, after some work, we obtain

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

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

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

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

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

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

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

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

Figure 18.14 An algorithm for learning decision lists.

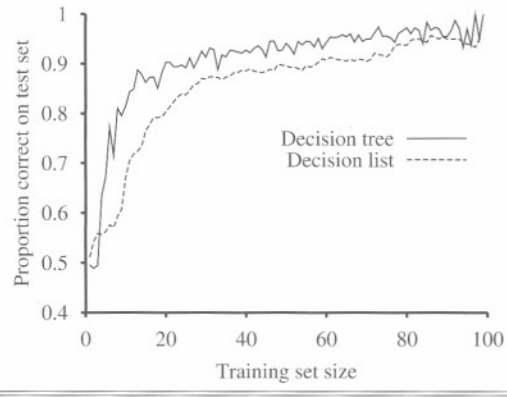

Figure 18.15 Learning curve for DECISION-LIST-LEARNING algorithm on the restaurant data. The curve for DECISION-TREE-LEARNING is shown for comparison.

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

Discussion

IDENTIFICATION IN THE LIMIT

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

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

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

18.6 SUMMARY

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

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

19 KNOWLEDGE IN LEARNING

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

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

PRIOR KNOWLEDGE

19.1 A LOGICAL FORMULATION OF LEARNING

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

Examples and hypotheses

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

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

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

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

$$WillWait(X_1) \ .$$

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

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

CANDIDATE DEFINITION

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

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

EXTENSION

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

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

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

FALSE NEGATIVE

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

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

FALSE POSITIVE

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

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

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

Current-best-hypothesis search

CURRENT-BEST-HYPOTHESIS

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

Suppose we have some hypothesis such as H_r, of which we have grown quite fond. As long as each new example is consistent, we need do nothing. Then along comes a false negative example, X_{13}. What do we do? Figure 19.1(a) shows H_r schematically as a region: everything inside the rectangle is part of the extension of H_r. The examples that have actually been seen so far are shown as "+" or "–", and we see that H_r correctly categorizes all the examples as positive or negative examples of $WillWait$. In Figure 19.1(b), a new example (circled) is a false negative: the hypothesis says it should be negative but it is actually positive.

GENERALIZATION

The extension of the hypothesis must be increased to include it. This is called **generalization**; one possible generalization is shown in Figure 19.1(c). Then in Figure 19.1(d), we see a false positive: the hypothesis says the new example (circled) should be positive, but it actually is negative. The extension of the hypothesis must be decreased to exclude the example. This is

SPECIALIZATION

called **specialization**; in Figure 19.1(e) we see one possible specialization of the hypothesis.

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

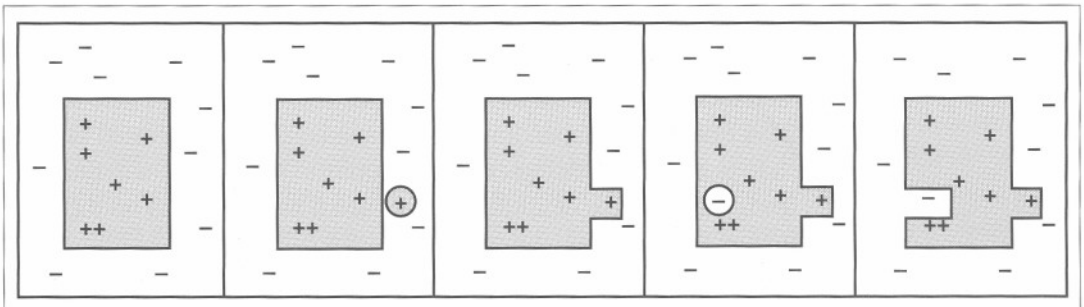

Figure 19.1 (a) A consistent hypothesis. (b) A false negative. (c) The hypothesis is generalized. (d) A false positive. (e) The hypothesis is specialized.

The "more general than" and "more specific than" relations between hypotheses provide the logical structure on the hypothesis space that makes efficient search possible.

We can now specify the CURRENT-BEST-LEARNING algorithm, shown in Figure 19.2. Notice that each time we consider generalizing or specializing the hypothesis, we must check for consistency with the other examples, because an arbitrary increase/decrease in the extension might include/exclude previously seen negative/positive examples.

```
function CURRENT-BEST-LEARNING(examples) returns a hypothesis

    H ← any hypothesis consistent with the first example in examples
    for each remaining example in examples do
        if e is false positive for H then
            H ← choose a specialization of H consistent with examples
        else if e is false negative for H then
            H ← choose a generalization of H consistent with examples
        if no consistent specialization/generalization can be found then fail
    return H
```

Figure 19.2 The current-best-hypothesis learning algorithm. It searches for a consistent hypothesis and backtracks when no consistent specialization/generalization can be found.

We have defined generalization and specialization as operations that change the *extension* of a hypothesis. Now we need to determine exactly how they can be implemented as syntactic operations that change the candidate definition associated with the hypothesis, so that a program can carry them out. This is done by first noting that generalization and specialization are also *logical* relationships between hypotheses. If hypothesis H_1, with definition C_1, is a generalization of hypothesis H_2 with definition C_2, then we must have

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

Therefore in order to construct a generalization of H_2, we simply need to find a definition C_1 that is logically implied by C_2. This is easily done. For example, if $C_2(x)$ is

DROPPING CONDITIONS

$Alternate(x) \wedge Patrons(x, Some)$, then one possible generalization is given by $C_1(x) \equiv Patrons(x, Some)$. This is called **dropping conditions**. Intuitively, it generates a weaker definition and therefore allows a larger set of positive examples. There are a number of other generalization operations, depending on the language being operated on. Similarly, we can specialize a hypothesis by adding extra conditions to its candidate definition or by removing disjuncts from a disjunctive definition. Let us see how this works on the restaurant example, using the data in Figure 18.3.

- The first example X_1 is positive. $Alternate(X_1)$ is true, so let the initial hypothesis be

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

- The second example X_2 is negative. H_1 predicts it to be positive, so it is a false positive. Therefore, we need to specialize H_1. This can be done by adding an extra condition that will rule out X_2. One possibility is

$$H_2 : \ \forall x \ \ WillWait(x) \Leftrightarrow Alternate(x) \wedge Patrons(x, Some) \ .$$

- The third example X_3 is positive. H_2 predicts it to be negative, so it is a false negative. Therefore, we need to generalize H_2. We drop the $Alternate$ condition, yielding

$$H_3 : \ \forall x \ \ WillWait(x) \Leftrightarrow Patrons(x, Some) \ .$$

- The fourth example X_4 is positive. H_3 predicts it to be negative, so it is a false negative. We therefore need to generalize H_3. We cannot drop the $Patrons$ condition, because that would yield an all-inclusive hypothesis that would be inconsistent with X_2. One possibility is to add a disjunct:

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

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

$$H_4' : \ \forall x \ \ WillWait(x) \Leftrightarrow \neg WaitEstimate(x, 30\text{-}60) \ .$$

$$H_4'' : \ \forall x \ \ WillWait(x) \Leftrightarrow Patrons(x, Some) \vee (Patrons(x, Full) \wedge WaitEstimate(x, 10\text{-}30)) \ .$$

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 choices that are made will not necessarily lead to the simplest hypothesis, and may lead to an unrecoverable situation where no simple modification of the hypothesis is consistent with all of the data. In such cases, the program must backtrack to a previous choice point.

The CURRENT-BEST-LEARNING algorithm and its variants have been used in many machine learning systems, starting with Patrick Winston's (1970) "arch-learning" program. With a large number of instances and a large space, however, some difficulties arise:

1. Checking all the previous instances over again for each modification is very expensive.
2. The search process may involve a great deal of backtracking. As we saw in Chapter 18, hypothesis space can be a doubly exponentially large place.

Least-commitment search

Backtracking arises because the current-best-hypothesis approach has to *choose* a particular hypothesis as its best guess even though it does not have enough data yet to be sure of the choice. What we can do instead is to keep around all and only those hypotheses that are consistent with all the data so far. Each new instance will either have no effect or will get rid of some of the hypotheses. Recall that the original hypothesis space can be viewed as a disjunctive sentence

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

As various hypotheses are found to be inconsistent with the examples, this disjunction shrinks, retaining only those hypotheses not ruled out. Assuming that the original hypothesis space does in fact contain the right answer, the reduced disjunction must still contain the right answer because only incorrect hypotheses have been removed. The set of hypotheses remaining is called the **version space**, and the learning algorithm (sketched in Figure 19.3) is called the version space learning algorithm (also the **candidate elimination** algorithm).

VERSION SPACE

CANDIDATE ELIMINATION

function VERSION-SPACE-LEARNING($examples$) **returns** a version space
  local variables: V, the version space: the set of all hypotheses

  $V \leftarrow$ 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 the $examples$.

One important property of this approach is that it is *incremental*: one never has to go back and reexamine the old examples. All remaining hypotheses are guaranteed to be consistent with them anyway. It is also a **least-commitment** algorithm because it makes no arbitrary choices (cf. the partial-order planning algorithm in Chapter 11). But there is an obvious problem. We already said that the hypothesis space is enormous, so how can we possibly write down this enormous disjunction?

The following simple analogy is very helpful. How do you represent all the real numbers between 1 and 2? After all, there is an infinite number of them! The answer is to use an interval representation that just specifies the boundaries of the set: [1,2]. It works because we have an *ordering* on the real numbers.

We also have an ordering on the hypothesis space, namely, generalization/specialization. This is a partial ordering, which means that each boundary will not be a point but rather a set of hypotheses called a **boundary set**. The great thing is that we can represent the entire

BOUNDARY SET

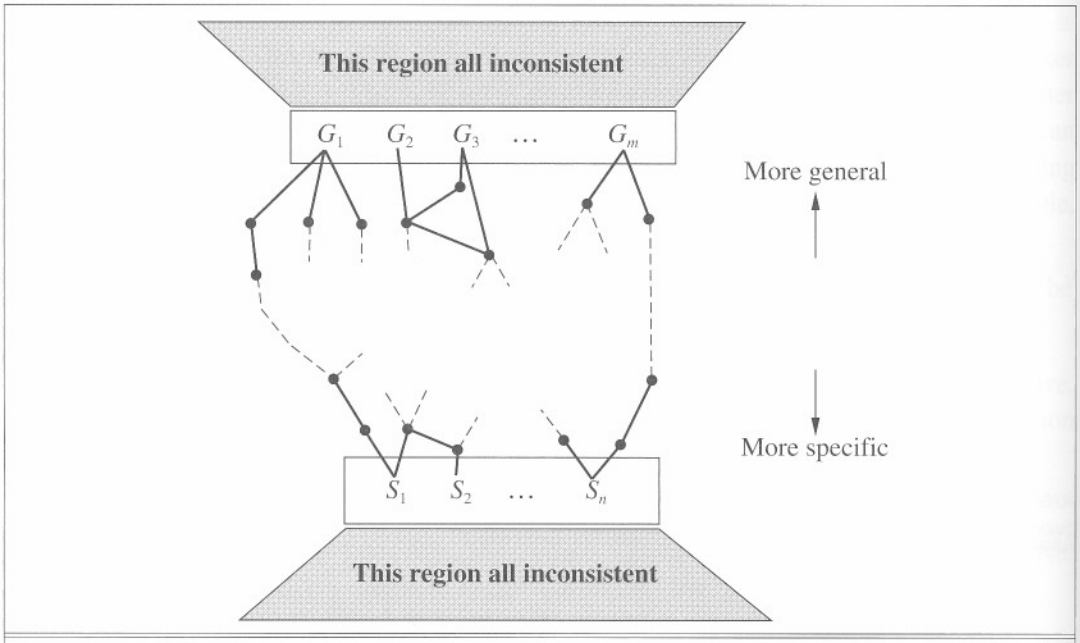

Figure 19.4 The version space contains all hypotheses consistent with the examples.

G-SET
S-SET

version space using just two boundary sets: a most general boundary (the **G-set**) and a most specific boundary (the **S-set**). *Everything in between is guaranteed to be consistent with the examples*. Before we prove this, let us recap:

- The current version space is the set of hypotheses consistent with all the examples so far. It is represented by the S-set and G-set, each of which is a set of hypotheses.
- Every member of the S-set is consistent with all observations so far, and there are no consistent hypotheses that are more specific.
- Every member of the G-set is consistent with all observations so far, and there are no consistent hypotheses that are more general.

We want the initial version space (before any examples have been seen) to represent all possible hypotheses. We do this by setting the G-set to contain *True* (the hypothesis that contains everything), and the S-set to contain *False* (the hypothesis whose extension is empty).

Figure 19.4 shows the general structure of the boundary set representation of the version space. To show that the representation is sufficient, we need the following two properties:

1. Every consistent hypothesis (other than those in the boundary sets) is more specific than some member of the G-set, and more general than some member of the S-set. (That is, there are no "stragglers" left outside.) This follows directly from the definitions of S and G. If there were a straggler h, then it would have to be no more specific than any member of G, in which case it belongs in G; or no more general than any member of S, in which case it belongs in S.
2. Every hypothesis more specific than some member of the G-set and more general than some member of the S-set is a consistent hypothesis. (That is, there are no "holes" be-

tween the boundaries.) Any h between S and G must reject all the negative examples rejected by each member of G (because it is more specific), and must accept all the positive examples accepted by any member of S (because it is more general). Thus, h must agree with all the examples, and therefore cannot be inconsistent. Figure 19.5 shows the situation: there are no known examples outside S but inside G, so any hypothesis in the gap must be consistent.

We have therefore shown that *if* S and G are maintained according to their definitions, then they provide a satisfactory representation of the version space. The only remaining problem is how to *update* S and G for a new example (the job of the VERSION-SPACE-UPDATE function). This may appear rather complicated at first, but from the definitions and with the help of Figure 19.4, it is not too hard to reconstruct the algorithm.

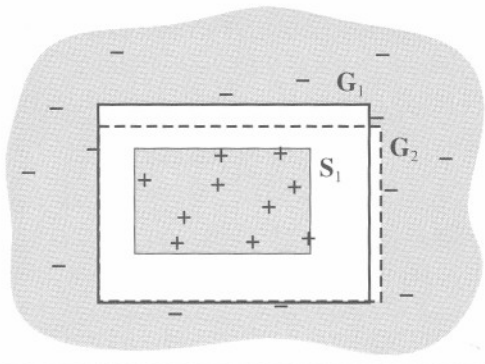

Figure 19.5 The extensions of the members of G and S. No known examples lie in between the two sets of boundaries.

We need to worry about the members S_i and G_i of the S- and G-sets. For each one, the new instance may be a false positive or a false negative.

1. False positive for S_i: This means S_i is too general, but there are no consistent specializations of S_i (by definition), so we throw it out of the S-set.
2. False negative for S_i: This means S_i is too specific, so we replace it by all its immediate generalizations, provided they are more specific than some member of G.
3. False positive for G_i: This means G_i is too general, so we replace it by all its immediate specializations, provided they are more general than some member of S.
4. False negative for G_i: This means G_i is too specific, but there are no consistent generalizations of G_i (by definition) so we throw it out of the G-set.

We continue these operations for each new instance until one of three things happens:

1. We have exactly one concept left in the version space, in which case we return it as the unique hypothesis.
2. The version space *collapses*—either S or G becomes empty, indicating that there are no consistent hypotheses for the training set. This is the same case as the failure of the simple version of the decision tree algorithm.

3. We run out of examples with several hypotheses remaining in the version space. This means the version space represents a disjunction of hypotheses. For any new example, if all the disjuncts agree, then we can return their classification of the example. If they disagree, one possibility is to take the majority vote.

We leave as an exercise the application of the VERSION-SPACE-LEARNING algorithm to the restaurant data.

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

- If the domain contains noise or insufficient attributes for exact classification, the version space will always collapse.
- If we allow unlimited disjunction in the hypothesis space, the S-set will always contain a single most-specific hypothesis, namely, the disjunction of the descriptions of the positive examples seen to date. Similarly, the G-set will contain just the negation of the disjunction of the descriptions of the negative examples.
- For some hypothesis spaces, the number of elements in the S-set of G-set may grow exponentially in the number of attributes, even though efficient learning algorithms exist for those hypothesis spaces.

GENERALIZATION HIERARCHY

To date, no completely successful solution has been found for the problem of noise. The problem of disjunction can be addressed by allowing limited forms of disjunction or by including a **generalization hierarchy** of more general predicates. For example, instead of using the disjunction $WaitEstimate(x, 30\text{-}60) \vee WaitEstimate(x, >60)$, we might use the single literal $LongWait(x)$. The set of generalization and specialization operations can be easily extended to handle this.

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

19.2 KNOWLEDGE IN LEARNING

The preceding section described the simplest setting for inductive learning. To understand the role of prior knowledge, we need to talk about the logical relationships among hypotheses, example descriptions, and classifications. Let *Descriptions* denote the conjunction of all the example descriptions in the training set, and let *Classifications* denote the conjunction of all the example classifications. Then a *Hypothesis* that "explains the observations" must satisfy

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

$$Hypothesis \wedge Descriptions \models Classifications \,. \tag{19.3}$$

ENTAILMENT CONSTRAINT

We call this kind of relationship an **entailment constraint**, in which *Hypothesis* is the "unknown." Pure inductive learning means solving this constraint, where *Hypothesis* is drawn from some predefined hypothesis space. For example, if we consider a decision tree as a logical formula (see Equation (19.1) on page 679), then a decision tree that is consistent with all the examples will satisfy Equation (19.3). If we place *no* restrictions on the logical form of the hypothesis, of course, then $Hypothesis = Classifications$ also satisfies the constraint. Ockham's razor tells us to prefer *small*, consistent hypotheses, so we try to do better than simply memorizing the examples.

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

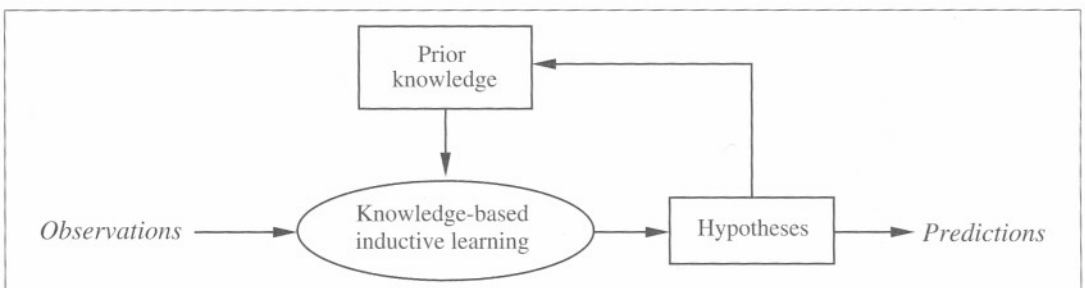

Figure 19.6 A cumulative learning process uses, and adds to, its stock of background knowledge over time.

If we want to build an autonomous learning agent that uses background knowledge, the agent must have some method for obtaining the background knowledge in the first place, in order for it to be used in the new learning episodes. This method must itself be a learning process. The agent's life history will therefore be characterized by *cumulative*, or *incremental*, development. Presumably, the agent could start out with nothing, performing inductions *in vacuo* like a good little pure induction program. But once it has eaten from the Tree of Knowledge, it can no longer pursue such naive speculations and should use its background knowledge to learn more and more effectively. The question is then how to actually do this.

Some simple examples

Let us consider some commonsense examples of learning with background knowledge. Many apparently rational cases of inferential behavior in the face of observations clearly do not follow the simple principles of pure induction.

- Sometimes one leaps to general conclusions after only one observation. Gary Larson once drew a cartoon in which a bespectacled caveman, Zog, is roasting his lizard on

the end of a pointed stick. He is watched by an amazed crowd of his less intellectual contemporaries, who have been using their bare hands to hold their victuals over the fire. This enlightening experience is enough to convince the watchers of a general principle of painless cooking.

- Or consider the case of the traveller to Brazil meeting her first Brazilian. On hearing him speak Portuguese, she immediately concludes that Brazilians speak Portuguese, yet on discovering that his name is Fernando, she does not conclude that all Brazilians are called Fernando. Similar examples appear in science. For example, when a freshman physics student measures the density and conductance of a sample of copper at a particular temperature, she is quite confident in generalizing those values to all pieces of copper. Yet when she measures its mass, she does not even consider the hypothesis that all pieces of copper have that mass. On the other hand, it would be quite reasonable to make such a generalization over all pennies.
- Finally, consider the case of a pharmacologically ignorant but diagnostically sophisticated medical student observing a consulting session between a patient and an expert internist. After a series of questions and answers, the expert tells the patient to take a course of a particular antibiotic. The medical student infers the general rule that that particular antibiotic is effective for a particular type of infection.

These are all cases in which *the use of background knowledge allows much faster learning than one might expect from a pure induction program.*

Some general schemes

In each of the preceding examples, one can appeal to prior knowledge to try to justify the generalizations chosen. We will now look at what kinds of entailment constraints are operating in each case. The constraints will involve the *Background* knowledge, in addition to the *Hypothesis* and the observed *Descriptions* and *Classifications*.

In the case of lizard toasting, the cavemen generalize by *explaining* the success of the pointed stick: it supports the lizard while keeping the hand away from the fire. From this explanation, they can infer a general rule: that any long, rigid, sharp object can be used to toast small, soft-bodied edibles. This kind of generalization process has been called **explanation-based learning**, or **EBL**. Notice that the general rule *follows logically* from the background knowledge possessed by the cavemen. Hence, the entailment constraints satisfied by EBL are the following:

EXPLANATION-BASED LEARNING

$$Hypothesis \wedge Descriptions \models Classifications$$
$$Background \models Hypothesis\ .$$

Because EBL uses Equation (19.3), it was initially thought to be a better way to learn from examples. But because it requires that the background knowledge be sufficient to explain the *Hypothesis*, which in turn explains the observations, *the agent does not actually learn anything factually new from the instance.* The agent *could have* derived the example from what it already knew, although that might have required an unreasonable amount of computation. EBL is now viewed as a method for converting first-principles theories into useful, special-purpose knowledge. We describe algorithms for EBL in Section 19.3.

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

RELEVANCE

In each case, the prior knowledge *Background* concerns the **relevance** of a set of features to the goal predicate. This knowledge, *together with the observations*, allows the agent to infer a new, general rule that explains the observations:

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

RELEVANCE-BASED LEARNING

We call this kind of generalization **relevance-based learning**, or **RBL** (although the name is not standard). Notice that whereas RBL does make use of the content of the observations, it does not produce hypotheses that go beyond the logical content of the background knowledge and the observations. It is a *deductive* form of learning and cannot by itself account for the creation of new knowledge starting from scratch.

In the case of the medical student watching the expert, we assume that the student's prior knowledge is sufficient to infer the patient's disease D from the symptoms. This is not, however, enough to explain the fact that the doctor prescribes a particular medicine M. The student needs to propose another rule, namely, that M generally is effective against D. Given this rule and the student's prior knowledge, the student can now explain why the expert prescribes M in this particular case. We can generalize this example to come up with the entailment constraint:

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

That is, *the background knowledge and the new hypothesis combine to explain the examples.* As with pure inductive learning, the learning algorithm should propose hypotheses that are as simple as possible, consistent with this constraint. Algorithms that satisfy constraint (19.5) are called **knowledge-based inductive learning**, or **KBIL**, algorithms.

KNOWLEDGE-BASED INDUCTIVE LEARNING

KBIL algorithms, which are described in detail in Section 19.5, have been studied mainly in the field of **inductive logic programming**, or **ILP**. In ILP systems, prior knowledge plays two key roles in reducing the complexity of learning:

INDUCTIVE LOGIC PROGRAMMING

1. Because any hypothesis generated must be consistent with the prior knowledge as well as with the new observations, the effective hypothesis space size is reduced to include only those theories that are consistent with what is already known.
2. For any given set of observations, the size of the hypothesis required to construct an explanation for the observations can be much reduced, because the prior knowledge will be available to help out the new rules in explaining the observations. The smaller the hypothesis, the easier it is to find.

In addition to allowing the use of prior knowledge in induction, ILP systems can formulate hypotheses in general first-order logic, rather than in the restricted attribute-based language of Chapter 18. This means that they can learn in environments that cannot be understood by simpler systems.

19.3 EXPLANATION-BASED LEARNING

As we explained in the introduction to this chapter, explanation-based learning is a method for extracting general rules from individual observations. As an example, consider the problem of differentiating and simplifying algebraic expressions (Exercise 9.15). If we differentiate an expression such as X^2 with respect to X, we obtain $2X$. (Notice that we use a capital letter for the arithmetic unknown X, to distinguish it from the logical variable x.) In a logical reasoning system, the goal might be expressed as $\text{ASK}(Derivative(X^2, X) = d,\ KB)$, with solution $d = 2X$.

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

MEMOIZATION

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

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

If the knowledge base contains such a rule, then any new case that is an instance of this rule can be solved immediately.

This is, of course, merely a trivial example of a very general phenomenon. Once something is understood, it can be generalized and reused in other circumstances. It becomes an "obvious" step and can then be used as a building block in solving problems still more complex. Alfred North Whitehead (1911), co-author with Bertrand Russell of *Principia Mathe-*

[2] Of course, a general rule for u^n can also be produced, but the current example suffices to make the point.